Все о Perl 6

Справочное руководство

Джонатан Вортингтон

Патрик Мишо

Карл Мэсак

Мориц Ленц

Скот Даф, Джонатан

Все о Perl 6 : Справочное руководство

Джонатан Вортингтон
Патрик Мишо
Карл Мэсак
Мориц Ленц
Скот Даф, Джонатан

Аннотация

Данная книга является сборником статей о Perl 6.

Содержание

Список таблиц

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.

Александр Загацкий

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.

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 .

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 = B<%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.). 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 результата

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".

Глава 4. Подпрограммы и сигнатуры

Подпрограмма представляет собой участок кода, выполняющий определенную задачу. Она может оперировать передаваемыми ей при вызове данными *(аргументами)* и может производить результаты *(возвращаемые значения)*. *Сигнатурой* подпрограммы является описание всех передаваемых при вызове аргументов и любых возвращаемых значений.

Первая глава демонстрирует простые подпрограммы. Операторы, описанные во второй главе, являются подпрограммами, которые Perl 6 обрабатывает необычным способом. Однако, они будут описаны поверхностно насколько это возможно.

Определение подпрограмм

Определение подпрограммы состоит из нескольких частей. Сперва следует декларатор `sub`, указывающий начало определения подпрограммы. Затем - необязательное имя и необязательная сигнатура. И наконец - тело подпрограммы: ограниченный фигурными скобками блок кода. Этот код выполняется каждый раз при вызове подпрограммы.

К примеру, в коде:

```
sub panic() {
    say "Oh no! Something has gone most terribly wrong!";
}
```

... определена подпрограмма с именем `panic`. Ее сигнатура отсутствует, а тело состоит их единственного оператора `say`.

По умолчанию, подпрограммы ограничена областью лексической видимости, как и любая переменная объявленная с помощью `my`. Это подразумевает, что подпрограмма может быть вызвана, только в границах той области видимости *(Как правило это блок кода)*, внутри которой она была определена. Чтобы подпрограмма стала доступной внутри всего пакета используется декларатор *(ключевое слово)* `our`:

```
{
    our sub eat() {
        say "om nom nom";
    }

    sub drink() {
        say "glug glug";
    }
}

eat();    # om nom nom
```

```
drink();  # fails, can't drink outside of the block
```

`our` также делает подпрограмму видимой вне пакета или модуля:

```
module EatAndDrink {
     our sub eat() {
         say "om nom nom";
     }

     sub drink() {
         say "glug glug";
     }
}
EatAndDrink::eat();    # om nom nom
EatAndDrink::drink();  # fails, not declared with "our"
```

Чтобы подпрограмма стала доступна в другой области видимости, используется экспорт (см. *Экспортирование*).

Подпрограммы в Perl 6 представляют собой объекты. Их можно передавать и хранить в составе структур данных, а так производить те же действия, что и по отношению к другим данным. Дизайнеры языков программирования часто называют их *подпрограммами первого класса*. Они являются такими же основополагающими для использования в языке, как и хэши или массивы.

Подпрограммы первого класса позволяют решать сложные задачи. Например, для создания небольшого ASCII рисунка с изображением танцующих фигур, возможно построение хэша, ключами которого будут названия движений в танце, а значениями - анонимные подпрограммы. Допустим, что пользователи могут вводить названия списки движений *(возможно с коврика для танцев или другого экзотического устройства)*. Как можно организовать легко изменяемый список движений, а также возможно безопасно сделать проверку вводимых пользователем названий движений на предмет допустимых ? Возможно следующая структура программы станет отправной точкой для достижения результата:

```
my $dance = '';
my %moves =
    hands-over-head => sub { $dance ~= '/o\ '   },
    bird-arms       => sub { $dance ~= '|/o\| ' },
    left            => sub { $dance ~= '>o '    },
    right           => sub { $dance ~= 'o< '    },
    arms-up         => sub { $dance ~= '\o/ '   };

my @awesome-dance = <arms-up bird-arms right hands-over-head>;

for @awesome-dance -> $move {
    %moves{$move}.();
}

say $dance;
```

На основании вывода этой программы, вы сможете убедиться что танец YMCA [1] также плохо выглядит в ASCII виде, как и в реальной жизни.

Добавление сигнатур

Сигнатура подпрограммы решает две задачи. Во первых, она объявляет список обязательных и необязательных аргументов, передаваемых при вызове подпрограммы. Во вторых, с помощью сигнатуры объявляются переменные и их связь с аргументами подпрограммы. Эти переменные называются *параметрами*. Сигнатуры в Perl 6 обладают дополнительными возможностями: они позволяют ограничивать значения аргументов, сравнивать и извлекать сложные структуры данных.

Основы

В своей простой форме сигнатура - список разделенных запятой имен переменных, с которыми связываются входные аргументы подпрограммы.

```
sub order-beer($type, $pints) {
    say ($pints == 1 ?? 'A pint' !! "$pints pints") ~ " of $type, please."
}

order-beer('Hobgoblin', 1);    # A pint of Hobgoblin, please.
order-beer('Zlatц+ Bae+ant', 3); # 3 pints of Zlatц+ Bae+ant, please.
```

Использование термина *связываются* вместо *присваиваются* весьма существенно. Переменные в сигнатуре являются ссылками в режиме "чтения" на передаваемые подпрограмме аргументы. Это делает недоступными для модификаций входные значения.

Связывание в режиме "только чтение" можно отменить. Если пометить параметр атрибутом `is rw`, то передаваемое значение можно будет изменять. Эти изменения будут применены также к оригинальным данным, передаваемым при вызове подпрограммы. В случае, если будет передан литерал или другое константное значение для `rw` параметра, то связывание завершиться ошибкой в месте вызова подпрограммы, вызвав программное исключение:

```
sub make-it-more-so($it is rw) {
    $it ~= substr($it, $it.chars - 1) x 5;
}

my $happy = "yay!";
make-it-more-so($happy);
say $happy;                # yay!!!!!!
make-it-more-so("uh-oh");  # Fails; can't modify a constant
```

Также возможно создание копии передаваемых значений с помощью `is copy`. В таком случае, данные вне подпрограммы будут защищены от модификаций, а внутри подпрограммы могут быть изменены:

[1] Возможно имеется в виду Y.M.C.A. [http://en.wikipedia.org/wiki/Y.M.C.A._%28song%29]

```
sub say-it-one-higher($it is copy) {
    $it++;
    say $it;
}

my $unanswer = 41;
say-it-one-higher($unanswer);  # 42
say-it-one-higher(41);         # 42
```

Столь подробная маркировка изменяемых параметров может показать чрезмерной, но скорее всего вы не будете использовать эти модификаторы часто. В то время как некоторые языки требуют пометки параметров `rw` для эмуляции возврата множественных результатов, Perl 6 позволяет напрямую возвращать несколько значений в ответе без подобных фокусов.

Передача массивов, хэшей и кода

Сигил переменной указывает на ее предназначение. В сигнатуре, сигил переменной ограничивает типы передаваемых аргументов. Например, сигил `@` определяет проверку передаваемых значений на соответствие типу `Positional` *(Позиционный)*, который включает в себя типы наподобие `Array` *(массивов)* и списков. При передаче параметров нарушающих это ограничение на экран будет выведено сообщение об ошибке.

```
sub shout-them(@words) {
    for @words -> $w {
        print uc("$w ");
    }
}

my @last_words = <do not want>;
shout-them(@last_words);  # DO NOT WANT
shout-them('help');       # Fails; a string is not Positional
```

Соответственно, сигил `%` указывает, что ожидается нечто `Associative` *(Ассоциативное)*, т.е. что-то позволяющее индексирование с помощью операторов `<...>` или `{...}`. В свою очередь сигил `&` требует указания чего-то вызываемого, например анонимной подпрограммы. В таком случае производить вызов этого параметра можно без указания сигила `&`:

```
sub do-it-lots(&it, $how-many-times) {
    for 1..$how-many-times {
        it();
    }
}

do-it-lots(sub { say "Eating a stroopwafel" }, 10);
```

Скаляр (сигил `&`) не имеет ограничений. Что угодно может быть связано с ним, даже если оно может связываться с другими сигилами.

Интерполяция массивов и хэшей

Иногда требуется заполнить позиционные аргументы значениями из массива. Вместо написания `eat(@food[0], @food[1], @food[2], ...)` и так далее, вы можете линеаризовать *(flatten)* его в список аргументов предварив вертикальной чертой: `eat(|@food)`.

Кроме того, можно интерполировать хэши в именованные аргументы:

```
sub order-shrimps($count, $from) {
    say "I'd like $count pieces of shrimp from the $from, please";
}

my %user-preferences = ( from => 'Northern Sea' );
order-shrimps(3, |%user-preferences)
```

Необязатльные параметры

Иногда аргументы могут быть необязательными. Например, достаточно других параметров с их значениями по умолчанию. В таких случаях подобные необязательные параметры можно пометить как опциональные. При вызовах таких подпрограмм появляется выбор в наборе передаваемых аргументов.

Либо присвоить значение параметра по умолчанию в сигнатуре :

```
sub order-steak($how = 'medium') {
    say "I'd like a steak, $how";
}

order-steak();
order-steak('well done');
```

... или добавить знак вопроса к имени параметра. В последнем случае параметр получает неопределенное значение, если аргумент не передан:

```
sub order-burger($type, $side?) {
   say "I'd like a $type burger" ~
       ( defined($side) ?? " with a side of $side" !! "" );
}

order-burger("triple bacon", "deep fried onion rings");
```

Именованные параметры

Когда подпрограмма имеет много параметров, зачастую, проще привязывать параметры к имени вместо к их позиции в списке передаваемых аргументов. Как следствие, порядок следования аргументов при вызове становиться неважным:

```
sub order-beer($type, $pints) {
    say ($pints == 1 ?? 'A pint' !! "$pints pints") ~ " of $type, please."
}

order-beer(type => 'Hobgoblin', pints => 1);
# A pint of Hobgoblin, please.

order-beer(pints => 3, type => 'ZlatГ?? BaE??ant');
# 3 pints of ZlatГ?? BaE??ant, please.
```

Возможно определить входной аргумент, который может быть передан только по имени, а не позиционно при вызове. Для этого перед именем параметра указывается двоеточие:

```
sub order-shrimps($count, :$from = 'North Sea') {
    say "I'd like $count pieces of shrimp from the $from, please";
}

order-shrimps(6);                       # takes 'North Sea'
order-shrimps(4, from => 'Atlantic Ocean');
order-shrimps(22, 'Mediterranean Sea'); # not allowed, :$from is named only
```

В отличии от позиционных параметров, именованные являются необязательными по умолчанию. Чтобы сделать именованный параметр обязательным необходимо добавить к имени параметра восклицательный знак `!`.

```
sub design-ice-cream-mixture($base = 'Vanilla', :$name!) {
    say "Creating a new recipe named $name!"
}

design-ice-cream-mixture(name => 'Plain');
design-ice-cream-mixture(base => 'Strawberry chip'); # missing $name
```

Переименование параметров

Так как требуется указывать имена при передаче именованных параметров, то данные имена являются частью общедоступного API подпрограмм. Выбирайте имена осторожно ! Иногда может оказаться полезным отделить имя параметра от имени переменной подпрограммы, с которой он связан:

```
sub announce-time(:dinner($supper) = '8pm') {
    say "We eat dinner at $supper";
}

announce-time(dinner => '9pm');      # We eat dinner at 9pm
```

Параметры могут иметь несколько имен ! Если часть пользователей британцы, а остальные - американцы, то можно написать:

```
sub paint-rectangle(
        :$x      =   0,
        :$y      =   0,
        :$width  = 100,
        :$height =  50,
        :color(:colour($c))) {

   # print a piece of SVG that reprents a rectangle
   say qq[<rect x="$x" y="$y" width="$width" height="$height"
                style="fill: $c" />]
}

# both calls work the same
paint-rectangle :color<Blue>;
paint-rectangle :colour<Blue>;

# of course you can still fill the other options
paint-rectangle :width(30), :height(10), :colour<Blue>;
```

Альтернативный синтаксис Именованных параметров

Именованные параметры на самом деле являются `Парами` (`Pair`, пара ключ - значение). Существует несколько способов описания `Пар`. Отличаются они степенью наглядности, так как каждый вариант предусматривает различные механизмы оформления. Следующие три вызова эквивалентны:

```
announce-time(dinner => '9pm');
announce-time(:dinner('9pm'));
announce-time(:dinner<9pm>);
```

Если передается логическое значение, то достаточно определения ключа:

```
toggle-blender( :enabled); # enables  the blender
toggle-blender(:!enabled); # disables the blender
```

Именованный параметр `:name` без указанного значения подразумевает неявное логическое значение `Bool::True`. Противоположная форма, `:!name`, указывает на неявное значение `Bool::False`.

При создании пары, ключ которой совпадает с именем переменной, возможна следующая форма:

```
my $dinner = '9pm';
```

```
announce-dinner :$dinner;  # same as dinner => $dinner;
```

В следующей таблице приведены возможные формы Пар и их значения.

Таблица 4.1. Формы Пар и их значения

Краткая форма	Полная форма	Описание
`:allowed`	`allowed => Bool::True`	Логичекий флаг
`:!allowed`	`allowed => Bool::False`	Логичекий флаг
`:bev<tea coffee>`	`bev => ('tea', 'coffee')`	Список
`:times[1, 3]`	`times => [1, 3]`	Массив
`:opts{ a => 2 }`	`opts => { a => 2 }`	Хэш
`:$var`	`var => $var`	Скалярная переменная
`:@var`	`var => @var`	Переменная - массив
`:%var`	`var => %var`	Переменная - хэш

Возможно использование любой из указанных форм в любом контексте, где возможно использование объекта `Pair`.Например, для заполнения массива:

```
# TODO: better example
my $black = 12;
my %color-popularities = :$black, :blue(8),
                         red => 18, :white<0>;
# same as
# my %color-popularities =
#       black => 12,
#       blue  => 8,
#       red   => 18,
#       white => 0;
```

И наконец, чтобы передать существующий объект `Pair` в подпрограмму как позиционный параметр *(не именованный)*, необходимо либо заключить его в круглые скобки (`(:$thing)`), либо использовать оператор `=>` с взятой в кавычки левой частью: `"thing" => $thing`.

Последовательность параметров

Когда используются в сигнатуре оба типа параметров, позиционные и именованные, то все позиционные параметры должны быть указаны перед именованными.

```
sub mix(@ingredients, :$name) { ... }    # OK
sub notmix(:$name, @ingredients) { ... } # Error
```

Обязательные позиционные параметры также должны быть указаны перед опциональными *(необязательными)* позиционными. Для именованных параметров нет подобных требований.

Slurpy параметры

В приведенном ранее примере функция `shout-it` ожидала массив в качестве аргумента. Это предотвращало передачу одиночного аргумента. Что бы сделать возможным передачу нескольких позиционных аргументов и даже массивов аргументов, которые будут затем в подпрограмме выравнены *(flatten)* в один аргумент "массив", необходимо предварить имя параметра *slurpy* префиксом (`*`):

```
sub shout-them(*@words) {
    for @words -> $w {
        print uc("$w ");
    }
}

# now you can pass items
shout-them('go');           # GO
shout-them('go', 'home');   # GO HOME

my @words = ('go', 'home');
shout-them(@words);         # still works
```

Slurpy параметр *(параметр с прешествующей его имени звездочкой *)* сохраняет ожидаемые позиционные параметры в массив. Кроме того, `*%hash` захватывает [2] все входящие несвязанные именованные аргументы в хэш.

Slurpy массивы и хши позволяют передавать все позиционные и именованные параметры в другом порядке:

```
sub debug-wrapper(&code, *@positional, *%named) {
    warn "Calling '&code.name()' with arguments "
         ~ "@positional.perl(), %named.perl()\n";
    code(|@positional, |%named);
    warn "... back from '&code.name()'\n";
}

debug-wrapper(&order-shrimps, 4, from => 'Atlantic Ocean');
```

Возвращаемые результаты

Подпрограммы могут также возвращать результаты. Пример, описанный ранее, с танцами в виде ASCII графики упрощается при использовании подпрограмм, возвращающих новые строки:

[2] slurp, переводиться как - хлебать

```
my %moves =
   hands-over-head => sub { return '/o\ '   },
   bird-arms       => sub { return '|/o\| ' },
   left            => sub { return '>o '    },
   right           => sub { return 'o< '    },
   arms-up         => sub { return '\o/ '   };

my @awesome-dance = <arms-up bird-arms right hands-over-head>;

for @awesome-dance -> $move {
    print %moves{$move}.();
}

print "\n";
```

Подпрограммы в Perl могут возвращать несколько результатов:

```
sub menu {
    if rand < 0.5 {
        return ('fish', 'white wine')
    } else {
        return ('steak', 'red wine');
    }
}

my ($food, $beverage) = menu();
```

Если опустить оператор `return`, Perl вернет значение последнего оператора, выполненного внутри подпрограммы. Это упрощает предидущий пример:

```
sub menu {
    if rand < 0.5 {
        'fish', 'white wine'
    } else {
        'steak', 'red wine';
    }
}

my ($food, $beverage) = menu();
```

Будьте осторожны, прежде чем полностью полагаться на это: в случае сложной логики *(условные переходы, несколько точек завершения)* добавление `return` позволит сделать код

нагляднее. В качестве общего правила можно принять следующее утверждение: использование неявного `return` имеет положительный эффект только в простых подпрограммах.

`return` имеет дополнительный эффект при раннем завершении подпрограммы:

```
sub create-world(*%characteristics) {
    my $world = World.new(%characteristics);
    return $world if %characteristics<temporary>;

    save-world($world);
}
```

... в таком случае новая переменная `$world` точно не будет потеряна и будет возвращена в качестве ответа.

Работа с типами

Зачастую подпрограммы не могут осмысленно работать с произвольными входными данными и требуют поддержки определенных методом или свойств от входных параметров. В таких случаях имеет смысл ограничить типы передаваемых параметров, так чтобы передача неверных значений в качестве аргументов подпрограммы приводило к ошибке во время вызова.

Базовые типы

Наиболее простой путь ограничить допустимые передаваемые значения аргументов - указать имя типа перед параметром. Например, подпрограмма, производящая математические операции над своими параметрами, ожидает аргументы с типом `Numeric`:

```
sub mean(Numeric $a, Numeric $b) {
    return ($a + $b) / 2;
}

say mean 2.5, 1.5;
say mean 'some', 'strings';
```

Результат работы будет следующим:

```
2
Nominal type check failed for parameter '$a';
    expected Numeric but got Str instead
```

Если несколько параметров имеют ограничения по типу, то каждый из аргументов должен соответствовать ограничениям параметра, с которым он связан.

Добавление ограничений

Иногда указание имени типа недостаточно для описания требований к аргументу. В таких случаях указывается дополнительное *ограничение* для параметра с помощью блока `where`:

```
sub circle-radius-from-area(Numeric $area where { $area >= 0 }) {
    ($area / pi).sqrt
}

say circle-radius-from-area(3);    # OK
say circle-radius-from-area(-3);   # Error
```

Так как расчет имеет смысл только для неотрицательных чисел, дополнительное ограничение возвращает `True` при соответствии входных значений этому диапазону. Если ограничение возвращает значение "Ложь", проверка завершается ошибкой.

Блок после `where` необязательный. Perl выполняет проверку посредством "умного" сопоставления, используя в качестве аргументов все, что находится после `where`. Таким образом возможно явное указание диапазона:

```
sub set-volume(Numeric $volume where 0..11) {
    say "Turning it up to $volume";
}
```

Для ограничения аргументов существующими ключами хэша:

```
my %in-stock = 'Staropramen' => 8, 'Mori' => 5, 'La Trappe' => 9;

sub order-beer(Str $name where %in-stock) {
    say "Here's your $name";
    %in-stock{$name}--;
    if %in-stock{$name} == 0 {
        say "OH NO! That was the last $name, folks! :'(";
        %in-stock.delete($name);
    }
}
```

Захватывания

В каком-то смысле, сигнатура представляет собой *"коллекцию"* аргументов. Захватывания *(captures)* представляют собой сущности того же уровня. Так же, как вы редко думаете о сигнатуре в целом, сосредотачиваясь на каждом отдельно взятом параметре, так же редко вы должны редко думать о *"захватываниях"*. Однако, Perl 6 позволяет управлять захваты-

ваниями напрямую. Захватывания состоят из позиционных и именованных частей, которые действуют аналогично спискам и хэшам соответственно. Списочная часть содержит позиционные аргументы, а хэш составляющая - именованные аргументы.

создание и использование захватываний

Чтобы создать *захватывание*, используется `\(...)` синтаксис. Подобно массивам и хэшам, можно интерполировать захватывание в один аргумент с помощью `|`:

```
sub act($left, $right, :$action) {
    $action($left, $right);
}

my @tasks = \(39, 3, action => { say $^a + $^b }),
            \(6, 7, action => { say $^a * $^b });

for @tasks -> $task-args {
    act(|$task-args);
}
```

Эта программа создает массив "захватываний", каждое из которых содержит два позиционных аргумента и один именованный. При выполнении цикла по элементам массива выполняется вызов `act`, аргументы которого заполняются содержимым захватывания. Perl 6 позволяет отдельно определить аргументы для вызова и сам вызов. Как следствие, это позволяет использовать одни и те же аргументы для многократных вызовов, а также использовать вызов для разных наборов аргументов. Коду приложения не обязательно знать является ли аргумент позиционным или именованным. В отличии от сигнатур, захватывания работают аналогично ссылкам. Любая переменная, указанная в "захватывании", представлена как *ссылка* на переменную.Таким образом `rw` параметры продолжают работать при использовании "захватываний".

```
my $value     = 7;
my $to-change = \($value);

sub double($x is rw) {
    $x *= 2;
}

sub triple($x is rw) {
    $x *= 3;
}

triple(|$to-change);
double(|$to-change);

say $value; # 42
```

Типы Perl с позиционными и именованными частями встречаются также в других ситуациях. Например, регулярные выражения имеют позиционные и именованные выражения:

объекты типа `Match` представляют собой "захватывания". Также возможно представить XML узлы одним из видов "захватываний", включающих в себя именованные атрибуты и позиционные потомки. Привязка такого узла к функции позволяет использовать единый синтаксис параметров для работы с различными потомками атрибутами.

Захватывания в Сигнатурах

Все вызовы создают захватывания на этапе вызова и разворачивают их в соответствии с сигнатурой внутри вызова [3]. Это позволяет написать сигнатуру, которая свяжет само "захватывание" с переменной. Это особенно полезно при написании процедур, которые делегируют управление другим процедурам с теми же параметрами.

```
sub visit-czechoslovakia(|$plan) {
    warn "Sorry, this country has been deprecated.";
    visit-slovakia(|$plan);
    visit-czech-republic(|$plan);
}
```

Преимущества такого использования в сравнении с сигнатурой вида `:(*@pos, *%named)` заключаются в отсутствии некоторого контекста в аргументах, который может быть преждевременным. Например, если при вызове передаются два массива, они будут линеаризованы в `@pos`. Это значит, что в дальнейшем эти два массива не могут быть восстановлены в оригинальном виде. "Захватывание" сохраняет аргументы из двух массивов, что поможет при связывании их в сигнатурах вызываемых в итоге подпрограмм.

Unpacking

В некоторых случаях требуется работать только с частью массива или хэша. Для этого можно использовать обычный доступ к срезам, а также сигнатурное связывание:

```
sub first-is-largest(@a) {
    my $first = @a.shift;
    # TODO: either explain junctions, or find a
    # concise way to write without them
    return $first >= all(@a);
}

# same thing:
sub first-is-largest(@a) {
    my :($first, *@rest) := \(|@a)
    return $first >= all(@rest);
}
```

Сигнатурное связывание может показаться неуклюжим, но при использовании в основной сигнатуре подпрограммы, открывается истинная сила.

[3] Оптимизатор Perl 6 может, конечно, допускать оптимизации на любом из этих этапов, в зависимости от информации, которая может быть доступна на этапе компиляции.

```
sub first-is-largest([$first, *@rest]) {
    return $first >= all(@rest);
}
```

Скобки в сигнатуре указывают компилятору ожидать списочный аргумент. Вместо привязки к массиву, компилятор *распаковывает* аргументы в последовательность параметров, в данном примере - в скаляр для первого элемента и массив для последующих. Приведенная *подсигнатура* также содержит дополнительное условие: сигнатурное связывание завершится неудачей, если в передаваемом массиве будет меньше двух элементов.

Таким же образом можно распаковать хэш, используя `%(...)` вместо квадратных скобок. В таком случае доступны только именованные параметры, а не позиционные.

```
sub create-world(%(:$temporary, *%characteristics)) {
    my $world = World.new(%characteristics);
    return $world if $temporary;

    save-world($world);
}
```

TODO: come up with a good example # maybe steal something from http://jnthn.net/papers/2010-yapc-eu-signatures.pdf # TODO: generic object unpacking

Карринг

Рассмотрим модуль, предложенный в качестве примера в секции "Необязательные параметры":

```
sub order-burger( $type, $side? ) { ... };
```

Если вы часто используете `order-burger` и в основном вместе с картофелем фри, то наличие процедуры `order-burger-and-fries` может оказаться кстати:

```
sub order-burger-and-fries ( $type ) {
    order-burger( $type, side => 'french fries' );
}
```

Если персональный заказ всегда вегетарианский, то может понадобится процедура `order-the-usual` с необязательным параметром:

```
sub order-the-usual ( $side? ) {
    if ( $side.defined ) {
        order-burger( 'veggie', $side );
    }
    else {
        order-burger( 'veggie' );
    }
}
```

Карринг позволяет создавать сокращения для подобных применений. С помощью карринга создается новая подпрограмма на основе существующей с некоторыми предустановленными параметрами. В Perl 6, карринг создается методом `.assuming`:

```
&order-the-usual        := &order-burger.assuming( 'veggie' );
&order-burger-and-fries := &order-burger.assuming( side => 'french fries' );
```

Новая подпрограммы похожи на другие и поддерживают одни и те же структуры входных параметров.

```
order-the-usual( 'salsa' );
order-the-usual( side => 'broccoli' );

order-burger-and-fries( 'plain' );
order-burger-and-fries( :type<<double-beef>> );
```

Интроспекция/самоанализ

Подпрограммы и их сигнатуры являются объектами. Кроме использования, имеется возможность некоторые их подробности и детали параметров:

```
sub logarithm(Numeric $x, Numeric :$base = exp(1)) {
    log($x) / log($base);
}

my @params = &logarithm.signature.params;
say @params.elems, ' parameters';

for @params {
    say "Name:       ", .name;
    say "  Type:     ", .type;
    say "  named?    ", .named    ?? 'yes' !! 'no';
    say "  slurpy?   ", .slurpy   ?? 'yes' !! 'no';
    say "  optional? ", .optional ?? 'yes' !! 'no';
}
```

```
2 parameters
Name:       $x
  Type:     Numeric()
  named?    no
  slurpy?   no
  optional? no
Name:       $base
  Type:     Numeric()
  named?    yes
  slurpy?   no
  optional? yes
```

Сигил & и следующее за ним имя подпрограммы представляют собой объект соответствующей подпрограммы. `&logarithm.signature` возвращает сигнатуру подпрограммы, а метод `.params` у сигнатуры - список параметров в виде объектов типа `Parameter`. Объекты `Parameter` описывают детали каждого параметра в отдельности

Таблица 4.2. Методы класса Parameter

method	description
name	Имя связанной лексической переменной
type	Номинальный тип
constraints	Все дальнейшие ограничения типа
readonly	"Истина", если параметр `is readonly`
rw	"Истина", если параметр `is rw`
copy	"Истина", если параметр `is copy`
named	"Истина", если параметр должен быть передан как именованный
named_names	Список названий именованных параметров
slurpy	"Истина", если параметр slurpy (захватывает)
optional	"Истина", если параметр необязательный
default	Замыкание возвращающее значение по умолчанию
signature	Вложенная сигнатура для установки привязок аргументов

```
# TODO: talk about &signature.cando once that's implemented
```

Анализ сигнатур позволяет создавать интерфейсы, которые могут анализировать ожидаемые сигнатурами данные, а затем передавать правильные данные в подпрограммы. Например, возможно создание программы-генератора web форм, которая будет создавать интерфейс пользователем, проверять введенные данные и затем обрабатывать их на основе информации полученной с помощью анализа сигнатур. Подобный подход позволяет также облегчить создание инструментов для работы в командной строке, обеспечивая справочную информацию о параметрах ввода.

Link to traits section, которой еще пока нет.

Помимо этого, *черты* позволяют связать с параметрами дополнительные данные. Эти метаданные выходят далеко за границы материала о подпрограммах, сигнатурах и параметрах.

Глава 5. Классы и Объекты

Следующая программа показывает как может выглядеть обработчик зависимостей в Perl 6. Она демонстрирует использоание ползовательских конструкторов, приватных и публичных атрибутов, методов а также некоторые аспекты сигнатур. Кода в примере приведено не много, тем не менее он интересен и местами полезен.

```
class Task {
    has      &!callback;
    has Task @!dependencies;
    has Bool $.done;

    method new(&callback, Task *@dependencies) {
        return self.bless(*, :&callback, :@dependencies);
    }

    method add-dependency(Task $dependency) {
        push @!dependencies, $dependency;
    }

    method perform() {
        unless $!done {
            .perform() for @!dependencies;
            &!callback();
            $!done = True;
        }
    }
}

my $eat =
    Task.new({ say 'eating dinner. NOM!' },
        Task.new({ say 'making dinner' },
            Task.new({ say 'buying food' },
                Task.new({ say 'making some money' }),
                Task.new({ say 'going to the store' })
            ),
            Task.new({ say 'cleaning kitchen' })
        )
    );

$eat.perform();
```

Приступая к изучению классов

Perl 6,как и много других языков, использует клюевое слово `class` для определения нового класса. Следующий затем блок, как и любой другой блок, может содержать произвольный код, однако классы обычно содержат определения состояний и поведений.

Код примера содержит атрибуты (состояния), определяемые с помощью ключевого слова `has`, и поведения, определяемые с помощью `method`.

Определение класса создает *объект-тип*, который по умолчанию становиться доступным внути текущего пакета (аналогично переменным, определенным с помощью `our`). Этот объект-тип является "пустым экземпляром" класса. С ними мы встречались в предидущих главах. Например, каждый из типов `Int` и `Str` относятся к объектам-типам одного из встроенных в Perl 6 классов. Код примера в начале главы демонстрирует, как имя класса `Task` может использоватья в роли ссылки в дальнейшем, например для создания экземпляров класса вызывая метод `new`.

Объекты-типы неопределены в том смысле, что они возвращают значение `False` если вызвать их метод `.defined`. Эту особенность можно использвать чтобы узнать является ли какой-либо объект объектом-типом или нет:

```
my $obj = Int;
if $obj.defined {
    say "Ordinary, defined object";
} else {
    say "Type object";
}
```

Могу ли я обладать состоянием?

В примере, первые три строки в блоке класса:

```
    has      &!callback;
    has Task @!dependencies;
    has Bool $.done;
```

определяют атрибуты (в других языках они могут называться *полями* или *хранилищем экземпляра*). Атрибуты действительно являются индивидульным местом храниения данных для каждого созданного объекта. Так же как переменные созданные с помощью `my` не могут быть доступны извне области их видимости так и атрибуты объектов доступны только внутри класса. Данная особенность является основой объекто-ориентированного проектирования и называется *инкапсуляцией*.

Первая строка среди атрибутов определяет память для хранения callback -- небольшого куска кода. Он будет вызван для выполнения задачи, котороую представляет экземпляр класса `Task`.

```
has &!callback;
```

Сигил & указывает? что аттрибут представляет собой нечно вызываемое *(invocable)*. Символ ! является твигилом *(twigil)*, или выражаясь иначе - вторым сигилом. Твигил является частью имени переменной. В данном случае твигил ! подчеркивает что данный атрибут является приватным *(собсвенным, private)*, то есть недоступен вне класса.

Определение второго атрибута класса `Task` также содержит приватный твигил:

```
has Task @!dependencies;
```

Однако этот атрибут представляет собой массив элементов и поэтому необходим сигил `@`. Каждый из элементов представляет собой задачу, а все вместе очередность задач, которая является условием завершения текущей. Кроме того тип атрибута сообщает, что элементы массива могут быть только экземплярами класса `Task` (или класса, производного от него).

Третий атрибут хранит статус готовности задачи:

```
has Bool $.done;
```

Этот скалярный атрибут (сигил $) имеет тип `Bool`. Вместо твигила ! используется твигил `.`. В то время как инкапсуляция атрибутов является в Perl 6 полноценной, язык позволяет избавиться от необходимости явно создавать методы доступа к атрибутам из вне *(accessor methods)*. Замена ! на . помимо определения атрибута `$!done` определит метод доступа `done`. То есть результат будет тот же, как если вы написали бы следующий код:

```
has Bool $!done;
method done() { return $!done }
```

Обратите внимание, что это отличается от определения публичного атрибута, как это позволяют некоторые языки: и вы действительно получаете недоступную снаружи переменную и метод, без необходимости писать этот метод вручную *(просто заменив ! на .)*. Подобный способ хорош пока вам не понадобится более сложные действия, чем просто возврат значения.

Стоит заметить что твигил . создает метод с доступом к атрибуту только в режиме чтения. Чтобы пользователи этого объекта могли сбросить статус готовности задачи (для выполнения ее например повторнао) можно изменить определение атрибута на следующее:

```
has Bool $.done is rw;
```

traits, is rw

The `is rw` trait causes the generated accessor method to return something external code can modify to change the value of the attribute.

Methods

While attributes give objects state, methods give objects behaviors. Ignore the `new` method temporarily; it's a special type of method. Consider the second method, `add-dependency`, which adds a new task to this task's dependency list.

```
method add-dependency(Task $dependency) {
    push @!dependencies, $dependency;
}
```

In many ways, this looks a lot like a `sub` declaration. However, there are two important differences. First, declaring this routine as a method adds it to the list of methods for the current class. Thus any instance of the `Task` class can call this method with the `.` method call operator. Second, a method places its invocant into the special variable `self`.

The method itself takes the passed parameter--which must be an instance of the `Task` class--and `push`es it onto the invocant's `@!dependencies` attribute.

The second method contains the main logic of the dependency handler:

```
method perform() {
    unless $!done {
        .perform() for @!dependencies;
        &!callback();
        $!done = True;
    }
}
```

It takes no parameters, working instead with the object's attributes. First, it checks if the task has already completed by checking the `$!done` attribute. If so, there's nothing to do.

Otherwise, the method performs all of the task's dependencies, using the `for` construct to iterate over all of the items in the `@!dependencies` attribute. This iteration places each item--each a `Task` object--into the topic variable, `$_`. Using the `.` method call operator without specifying an explicit invocant uses the current topic as the invocant. Thus the iteration construct calls the `.perform()` method on every `Task` object in the `@!dependencies` attribute of the current invocant.

After all of the dependencies have completed, it's time to perform the current `Task`'s task by invoking the `&!callback` attribute directly; this is the purpose of the parentheses. Finally, the method sets the `$!done` attribute to `True`, so that subsequent invocations of `perform` on this object (if this `Task` is a dependency of another `Task`, for example) will not repeat the task.

Constructors

Perl 6 is rather more liberal than many languages in the area of constructors. A constructor is anything that returns an instance of the class. Furthermore, constructors are ordinary methods. You inherit a default constructor named `new` from the base class `Object`, but you are free to override `new`, as this example does:

```
method new(&callback, Task *@dependencies) {
    return self.bless(*, :&callback, :@dependencies);
}
```

The biggest difference between constructors in Perl 6 and constructors in languages such as C# and Java is that rather than setting up state on a somehow already magically created object, Perl 6 constructors actually create the object themselves. This easiest way to do this is by calling the `bless` method, also inherited from `Object`. The `bless` method expects a positional parameter--the so-called "candidate"--and a set of named parameters providing the initial values for each attribute.

The example's constructor turns positional arguments into named arguments, so that the class can provide a nice constructor for its users. The first parameter is the callback (the thing to do to execute the task). The rest of the parameters are dependent `Task` instances. The constructor captures these into the `@dependencies` slurpy array and passes them as named parameters to `bless` (note that `:&callback` uses the name of the variable--minus the sigil--as the name of the parameter).

Consuming our class

After creating a class, you can create instances of the class. Declaring a custom constructor provides a simple way of declaring tasks along with their dependencies. To create a single task with no dependencies, write:

```
my $eat = Task.new({ say 'eating dinner. NOM!' });
```

An earlier section explained that declaring the class `Task` installed a type object had been installed in the namespace. This type object is a kind of "empty instance" of the class, specifically an instance without any state. You can call methods on that instance, as long as they do not try to access any state; `new` is an example, as it creates a new object rather than modifying or accessing an existing object.

Unfortunately, dinner never magically happens. It has dependent tasks:

```
my $eat =
    Task.new({ say 'eating dinner. NOM!' },
```

```
        Task.new({ say 'making dinner' },
            Task.new({ say 'buying food' },
                Task.new({ say 'making some money' }),
                Task.new({ say 'going to the store' })
            ),
            Task.new({ say 'cleaning kitchen' })
        )
    );
```

Notice how the custom constructor and sensible use of whitespace allows a layout which makes task dependencies clear.

Finally, the `perform` method call recursively calls the `perform` method on the various other dependencies in order, giving the output:

```
making some money
going to the store
buying food
cleaning kitchen
making dinner
eating dinner. NOM!
```

Inheritance

Object Oriented Programming provides the concept of inheritance as one of the mechanisms to allow for code reuse. Perl 6 supports the ability for one class to inherit from one or more classes. When a class inherits from another class that informs the method dispatcher to follow the inheritance chain to look for a method to dispatch. This happens both for standard methods defined via the method keyword and for methods generated through other means such as attribute accessors.

```
class Employee {
    has $.salary;

    method pay() {
        say "Here is \$$.salary";
    }

}

class Programmer is Employee {
    has @.known_languages is rw;
    has $.favorite_editor;

    method code_to_solve( $problem ) {
        say "Solving $problem using $.favorite_editor in "
        ~ $.known_languages[0] ~ '.';
    }
}
```

Now any object of type Programmer can make use of the methods and accessors defined in the Employee class as though they were from the Programmer class.

```
my $programmer = Programmer.new(
    salary => 100_000,
    known_languages => <Perl5 Perl6 Erlang C++>,
    favorite_edtor => 'vim'
);

$programmer.code_to_solve('halting problem');
$programmer.pay();
```

Overriding Inherited Methods

Of course, classes can override methods and attributes defined on ancestoral classes by defining their own. The example below demonstrates the Baker class overriding the Cook's cook method.

```
class Cook is Employee {
    has @.utensils  is rw;
    has @.cookbooks is rw;

    method cook( $food ) {
        say "Cooking $food";
    }

    method clean_utensils {
        say "Cleaning $_" for @.utensils;
    }
}

class Baker is Cook {
    method cook( $confection ) {
        say "Baking a tasty $confection";
    }
}

my $cook = Cook.new(
    utensils => (<spoon ladel knife pan>),
    cookbooks => ('The Joy of Cooking'),
    salary => 40000);

$cook.cook( 'pizza' ); # Cooking pizza

my $baker = Baker.new(
    utensils => ('self cleaning oven'),
    cookbooks => ("The Baker's Apprentice"),
    salary => 50000);

$baker.cook('brioche'); # Baking a tasty brioche
```

Because the dispatcher will see the cook method on Baker before it moves up to the parent class the Baker's cook method will be called.

Multiple Inheritance

As mentioned before, a class can inherit from multiple classes. When a class inherits from multiple classes the dispatcher knows to look at both classes when looking up a method to search for. As a side note, Perl 6 uses the C3 algorithm to linearize the multiple inheritance hierarchies, which is a significant improvement over Perl 5's approach to handling multiple inheritance.

```
class GeekCook is Programmer is Cook {
    method new( *%params ) {
        push( %params<cookbooks>, "Cooking for Geeks" );
        return self.bless(%params);
    }
}

my $geek = GeekCook.new(
    books           => ('Learning Perl 6'),
    utensils        => ('blingless pot', 'knife', 'calibrated oven'),
    favorite_editor => 'MacVim',
    known_languages => <Perl6>
);

$geek.cook('pizza');
$geek.code_to_solve('P =? NP');
```

Now all the methods made available by both the Programmer class and the Cook class are available from the GeekCook class.

While multiple inheritance is a useful concept to know and on occasion use, it is important to understand that there are more useful OOP concepts. When reaching for multiple inheritance it is good practice to consider whether the design wouldn't be better realized by using roles. For more information on roles check out the Roles chapter.

Introspection

Introspection is the process of gathering information about some objects in your program, not by reading the source code, but by querying the object (or a controlling object) for some properties, like its type.

Given an object `$p`, and the class definitions from the previous sections, we can ask it a few questions:

```
if $o ~~ Employee { say "It's an employee" };
if $o ~~ GeekCook { say "It's a geeky cook" };
say $o.WHAT;
say $o.perl;
say $o.^methods(:local).join(', ');
```

The output can look like this:

```
It's an employee
Programmer()
Programmer.new(known_languages => ["Perl", "Python", "Pascal"], favorite_edito
code_to_solve, known_languages, favorite_editor
```

The first two tests each smart-match against a class name. If the object is of that class, or of an inheriting class, it returns true. So the object in question is of class `Employee` or one that inherits from it, but not `GeekCook`.

The `.WHAT` method returns the type object associated with the object `$o`, which tells the exact type of `$o`: in this case `Programmer`.

`$o.perl` returns a string that can be executed as Perl code, and reproduces the original object `$o`. While this does not work perfectly in all cases[1], it is very useful for debugging simple objects.

Finally `$o.^methods(:local)` produces a list of methods that can be called on `$o`. The `:local` named argument limits the returned methods to those defined in the `Employee` class, and excludes the inherited methods.

The syntax of calling method with `.^` instead of a single dot means that it is actually a method call on the *meta class*, which is a class managing the properties of the `Employee` class - or any other class you are interested in. This meta class enables other ways of introspection too:

```
say $o.^attributes.join(', ');
say $o.^parents.join(', ');
```

Introspection is very useful for debugging, and for learning the language and new libraries. When a function or method returns an object you don't know about, finding its type with `.WHAT`, a construction recipe for it with `.perl` and so on you'll get a good idea what this return value is. With `.^methods` you can learn what you can do with it.

But there are other applications too: a routine that serializes objects to a bunch of bytes needs to know the attributes of that object, which it can find out via introspection.

Exercises

1. The method `add-dependency` in `Task` permits the creation of *cycles* in the dependency graph. That is, if you follow dependencies, you can eventually return to the original `Task`. Show how to create a graph with cycles and explain why the `perform` method of a `Task` whose dependencies contain a cycle would never terminate successfully.

Answer: You can create two tasks, and then "short-circuit" them with `add-dependency`:

[1] for example closures cannot easily be reproduced this way; if you don't know what a closure is don't worry. Also current implementations have problems with dumping cyclic data structures this way, but they are expected to be handlded correctly by `.perl` at some point.

```
my $a = Task.new({ say 'A' });
my $b = Task.new({ say 'B' }, $a);
$a.add-dependency($b);
```

The `perform` method will never terminate because the first thing the method does is to call all the `perform` methods of its dependencies. Because `$a` and `$b` are dependencies of each other, none of them would ever get around to calling their callbacks. The program will exhaust memory before it ever prints `'A'` or `'B'`.

2. Is there a way to detect the presence of a cycle during the course of a `perform` call? Is there a way to prevent cycles from ever forming through `add-dependency`?

Answer: To detect the presence of a cycle during a `perform` call, keep track of which `Task`s have started; prevent a `Task` from starting twice before finishing:

```
augment class Task {
    has Bool $!started = False;

    method perform() {
        if $!started++ && !$!done {
            die "Cycle detected, aborting";
        }

        unless $!done {
            .perform() for @!dependencies;
            &!callback();
            $!done = True;
        }
    }
}
```

Another approach is to stop cycles from forming during `add-dependency` by checking whether there's already a dependency running in the other direction. (This is the only situation in which a cycle can occur.) This requires the addition of a helper method `depends-on`, which checks whether a task depends on another one, either directly or transitively. Note the use of `б╩` and `[||]` to write succinctly what would otherwise have involved looping over all the dependencies of the `Task`:

```
augment class Task {
    method depends-on(Task $some-task) {
        $some-task === any(@!dependencies)
        [||] @!dependencies>>.depends-on($some-task)
    }

    method add-dependency(Task $dependency) {
        if $dependency.depends-on(self) {
            warn 'Cannot add that task, since it would introduce a cycle.';
            return;
        }
```

```
            push @!dependencies, $dependency;
        }
    }
```

3. How could `Task` objects execute their dependencies in parallel? (Think especially about how to avoid collisions in "diamond dependencies", where a `Task` has two different dependencies which in turn have the same dependency.)

Answer: Enabling parallelism is easy; change the line `.perform() for @!dependencies;` into `@!dependencies».perform()`. However, there may be race conditions in the case of diamond dependencies, wherein `Task`s `A` starts `B` and `C` in parallel, and both start a copy of `D`, making `D` run twice. The solution to this is the same as with the cycle-detection in Question 2: introducing an attribute `$!started`. Note that it's impolite to die if a `Task` has started but not yet finished, because this time it might be due to parallelism rather than cycles:

```
augment class Task {
    has Bool $!started = False;

    method perform() {
        unless $!started++ {
            @!dependencies».perform();
            &!callback();
            $!done = True;
        }
    }
}
```

Глава 6. Multis

Perl usually decides which function to call based on the name of the function or the contents of a function reference. This is simple to understand. Perl can also examine the contents of the arguments provided to decide which of several variants of a function--variants each with the same name--to call. In this case, the amount and types of the function's arguments help to distinguish between multiple variants of a function. This is *multidispatch*, and the functions to which Perl can dispatch in this case are *multis*.

Javascript Object Notation (*JSON*) is a simple data exchange format often used for communicating with web services. It supports arrays, hashes, numbers, strings, boolean values, and `null`, the undefined value.

`JSON::Tiny` is a minimal library used to convert Perl 6 data structures to JSON. See for the other part of that module, which parses JSON and turns it into Perl 6 data structures. The full code, containing additional documentation and tests, is available from http://github.com/moritz/json/. This snippet demonstrates how multis make the code simpler and more obvious:

```
multi to-json(Real $d) { ~$d }
multi to-json(Bool $d) { $d ?? 'true' !! 'false'; }
multi to-json(Str  $d) {
    '"'
    ~ $d.trans(['"',  '\\',   "\b", "\f", "\n", "\r", "\t"]
            => ['\"', '\\\\', '\b', '\f', '\n', '\r', '\t'])
    ~ '"'
}

multi to-json(Array $d) {
    return  '[ '
            ~ $d.values.map({ to-json($_) }).join(', ')
            ~ ' ]';
}

multi to-json(Hash  $d) {
    return '{ '
            ~ $d.pairs.map({ to-json(.key)
            ~ ' : '
            ~ to-json(.value) }).join(', ')
            ~ ' }';
}

multi to-json($d where {!defined $d}) { 'null' }

multi to-json($d) {
    die "Can't serialize an object of type " ~ $d.WHAT.perl
}
```

This code defines a single multi sub named `to-json`, which takes one argument and turns that into a string. `to-json` has many *candidates*; these subs all have the name `to-json` but differ in their signatures. Every candidate resembles:

```
multi to-json(Bool $data) { ... }
multi to-json(Real $data) { ... }
```

Which one is actually called depends on the type of the data passed to the subroutine. A call such as `to-json(Bool::True)` invokes the first candidate. Passing a numeric value of type `Real` instead invokes the second.

The candidate for handling `Real` is very simple; because JSON's and Perl 6's number formats coincide, the JSON converter can rely on Perl's conversion of these numbers to strings. The `Bool` candidate returns a literal string `'true'` or `'false'`.

The `Str` candidate does more work: it wraps its parameter in quotes and escapes literal characters that the JSON spec does not allow in strings--a tab character becomes `\t`, a newline `\n`, and so on.

The `to-json(Array $d)` candidate converts all elements of the array to JSON with recursive calls to `to-json`, joins them with commas, and surrounds them with square brackets. The recursive calls demonstrate a powerful truth of multidispatch: these calls do not necessarily recurse to the `Array` candidate, but dispatch to the appropriate candidate based on the types of *their* arguments.

The candidate that processes hashes turns them into the form `{ "key1" : "value1", "key2" : [ "second", "value" ] }`. It does this again by recursing into `to-json`.

Constraints

Candidates can specify more complex signatures:

```
multi to-json($d where {!defined $d}) { 'null' }
```

This candidate adds two new twists. It contains no type definition, in which case the type of the parameter defaults to `Any`, the root of the normal branch of the type hierarchy. More interestingly, the `where {!defined $d}` clause is a *constraint*, which defines a so-called *subset type*. This candidate will match only *some* values of the type `Any`--those where the value is undefined.

Whenever the compiler performs a type check on the parameter `$d`, it first checks the *nominal* type (here, `Any`). If that check succeeds, it calls the code block. The entire type check can only succeed if the code block returns a true value.

The curly braces for the constraint can contain arbitrary code. You can abuse this to count how often a type check occurs:

```
my $counter = 0;

multi a(Int $x)  { };
multi a($x)      { }
multi a($x where { $counter++; True }) { };

a(3);
say $counter;       # says B<0>
a('str');
say $counter;       # says B<2>
```

This code defines three multis, one of which increases a counter whenever its `where` clause executes. Any Perl 6 compiler is free to optimize away type checks it knows will succeed. In the current Rakudo implementation, the second line with `say` will print a higher number than the first.

In the first call of `a(3)`, the nominal types alone already determine the best candidate match, so the where block never executes and the first `$counter` output is always `0`.

The output after the second call is at least `1`. The compiler has to execute the where-block at least once to check if the third candidate is the best match, but the specification does not require the *minimal* possible number of runs. This is illustrated in the second `$counter` output. The specific implementation used to run this test actually executes the where-block twice. Keep in mind that the number of times the subtype checks blocks execute is specific to any particular implementation of Perl 6.

> Avoid writing code like this in anything other than example code. Relying on the side effects of type checks produces unreliable code.

Narrowness

One candidate remains from the JSON example:

```
multi to-json($d) {
    die "Can't serialize an object of type " ~ $d.WHAT.perl
}
```

With no explicit type or constraint on the parameter `$d`, its type defaults to `Any`--and thus it matches any passed object. The body of this function complains that it doesn't know what to do with the argument. This works for the example, because JSON's specification covers only a few basic structures.

The declaration and intent may seem simple at first, but look closer. This final candidate matches not only objects for which there is no candidate defined, but it can match for *all* objects, including `Int`, `Bool`, `Num`. A call like `to-json(2)` has *two* matching candidates--`Int` and `Any`.

If you run that code, you'll discover that the `Int` candidate gets called. Because `Int` is a type that conforms to `Any`, it is a *narrower* match for an integer. Given two types `A` and `B`, where `A` conforms to `B` (`A ~~ B`, in Perl 6 code), an object which conforms to `A` does so more narrowly than to `B`. In the case of multi dispatch, the narrowest match always wins.

A successfully evaluated constraint makes a match narrower than a similar signature without a constraint. In the case of:

```
multi to-json($d) { ... }
multi to-json($d where {!defined $d}) { ... }
```

... an undefined value dispatches to the second candidate.

However, a matching constraint always contributes less to narrowness than a more specific match in the nominal type.

```
TODO: Better example

multi a(Any $x where { $x > 0 }) { 'Constraint'   }
multi a(Int $x)                  { 'Nominal type' }

say a(3), ' wins';       # says B<Nominal type wins>
```

This restriction allows a clever compiler optimization: it can sort all candidates by narrowness once to find the candidate with the best matching signature by examining nominal type constraints. These are far cheaper to check than constraint checks. Constraint checking occurs next, then the compiler considers the nominal types of candidates.

With some trickery it is possible to get an object which conforms to a built-in type (`Num`, for example) but which is also an undefined value. In this case the candidate that is specific to `Num` wins, because the nominal type check is narrower than the `where {!defined $d}` constraint.

Multiple arguments

Candidate signatures may contain any number of positional and named arguments, both explicit and slurpy. However only positional parameters contribute to the narrowness of a match:

```
# RAKUDO has problems with an enum here,
# it answers with "Player One wins\nDraw\nDraw"
# using separate classes would fix that,
# but is not as pretty.
enum Symbol <Rock Paper Scissors>;
multi wins(Scissors $, Paper    $) { +1 }
multi wins(Paper    $, Rock     $) { +1 }
multi wins(Rock     $, Scissors $) { +1 }
multi wins(::T      $, T        $) {  0 }
```

```
multi wins(         $,          $) { -1 }

sub play($a, $b) {
    given wins($a, $b) {
        when +1 { say 'Player One wins' }
        when  0 { say 'Draw'            }
        when -1 { say 'Player Two wins' }
    }
}

play(Scissors, Paper);
play(Paper,    Paper);
play(Rock,     Paper);
```

This example demonstrates how multiple dispatch can encapsulate all of the rules of a popular game. Both players independently select a symbol (rock, paper, or scissors). Scissors win against paper, paper wraps rock, and scissors can't cut rock, but go blunt trying. If both players select the same item, it's a draw.

The code creates a type for each possible symbol by declaring an enumerated type, or *enum*. For each combination of chosen symbols for which Player One wins there's a candidate of the form:

```
multi wins(Scissors $, Paper $) { +1 }
```

Because the bodies of the subs here do not use the parameters, there's no reason to force the programmer to name them; they're *anonymous parameters*. A single `$` in a signature identifies an anonymous scalar variable.

The fourth candidate, `multi wins(::T $, T $) { 0 }` uses `::T`, which is a *type capture* (similar to *generics* or *templates* in other programming languages). It binds the nominal type of the first argument to `T`, which can then act as a type constraint. If you pass a `Rock` as the first argument, `T` acts as an alias for `Rock` inside the rest of the signature and the body of the routine. The signature `(::T $, T $)` will bind only two objects of the same type, or where the second is of a subtype of the first.

In this game, that fourth candidate matches only for two objects of the same type. The routine returns `0` to indicate a draw.

The final candidate is a fallback for the cases not covered yet--every case in which Player Two wins.

If the `(Scissors, Paper)` candidate matches the supplied argument list, it is two steps narrower than the `(Any, Any)` fallback, because both `Scissors` and `Paper` are direct subtypes of `Any`, so both contribute one step.

If the `(::T, T)` candidate matches, the type capture in the first parameter does not contribute any narrowness--it is not a constraint, after all. However `T` *is* a constraint for the second parameter

which accounts for as many steps of narrowness as the number of inheritance steps between `T` and `Any`. Passing two `Rock`s means that `::T, T` is one step narrower than `Any, Any`. A possible candidate:

```
multi wins(Rock $, Rock $) {
    say "Two rocks? What is this, 20,000 years ago?"
}
```

... would win against `(::T, T)`.

Bindability checks

Traits can apply *implicit constraints*:

```
multi swap($a is rw, $b is rw) {
    ($a, $b) = ($b, $a);
}
```

This routine exchanges the contents of its two arguments. It must bind the two arguments as `rw`--both readable and writable. Calling the `swap` routine with an immutable value (for example a number literal) will fail.

The built-in function `substr` can not only extract parts of strings, but also modify them:

```
# substr(String, Start, Length)
say substr('Perl 5', 0, 4);         # prints B<Perl>

my $p = 'Perl 5';
# substr(String, Start, Length, Substitution)
substr($p, 6, 1, '6');
# now $p contains the string B<Perl 6>
```

You already know that the three-argument version and the four-argument version have different candidates: the latter binds its first argument as `rw`:

```
multi substr($str, $start = 0, $length = *) { ... }
multi substr($str is rw, $start, $length, $substitution) { ... }
```

This is also an example of candidates with different *arity* (number of expected arguments). This is seldom really necessary, because it is often a better alternative to make parameters optional. Cases where an arbitrary number of arguments are allowed are handled with slurpy parameters instead:

```
sub mean(*@values) {
    ([+] @values) / @values;
}
```

Nested Signatures in Multi-dispatch

An earlier chapter showed how to use nested signatures to look deeper into data structures and extract parts of them. In the context of multiple dispatch, nested signatures take on a second task: they act as constraints to distinguish between the candidates. This means that it is possible to dispatch based upon the shape of a data structure. This brings Perl 6 a lot of the expressive power provided by pattern matching in various functional languages.

Some algorithms have very tidy and natural expressions with this feature, especially those which recurse to a simple base case. Consider quicksort. The base case is that of the empty list, which trivially sorts to the empty list. A Perl 6 version might be:

```
multi quicksort([]) { () }
```

The `[]` declares an empty nested signature for the first positional parameter. Additionally, it requires that the first positional parameter be an indexable item--anything that would match the `@` sigil. The signature will only match if the multi has a single parameter which is an empty list.

The other case is a list which contains at least one value--the pivot--and possibly other values to partition according to the pivot. The rest of quicksort is a couple of recursive calls to sort both partitions:

```
multi quicksort([$pivot, *@rest]) {
    my @before = @rest.grep({ $_ <= $pivot });
    my @after  = @rest.grep({ $_ >  $pivot });

    return quicksort(@before), $pivot, quicksort(@after);
}
```

Protos

You have two options to write multi subs: either you start every candidate with `multi sub ...` or `multi ...`, or you declare once and for all that the compiler shall view every sub of a given name as a multi candidate. Do the latter by installing a *proto* routine:

```
proto to-json($) { ... }       # literal ... here

# automatically a multi
sub to-json(Bool $d) { $d ?? 'true' !! 'false' }
```

Nearly all Perl 6 built-in functions and operators export a proto definition, which prevents accidental overriding of built-ins[1].

> To hide all candidates of a multi and replace them by another sub, declare it as `only sub YourSub`. At the time of writing, no compiler supports this.

Toying with the candidate list

Each multi dispatch builds a list of candidates, all of which satisfy the nominal type constraints. For a normal sub or method call, the dispatcher invokes the first candidate which passes any additional constraint checks.

A routine can choose to delegate its work to other candidates in that list. The `callsame` primitive calls the next candidate, passing along the arguments received. The `callwith` primitive calls the next candidate with different (and provided) arguments. After the called routine has done its work, the callee can continue its work.

If there's no further work to do, the routine can decide to hand control completely to the next candidate by calling `nextsame` or `nextwith`. The former reuses the argument list and the latter allows the use of a different argument list. This delegation is common in object destructors, where each subclass may perform some cleanup for its own particular data. After it finishes its work, it can delegate to its parent class meethod by calling `nextsame`.

[1] One of the very rare exceptions is the smart match operator `infix:<~~>` which is not easily overloadable. Instead it redispatches to overloadable multi methods.

Глава 7. Roles

A *role* is a standalone, named, and reusable unit of behavior. You can compose a role into a class at compile time or add it to an individual object at runtime.

That's an abstract definition best explained by an example. This program demonstrates a simple and pluggable IRC bot framework which understands a few simple commands.

```
# XXX This is VERY preliminary code and needs filling out. But it
# does provide opportunities to discuss runtime mixins, compile time
# composition, requirements and a few other bits.

my regex nick { \w+ }
my regex join-line { ... <nick> ... }
my regex message-line { $<sender>=[...] $<message>=[...] }

class IRCBot {
    has $.bot-nick;
    method run($server) {
        ...
    }
}

role KarmaTracking {
    has %!karma-scores;

    multi method on-message($sender, $msg where /^karma <ws> <nick>/) {
        if %!karma-scores{$<nick>} -> $karma {
            return $<nick> ~ " has karma $karma";
        }
        else {
            return $<nick> ~ " has neutral karma";
        }
    }

    multi method on-message($sender, $msg where /<nick> '++'/) {
        %!karma-scores{$<nick>}++;
    }

    multi method on-message($sender, $msg where /<nick> '--'/) {
        %!karma-scores{$<nick>}--;
    }
}

role Oping {
    has @!whoz-op;

    multi method on-join($nick) {
        if $nick eq any(@!whoz-op) {
            return "/mode +o $nick";
```

```
        }
    }

    # I'm tempted to add another candidate here which checks any(@!whoz-op)
    multi method on-message($sender, $msg where /^trust <ws> <nick>/) {
        if $sender eq any(@!whoz-op) {
            push @!whoz-op, $<nick>;
            return "I now trust " ~ $<nick>;
        }
        else {
            return "But $sender, I don't trust you";
        }
    }
}

role AnswerToAll {
    method process($raw-in) {
        if $raw-in ~~ /<on-join>/ {
            self.*on-join($<nick>);
        }
        elsif $raw-in ~~ /<on-message>/ {
            self.*on-message($<sender>, $<message>)
        }
    }
}

role AnswerIfTalkedTo {
    method bot-nick() { ... }

    method process($raw-in) {
        if $raw-in ~~ /<on-join>/ {
            self.*on-join($<nick>);
        }
        elsif $raw-in ~~ /<on-message>/ -> $msg {
            my $my-nick = self.bot-nick();
            if $msg<msg> ~~ /^ $my-nick ':'/ {
                self.*on-message($msg<sender>, $msg<message>)
            }
        }
    }
}

my %pluggables =
    karma => KarmaTracking,
    op    => Oping;

role Plugins {
    multi method on-message($self is rw: $sender, $msg where /^youdo <ws> (\w+
        if %pluggables{$0} -> $plug-in {
            $self does $plug-in;
            return "Loaded $0";
        }
    }
}
```

```
class AdminBot    is IRCBot does KarmaTracking    does Oping       {}
class KarmaKeeper is IRCBot does KarmaTracking    does AnswerToAll {}
class NothingBot  is IRCBot does AnswerIfTalkedTo does Plugins     {}
```

You don't have to understand everything in this example yet. It's only important right now to notice that the classes `KarmaKeeper` and `NothingBot` share some behavior by inheriting from `IRCBot` and differentiate their behaviors by performing different roles.

What is a role?

Previous chapters have explained classes and grammars. A role is another type of package. Like classes and grammars, a role can contain methods (including named regexes) and attributes. However, a role cannot stand on its own; you cannot instantiate a role. To use a role, you must incorporate it into an object, class, or grammar.

In other object systems, classes perform two tasks. They represent entities in the system, providing models from which to create instances. They also provide a mechanism for code re-use. These two tasks contradict each other to some degree. For optimal re-use, classes should be small, but in order to represent a complex entity with many behaviors, classes tend to grow large. Large projects written in such systems often have complex interactions and workarounds for classes which want to reuse code but do not want to take on additional unnecessary capabilities.

Perl 6 classes retain the responsibility for modeling and managing instances. Roles handle the task of code reuse. A role contains the methods and attributes required to provide a named, reusable unit of behavior. Building a class out of roles uses a safe mechanism called *flattening composition*. You may also apply a role to an individual object. Both of these design techniques appear in the example code.

Some roles--*parametric roles*--allow the use of specific customizations to change how they provide the features they provide. This helps Perl 6 provide generic programming, along the lines of generics in C# and Java, or templates in C++.

Compile Time Composition

Look at the `KarmaKeeper` class declaration. The body is empty; the class defines no attributes or methods of its own. The class inherits from `IRCBot`, using the `is` trait modifier--something familiar from earlier chapters--but it also uses the `does` trait modifier to compose two roles into the class.

The process of role composition is simple. Perl takes the attributes and methods defined in each role and copies them into the class. After composition, the class appears as if those attributes and methods had been declared in the class's declaration itself. This is part of the flattening property: after composing a role into the class, the roles in and of themselves are only important when querying the class to determine *if* it performs the role. Querying the methods of the `KarmaKeeper`

class through introspection will report that the class has both a `process` method and an `on-message` multi method.

If this were all that roles provided, they'd have few advantages over inheritance or mixins. Roles get much more interesting in the case of a conflict. Consider the class definition:

```
class MyBot is IRCBot does AnswerToAll does AnswerIfTalkedTo {}
```

Both the `AnswerToAll` and `AnswerIfTalkedTo` roles provide a method named `process`. Even though they share a name, the methods perform semantically different--and conflicting--behaviors. The role composer will produce a compile-time error about this conflict, asking the programmer to provide a resolution.

Multiple inheritance and mixin mechanisms rarely provide this degree of conflict resolution. In those situations, the order of inheritance or mixin decides which method wins. All possible roles are equal in role composition.

What can you do if there is a conflict? In this case, it makes little sense to compose both of the roles into a class. The programmer here has made a mistake and should choose to compose only one role to provide the desired behavior. An alternative way to resolve a conflict is to write a method with the same name in the class body itself:

```
class MyBot is IRCBot does AnswerToAll does AnswerIfTalkedTo {
    method process($raw-in) {
        # Do something sensible here...
    }
}
```

If the role composer detects a method with the same name in the class body, it will then disregard all of the (possibly conflicting) ones from the roles. Put simply, methods in the class always supersede methods which a role may provide.

> What happens when a class performs a role but overrides all of its methods? That's okay too: declaring that a class performs a role does not require you to compose in any behavior from the role. The role composer will verify that all of the role's requirements are satisfied once and only once, and from then on Perl's type system will consider all instances of the class as corresponding to the type implied by the role.

Multi-methods and composition

Sometimes it's okay to have multiple methods of the same name, provided they have different signatures such that the multidispatch mechanism can distinguish between them. Multi methods with the same name from different roles will not conflict with each other. Instead, the candidates from all of the roles will combine during role composition.

If the class provides a method of the same name that is also multi, then all methods defined in the role and the class will combine into a set of multi candidates. Otherwise, if the class has a method

of the same name that is *not* declared as a multi, then the method in the class alone--as usual--will take precedence. This is the mechanism by which the `AdminBot` class can perform the appropriate `on-message` method provided by both the `KarmaTracking` and the `Oping` roles.

When a class composes multiple roles, an alternate declaration syntax may be more readable:

```
class KarmaKeeper is IRCBot {
    does AnswerToAll;
    does KarmaTracking;
    does Oping;
}
```

Calling all candidates

The `process` methods of the roles `AnswerToAll` and `AnswerIfTalkedTo` use a modified syntax for calling methods:

```
self.*on-message($msg<sender>, $msg<message>)
```

The `.*` method calling syntax changes the semantics of the dispatch. Just as the `*` quantifier in regexes means "zero or more", the `.*` dispatch operator will call zero or more matching methods. If no `on-message` multi candidates match, the call will not produce an error. If more than one `on-message` multi candidate matches, Perl will call all of them, whether found by multiple dispatch, searching the inheritance hierarchy, or both.

There are two other variants. `.+` greedily calls all methods but dies unless it can call at least one method. `.?`, tries to call one method, but returns a `Failure` rather then throwing an exception. These dispatch forms may seem rare, but they're very useful for event driven programming. One-or-failure is very useful when dealing with per-object role application.

Expressing requirements

The role `AnswerIfTalkedTo` declares a stub for the method `bot-nick`, but never provides an implementation.

```
method bot-nick() { ... }
```

In the context of a role, this means that any class which composes this role must somehow provide a method named `bot-nick`. The class itself may provide it, another role must provide it, or a parent class must provide it. `IRCBot` does the latter; it `IRCBot` defines an attribute `$!bot-nick` along with an accessor method.

If you do not make explicit the methods on which your role depends, the role composer will not verify their existence at compilation time. Any missing methods will cause runtime errors (barring

the use of something like `AUTOMETH`). As compile-time verification is an important feature of roles, it's best to mark your dependencies.

Runtime Application of Roles

Class declarations frozen at compilation time are often sufficient, but sometimes it's useful to add new behaviors to individual objects. Perl 6 allows you to do so by applying roles to individual objects at runtime.

The example in this chapter uses this to give bots new abilities during their lifetimes. The `Plugins` role is at the heart of this. The signature of the method `on-message` captures the invocant into a variable `$self` marked `rw`, which indicates that the invocant may be modified. Inside the method, that happens:

```
if %pluggables{$0} -> $plug-in {
    B<$self does $plug-in;>
    return "Loaded $0";
}
```

Roles in Perl 6 are first-class entities, just like classes. You can pass roles around just like any other object. The `%pluggables` hash maps names of plug-ins to Role objects. The lookup inside `on-message` stores a Role in `$plug-in`. The `does` operator adds this role to `$self`--not the *class* of `$self`, but the instance itself. From this point on, `$self` now has all of the methods from the role, in addition to all of the ones that it had before. This does affect any other instances of the same class; only this one instance has changed.

Differences from compile time composition

Runtime application differs from compile time composition in that methods in the applied role in will automatically override any of the same name within the class of the object. It's as if you had written an anonymous subclass of the current class of the object that composed the role into it. This means that `.*` will find both those methods that mixed into the object from one or more roles along with any that already existed in the class.

If you wish to apply multiple roles at a time, list them all with `does`. This case behaves the same way as compile-time composition, in that the role composer will compose them all into the imaginary anonymous subclass. Any conflicts will occur at this point.

This gives a degree of safety, but it happens at runtime and is thus not as safe as compile time composition. For safety, perform your compositions at compile time. Instead of applying multiple roles to an instance, compose them into a new role at compile time and apply that role to the instance.

The **`but`** operator

Runtime role application with `does` modifies an object in place: `$x does SomeRole` modifies the object stored in `$x`. Sometimes this modification is not what you want. In that case, use the

`but` operator, which clones the object, performs the role composition with the clone, and returns the clone. The original object stays the same.

TODO: example

Parametric Roles

Roles and Types

Глава 8. Subtypes

```
enum Suit <spades hearts diamonds clubs>;
enum Rank (2, 3, 4, 5, 6, 7, 8, 9, 10,
           'jack', 'queen', 'king', 'ace');

class Card {
    has Suit $.suit;
    has Rank $.rank;

    method Str {
        $.rank.name ~ ' of ' ~ $.suit.name;
    }
}

subset PokerHand of List where { .elems == 5 && all(|$_) ~~ Card }

sub n-of-a-kind($n, @cards) {
    for @cards>>.rank.uniq -> $rank {
        return True if $n == grep $rank, @cards>>.rank;
    }
    return False;
}

subset Quad         of PokerHand where { n-of-a-kind(4, $_) }
subset ThreeOfAKind of PokerHand where { n-of-a-kind(3, $_) }
subset OnePair      of PokerHand where { n-of-a-kind(2, $_) }

subset FullHouse of PokerHand where OnePair & ThreeOfAKind;

subset Flush of PokerHand where -> @cards { [==] @cards>>.suit }

subset Straight of PokerHand where sub (@cards) {
    my @sorted-cards = @cards.sort({ .rank });
    my ($head, @tail) = @sorted-cards;
    for @tail -> $card {
        return False if $card.rank != $head.rank + 1;
        $head = $card;
    }
    return True;
}

subset StraightFlush of Flush where Straight;

subset TwoPair of PokerHand where sub (@cards) {
    my $pairs = 0;
    for @cards>>.rank.uniq -> $rank {
        ++$pairs if 2 == grep $rank, @cards>>.rank;
    }
    return $pairs == 2;
}
```

```
sub classify(PokerHand $_) {
    when StraightFlush { 'straight flush',  8 }
    when Quad          { 'four of a kind',  7 }
    when FullHouse     { 'full house',      6 }
    when Flush         { 'flush',           5 }
    when Straight      { 'straight',        4 }
    when ThreeOfAKind  { 'three of a kind', 3 }
    when TwoPair       { 'two pair',        2 }
    when OnePair       { 'one pair',        1 }
    when *             { 'high cards',      0 }
}

my @deck = map -> $suit, $rank { Card.new(:$suit, :$rank) },
                  (Suit.pick(*) X Rank.pick(*));

@deck .= pick(*);

my @hand1;
@hand1.push(@deck.shift()) for ^5;
my @hand2;
@hand2.push(@deck.shift()) for ^5;

say 'Hand 1: ', map { "\n  $_" }, @hand1>>.Str;
say 'Hand 2: ', map { "\n  $_" }, @hand2>>.Str;

my ($hand1-description, $hand1-value) = classify(@hand1);
my ($hand2-description, $hand2-value) = classify(@hand2);

say sprintf q[The first hand is a '%s' and the second one a '%s', so %s.],
    $hand1-description, $hand2-description,
    $hand1-value > $hand2-value
        ?? 'the first hand wins'
        !! $hand2-value > $hand1-value
            ?? 'the second hand wins'
            !! "the hands are of equal value"; # XXX: this is wrong
```

Глава 9. Pattern matching

Regular expressions are a computer science concept where simple patterns describe the format of text. Pattern matching is the process of applying these patterns to actual text to look for matches. Most modern regular expression facilities are more powerful than traditional regular expressions due to the influence of languages such as Perl, but the short-hand term `regex` has stuck and continues to mean "regular expression-like pattern matching". In Perl 6, though the specific syntax used to describe the patterns is different from PCRE[1] and POSIX[2], we continue to call them `regex`.

A common writing error is to duplicate a word by accident. It is hard to catch such errors by rereading your own text, but Perl can do it for you using `regex`:

```
my $s = 'the quick brown fox jumped over the the lazy dog';

if $s ~~ m/ « (\w+) \W+ $0 » / {
    say "Found '$0' twice in a row";
}
```

The simplest case of a regex is a constant string. Matching a string against that regex searches for that string:

```
if 'properly' ~~ m/ perl / {
    say "'properly' contains 'perl'";
}
```

The construct `m/ ... /` builds a regex. A regex on the right hand side of the `~~` smart match operator applies against the string on the left hand side. By default, whitespace inside the regex is irrelevant for the matching, so writing the regex as `m/ perl /`, `m/perl/` or `m/ p e rl/` all produce the exact same semantics--although the first way is probably the most readable.

Only word characters, digits, and the underscore cause an exact substring search. All other characters may have a special meaning. If you want to search for a comma, an asterisk, or another non-word character, you must quote or escape it[3]:

```
my $str = "I'm *very* happy";

# quoting
if $str ~~ m/ '*very*' /   { say '\o/' }
```

[1] Perl Compatible Regular Expressions
[2] Portable Operating System Interface for Unix. See IEEE standard 1003.1-2001
[3] To search for a literal string--without using the pattern matching features of regex--consider using `index` or `rindex` instead.

```
# escaping
if $str ~~ m/ \* very \* / { say '\o/' }
```

Searching for literal strings gets boring pretty quickly. Regex support special (also called *metasyntactic*) characters. The dot (`.`) matches a single, arbitrary character:

```
my @words = <spell superlative openly stuff>;

for @words -> $w {
    if $w ~~ m/ pe.l / {
        say "$w contains $/";
    } else {
        say "no match for $w";
    }
}
```

This prints:

```
spell contains pell
superlative contains perl
openly contains penl
no match for stuff
```

The dot matched an `l`, `r`, and `n`, but it will also match a space in the sentence *the spectrosco**pe l**acks resolution*--regexes ignore word boundaries by default. The special variable `$/` stores (among other things) only the part of the string that matched the regular expression. `$/` holds these so-called *match object*s.

Suppose you want to solve a crossword puzzle. You have a word list and want to find words containing `pe`, then an arbitrary letter, and then an `l` (but not a space, as your puzzle has extra markers for those). The appropriate regex for that is `m/pe \w l/`. The `\w` control sequence stands for a "Word" character--a letter, digit, or an underscore. This chapter's example uses `\w` to build the definition of a "word".

Several other common control sequences each match a single character:

Таблица 9.1. Backslash sequences and their meaning

Symbol	Description	Examples
`\w`	word character	l, ö, 3, _

Symbol	Description	Examples
`\d`	digit	0, 1
`\s`	whitespace	(tab), (blank), (newline)
`\t`	tabulator	(tab)
`\n`	newline	(newline)
`\h`	horizontal whitespace	(space), (tab)
`\v`	vertical whitespace	(newline), (vertical tab)

Invert the sense of each of these backslash sequences by uppercasing its letter: `\W` matches a character that's *not* a word character and `\N` matches a single character that's not a newline.

These matches extend beyond the ASCII range--`\d` matches Latin, Arabic-Indic, Devanagari and other digits, `\s` matches non-breaking whitespace, and so on. These *character classes* follow the Unicode definition of what is a letter, a number, and so on.

To define your own custom character classes, listing the appropriate characters inside nested angle and square brackets `<[ ... ]>`:

```
if $str ~~ / <[aeiou]> / {
    say "'$str' contains a vowel";
}

# negation with a -
if $str ~~ / <-[aeiou]> / {
    say "'$str' contains something that's not a vowel";
}
```

Rather than listing each character in the character class individually, you may specify a range of characters by placing the range operator `..` between the beginning and ending characters:

```
# match a, b, c, d, ..., y, z
if $str ~~ / <[a..z]> / {
    say "'$str' contains a lower case Latin letter";
}
```

You may add characters to or subtract characters from classes with the `+` and `-` operators:

```
if $str ~~ / <[a..z]+[0..9]> / {
    say "'$str' contains a letter or number";
```

```
}

if $str ~~ / <[a..z]-[aeiou]> / {
    say "'$str' contains a consonant";
}
```

The negated character class is a special application of this idea.

A *quantifier* specifies how often something has to occur. A question mark `?` makes the preceding unit (be it a letter, a character class, or something more complicated) optional, meaning it can either be present either zero or one times. `m/ho u? se/` matches either `house` or `hose`. You can also write the regex as `m/hou?se/` without any spaces, and the `?` will still quantify only the `u`.

The asterisk `*` stands for zero or more occurrences, so `m/z\w*o/` can match `zo`, `zoo`, `zero` and so on. The plus `+` stands for one or more occurrences, `\w+` *usually* matches what you might consider a word (though only matches the first three characters from `isn't` because `'` isn't a word character).

The most general quantifier is `**`. When followed by a number, it matches that many times. When followed by a range, it can match any number of times that the range allows:

```
# match a date of the form 2009-10-24:
m/ \d**4 '-' \d\d '-' \d\d /

# match at least three 'a's in a row:
m/ a ** 3..* /
```

If the right hand side is neither a number nor a range, it becomes a delimiter, which means that `m/ \w ** ', '/` matches a list of characters each separated by a comma and whitespace.

If a quantifier has several ways to match, Perl will choose the longest one. This is *greedy* matching. Appending a question mark to a quantifier makes it non-greedy[4]

For example, you can parse HTML very badly[5]with the code:

```
my $html = '<p>A paragraph</p> <p>And a second one</p>';

if $html ~~ m/ '<p>' .* '</p>' / {
    say 'Matches the complete string!';
```

[4]The non-greedy general quantifier is `$thing **? $count`, so the question mark goes directly after the second asterisk.
[5]Using a proper stateful parser is always more accurate.

```
}

if $html ~~ m/ '<p>' .*? '</p>' / {
    say 'Matches only <p>A paragraph</p>!';
}
```

To apply a modifier to more than just one character or character class, group items with square brackets:

```
my $ingredients = 'milk, flour, eggs and sugar';
# prints "milk, flour, eggs"
$ingredients ~~ m/ [\w+] ** [\,\s*] / && say $/;
```

Separate *alternations*--parts of a regex of which *any* can match-- with vertical bars. One vertical bar between multiple parts of a regex means that the alternatives are tried in parallel and the longest matching alternative wins. Two bars make the regex engine try each alternative in order and the first matching alternative wins.

```
$string ~~ m/ \d**4 '-' \d\d '-' \d\d | 'today' | 'yesterday' /
```

Anchors

So far every regex could match anywhere within a string. Often it is useful to limit the match to the start or end of a string or line or to word boundaries. A single caret ^ anchors the regex to the start of the string and a dollar sign $ to the end. `m/ ^a /` matches strings beginning with an `a`, and `m/ ^ a $ /` matches strings that consist only of an `a`.

Таблица 9.2. Regex anchors

Anchor	Meaning
^	start of string
$	end of string
^^	start of a line
$$	end of a line
<<	left word boundary
«	left word boundary
> >	right word boundary
»	right word boundary

Captures

Regex can be very useful for *extracting* information too. Surrounding part of a regex with round brackets (aka parentheses) `(...)` makes Perl *capture* the string it matches. The string matched by the first group of parentheses is available in `$/[0]`, the second in `$/[1]`, etc. `$/` acts as an array containing the captures from each parentheses group:

```
my $str = 'Germany was reunited on 1990-10-03, peacefully';

if $str ~~ m/ (\d**4) \- (\d\d) \- (\d\d) / {
    say 'Year:  ', $/[0];
    say 'Month: ', $/[1];
    say 'Day:   ', $/[2];
    # usage as an array:
    say $/.join('-');       # prints 1990-10-03
}
```

If you quantify a capture, the corresponding entry in the match object is a list of other match objects:

```
my $ingredients = 'eggs, milk, sugar and flour';

if $ingredients ~~ m/(\w+) ** [\,\s*] \s* 'and' \s* (\w+)/ {
    say 'list: ', $/[0].join(' | ');
    say 'end:  ', $/[1];
}
```

This prints:

```
list: eggs | milk | sugar
end:  flour
```

The first capture, `(\w+)`, was quantified, so `$/[0]` contains a list of words. The code calls `.join` to turn it into a string. Regardless of how many times the first capture matches (and how many elements are in `$/[0]`), the second capture is still available in `$/[1]`.

As a shortcut, `$/[0]` is also available under the name `$0`, `$/[1]` as `$1`, and so on. These aliases are also available inside the regex. This allows you to write a regex that detects that common error of duplicated words, just like the example at the beginning of this chapter:

```
my $s = 'the quick brown fox jumped over the the lazy dog';
```

```
if $s ~~ m/ « (\w+) \W+ $0 » / {
    say "Found '$0' twice in a row";
}
```

The regex first anchors to a left word boundary with « so that it doesn't match partial duplication of words. Next, the regex captures a word (`(\w+)`), followed by at least one non-word character `\W+`. This implies a right word boundary, so there is no need to use an explicit boundary. Then it matches the previous capture followed by a right word boundary.

Without the first word boundary anchor, the regex would for example match *str**and and** beach* or *la**the the** table leg*. Without the last word boundary anchor it would also match ***the the**ory*.

Named regexes

You can declare regexes just like subroutines--and even name them. Suppose you found the example at the beginning of this chapter useful and want to make it available easily. Suppose also you want to extend it to handle contractions such as `doesn't` or `isn't`:

```
my regex word { \w+ [ \' \w+]? }
my regex dup  { « <word=&word> \W+ $<word> » }

if $s ~~ m/ <dup=&dup> / {
    say "Found '{$<dup><word>}' twice in a row";
}
```

This code introduces a regex named `word`, which matches at least one word character, optionally followed by a single quote. Another regex called `dup` (short for *duplicate*) contains a word boundary anchor.

Within a regex, the syntax `<&word>` locates the regex `word` within the current lexical scope and matches against the regex. The `<name=&regex>` syntax creates a capture named `name`, which records what `&regex` matched in the match object.

In this example, `dup` calls the `word` regex, then matches at least one non-word character, and then matches the same string as previously matched by the regex `word`. It ends with another word boundary. The syntax for this *backreference* is a dollar sign followed by the name of the capture in angle brackets[6].

Within the `if` block, `$<dup>` is short for `$/{'dup'}`. It accesses the match object that the regex `dup` produced. `dup` also has a subrule called `word`. The match object produced from that call is accessible as `$<dup><word>`.

[6] In grammars--see ()--`<word>` looks up a regex named `word` in the current grammar and parent grammars, and creates a capture of the same name.

Named regexes make it easy to organize complex regexes by building them up from smaller pieces.

Modifiers

The previous example to match a list of words was:

```
m/(\w+) ** [\,\s*] \s* 'and' \s* (\w+)/
```

This works, but the repeated "I don't care about whitespace" units are clumsy. The desire to allow whitespace *anywhere* in a string is common. Perl 6 regexes allow this through the use of the `:sigspace` modifier (shortened to `:s`):

```
my $ingredients = 'eggs, milk, sugar and flour';

if $ingredients ~~ m/:s ( \w+ ) ** \,'and' (\w+)/ {
    say 'list: ', $/[0].join(' | ');
    say 'end:  ', $/[1];
}
```

This modifier allows optional whitespace in the text wherever there one or more whitespace characters appears in the pattern. It's even a bit cleverer than that: between two word characters whitespace is mandatory. The regex does *not* match the string `eggs, milk, sugarandflour`.

The `:ignorecase` or `:i` modifier makes the regex insensitive to upper and lower case, so `m/ :i perl /` matches `perl`, `PerL`, and `PERL` (though who names a programming language in all uppercase letters?)

Backtracking control

In the course of matching a regex against a string, the regex engine may reach a point where an alternation has matched a particular branch or a quantifier has greedily matched all it can, but the final portion of the regex fails to match. In this case, the regex engine backs up and attempts to match another alternative or matches one fewer character of the quantified portion to see if the overall regex succeeds. This process of failing and trying again is *backtracking*.

When matching `m/\w+ 'en'/` against the string `oxen`, the `\w+` group first matches the whole string because of the greediness of `+`, but then the `en` literal at the end can't match anything. `\w+` gives up one character to match `oxe`. `en` still can't match, so the `\w+` group again gives up one character and now matches `ox`. The `en` literal can now match the last two characters of the string, and the overall match succeeds.

While backtracking is often useful and convenient, it can also be slow and confusing. A colon `:` switches off backtracking for the previous quantifier or alternation. `m/ \w+: 'en'/` can never match any string, because the `\w+` always eats up all word characters and never releases them.

The `:ratchet` modifier disables backtracking for a whole regex, which is often desirable in a small regex called often from other regexes. The duplicate word search regex had to anchor the regex to word boundaries, because `\w+` would allow matching only part of a word. Disabling backtracking makes `\w+` always match a full word:

```
my regex word { :ratchet \w+ [ \' \w+]? }
my regex dup  { <word=&word> \W+ $<word> }

# no match, doesn't match the 'and'
# in 'strand' without backtracking
'strand and beach' ~~ m/<&dup>/
```

The effect of `:ratchet` applies only to the regex in which it appears. The outer regex will still backtrack, so it can retry the regex `word` at a different staring position.

The `regex { :ratchet ... }` pattern is common that it has its own shortcut: `token { ... }`. An idiomatic duplicate word searcher might be:

```
my B<token> word { \w+ [ \' \w+]? }
my regex dup   { <word> \W+ $<word> }
```

A token with the `:sigspace` modifier is a `rule`:

```
my rule wordlist { <word> ** \, 'and' <word> }
```

Substitutions

Regexes are also good for data manipulation. The `subst` method matches a regex against a string. With `subst` matches, it substitutes the matched portion of the string its the second operand:

```
my $spacey = 'with    many  superfluous   spaces';

say $spacey.subst(rx/ \s+ /, ' ', :g);
```

```
# output: with many superfluous spaces
```

By default, `subst` performs a single match and stops. The `:g` modifier tells the substitution to work *globally* to replace every possible match.

Note the use of `rx/ ... /` rather than `m/ ... /` to construct the regex. The former constructs a regex object. The latter constructs the regex object *and* immediately matches it against the topic variable `$_`. Using `m/ ... /` in the call to `subst` creates a match object and passes it as the first argument, rather than the regex itself.

Other Regex Features

Sometimes you want to call other regexes, but don't want them to capture the matched text. When parsing a programming language you might discard whitespace characters and comments. You can achieve that by calling the regex as `<.otherrule>`.

If you use the `:sigspace` modifier, every continuous piece of whitespace calls the built-in rule `<.ws>`. This use of a rule rather than a character class allows you to define your own version of whitespace characters (see).

Sometimes you just want to peek ahead to check if the next characters fulfill some properties without actually consuming them. This is common in substitutions. In normal English text, you always place a whitespace after a comma. If somebody forgets to add that whitespace, a regex can clean up after the lazy writer:

```
my $str = 'milk,flour,sugar and eggs';
say $str.subst(/',' <?before \w>/, ', ',  :g);
# output: milk, flour, sugar and eggs
```

The word character after the comma is not part of the match, because it is in a look-ahead introduced by `<?before ... >`. The leading question mark indicates an *zero-width assertion*: a rule that never consumes characters from the matched string. You can turn any call to a subrule into an zero width assertion. The built-in token `<alpha>` matches an alphabetic character, so you can rewrite this example as:

```
say $str.subst(/',' <?alpha>/, ', ',  :g);
```

An leading exclamation mark negates the meaning, such that the lookahead must *not* find the regex fragment. Another variant is:

```
say $str.subst(/',' <!space>/, ', ',  :g);
```

You can also look behind to assert that the string only matches *after* another regex fragment. This assertion is `<?after>`. You can write the equivalent of many built-in anchors with look-ahead and look-behind assertions, though they won't be as efficient.

Таблица 9.3. Emulation of anchors with look-around assertions

Anchor	Meaning	Equivalent Assertion
^	start of string	`<!after .>`
^^	start of line	`<?after ^ | \n >`
$	end of string	`<!before .>`
> >	right word boundary	`<?after \w>` `<!before \w>`

Match objects

```
sub line-and-column(Match $m) {
    my $line   = ($m.orig.substr(0, $m.from).split("\n")).elems;
    # RAKUDO workaround for RT #70003, $m.orig.rindex(...) directly fails
    my $column = $m.from - ('' ~ $m.orig).rindex("\n", $m.from);
    $line, $column;
}

my $s = "the quick\nbrown fox jumped\nover the the lazy dog";

my token word { \w+ [ \' \w+]? }
my regex dup { <word> \W+ $<word> }

if $s ~~ m/ <dup> / {
    my ($line, $column) = line-and-column($/);
    say "Found '{$<dup><word>}' twice in a row";
    say "at line $line, column $column";
}

# output:
# Found 'the' twice in a row
# at line 3, column 6
```

Every regex match returns an object of type `Match`. In boolean context, a match object returns `True` for successful matches and `False` for failed ones. Most properties are only interesting after successful matches.

The `orig` method returns the string that was matched against. The `from` and `to` methods return the positions of the start and end points of the match.

In the previous example, the `line-and-column` function determines the line number in which the match occurred by extracting the string up to the match position (`$m.orig.substr(0, $m.from)`), splitting it by newlines, and counting the elements. It calculates the column by searching backwards from the match position and calculating the difference to the match position.

> The `index` method searches a string for another substring and returns the position of the search string. The `rindex` method does the same, but searches backwards from the end of the string, so it finds the position of the final occurrence of the substring.

Using a match object as an array yields access to the positional captures. Using it as a hash reveals the named captures. In the previous example, `$<dup>` is a shortcut for `$/<dup>` or `$/{ 'dup' }`. These captures are again `Match` objects, so match objects are really trees of matches.

The `caps` method returns all captures, named and positional, in the order in which their matched text appears in the source string. The return value is a list of `Pair` objects, the keys of which are the names or numbers of the capture and the values the corresponding `Match` objects.

```
if 'abc' ~~ m/(.) <alpha> (.) / {
    for $/.caps {
        say .key, ' => ', .value;

    }
}

# Output:
# 0 => a
# alpha => b
# 1 => c
```

In this case the captures occur in the same order as they are in the regex, but quantifiers can change that. Even so, `$/.caps` follows the ordering of the string, not of the regex. Any parts of the string which match but not as part of captures will not appear in the values that `caps` returns.

To access the non-captured parts too, use `$/.chunks` instead. It returns both the captured and the non-captured part of the matched string, in the same format as `caps`, but with a tilde `~` as key. If there are no overlapping captures (as occurs from look-around assertions), the concatenation of all the pair values that `chunks` returns is the same as the matched part of the string.

Глава 10. Grammars

Grammars organize regexes, just like classes organize methods. The following example demonstrates how to parse JSON, a data exchange format already introduced (see).

```
# file lib/JSON/Tiny/Grammar.pm

grammar JSON::Tiny::Grammar {
    rule TOP        { ^[ <object> | <array> ]$ }
    rule object     { '{' ~ '}' <pairlist>     }
    rule pairlist   { [ <pair> ** [ \, ]  ]?   }
    rule pair       { <string> ':' <value>     }
    rule array      { '[' ~ ']' [ <value> ** [ \, ] ]?  }

    proto token value { <...> };

    token value:sym<number> {
        '-'?
        [ 0 | <[1..9]> <[0..9]>* ]
        [ \. <[0..9]>+ ]?
        [ <[eE]> [\+|\-]? <[0..9]>+ ]?
    }

    token value:sym<true>    { <sym>    };
    token value:sym<false>   { <sym>    };
    token value:sym<null>    { <sym>    };
    token value:sym<object>  { <object> };
    token value:sym<array>   { <array>  };
    token value:sym<string>  { <string> }

    token string {
        \" ~ \" [ <str> | \\ <str_escape> ]*
    }

    token str {
        [
            <!before \t>
            <!before \n>
            <!before \\>
            <!before \">
            .
        ]+
    #    <-["\\\t\n]>+
    }

    token str_escape {
        <["\\/bfnrt]> | u <xdigit>**4
    }

}
```

```
# test it:
my $tester = '{
    "country":  "Austria",
    "cities": [ "Wien", "Salzburg", "Innsbruck" ],
    "population": 8353243
}';

if JSON::Tiny::Grammar.parse($tester) {
    say "It's valid JSON";
} else {
    # TODO: error reporting
    say "Not quite...";
}
```

A grammar contains various named regex. Regex names may be constructed the same as subroutine names or method names. While regex names are completely up to the grammar writer, a rule named `TOP` will, by default, be invoked when the `.parse()` method is executed on a grammar. The above call to `JSON::Tiny::Grammar.parse($tester)` starts by attempting to match the regex named `TOP` to the string `$tester`.

In this example, the `TOP` rule anchors the match to the start and end of the string, so that the whole string has to be in valid JSON format for the match to succeed. After matching the anchor at the start of the string, the regex attempts to match either an `<array>` or an `<object>`. Enclosing a regex name in angle brackets causes the regex engine to attempt to match a regex by that name within the same grammar. Subsequent matches are straightforward and reflect the structure in which JSON components can appear.

Regexes can be recursive. An `array` contains `value`. In turn a `value` can be an `array`. This will not cause an infinite loop as long as at least one regex per recursive call consumes at least one character. If a set of regexes were to call each other recursively without progressing in the string, the recursion could go on infinitely and never proceed to other parts of the grammar.

The example grammar given above introduces the *goal matching syntax* which can be presented abstractly as: `A ~ B C`. In `JSON::Tiny::Grammar`, `A` is `'{'`, `B` is `'}'` and `C` is `<pairlist>`. The atom on the left of the tilde (`A`) is matched normally, but the atom to the right of the tilde (`B`) is set as the goal, and then the final atom (`C`) is matched. Once the final atom matches, the regex engine attempts to match the goal (`B`). This has the effect of switching the match order of the final two atoms (`B` and `C`), but since Perl knows that the regex engine should be looking for the goal, a better error message can be given when the goal does not match. This is very helpful for bracketing constructs as it puts the brackets near one another.

Another novelty is the declaration of a *proto token*:

```
proto token value { <...> };

token value:sym<number> {
    '-'?
    [ 0 | <[1..9]> <[0..9]>* ]
```

```
    [ \. <[0..9]>+ ]?
    [ <[eE]> [\+|\-]? <[0..9]>+ ]?
}

token value:sym<true>    { <sym>    };
token value:sym<false>   { <sym>    };
```

The `proto token` syntax indicates that `value` will be a set of alternatives instead of a single regex. Each alternative has a name of the form `token value:sym<thing>`, which can be read as *alternative of* `value` *with parameter* `sym` *set to* `thing`. The body of such an alternative is a normal regex, where the call `<sym>` matches the value of the parameter, in this example `thing`.

When calling the rule `<value>`, the grammar engine attempts to match the alternatives in parallel and the longest match wins. This is exactly like normal alternation, but as we'll see in the next section, has the advantage of being extensible.

Grammar Inheritance

The similarity of grammars to classes goes deeper than storing regexes in a namespace as a class might store methods. You can inherit from and extend grammars, mix roles into them, and take advantage of polymorphism. In fact, a grammar is a class which by default inherits from `Grammar` instead of `Any`.

Suppose you want to enhance the JSON grammar to allow single-line C++ or JavaScript comments, which begin with `//` and continue until the end of the line. The simplest enhancement is to allow such a comment in any place where whitespace is valid.

However, `JSON::Tiny::Grammar` only implicitly matches whitespace through the use of *rules*, which are like tokens but with the `:sigspace` modifier enabled. Implicit whitespace is matched with the inherited regex `<ws>`, so the simplest approach to enable single- line comments is to override that named regex:

```
grammar JSON::Tiny::Grammar::WithComments
    is JSON::Tiny::Grammar {

    token ws {
        \s* [ '//' \N* \n ]?
    }
}

my $tester = '{
    "country":  "Austria",
    "cities": [ "Wien", "Salzburg", "Innsbruck" ],
    "population": 8353243 // data from 2009-01
}';

if JSON::Tiny::Grammar::WithComments.parse($tester) {
    say "It's valid (modified) JSON";
}
```

The first two lines introduce a grammar that inherits from `JSON::Tiny::Grammar`. As subclasses inherit methods from superclasses, so any grammar rule not present in the derived grammar will come from its base grammar.

In this minimal JSON grammar, whitespace is never mandatory, so `ws` can match nothing at all. After optional spaces, two slashes `'//'` introduce a comment, after which must follow an arbitrary number of non- newline characters, and then a newline. In prose, the comment starts with `'//'` and extends to the rest of the line.

Inherited grammars may also add variants to proto tokens:

```
grammar JSON::ExtendedNumeric is JSON::Tiny::Grammar  {
    token value:sym<nan> { <sym> }
    token value:sym<inf> { <[+-]>? <sym> }
}
```

In this grammar, a call to `<value>` matches either one of the newly added alternatives, or any of the old alternatives from the parent grammar `JSON::Tiny::Grammar`. Such extensibility is difficult to achieve with ordinary, `|` delimited alternatives.

Extracting data

The `parse` method of a grammar returns a `Match` object through which you can access all the relevant information of the match. Named regex that match within the grammar may be accessed via the `Match` object similar to a hash where the keys are the regex names and the values are the `Match` object that represents that part of the overall regex match. Similarly, portions of the match that are captured with parentheses are available as positional elements of the `Match` object (as if it were an array).

Once you have the Match object, what can you *do* with it? You could recursively traverse this object and create data structures based on what you find or execute code. An alternative solution exists: *action methods*.

```
class JSON::Tiny::Actions {
    method TOP($/)      { make $/.values.[0].ast }
    method object($/)   { make $<pairlist>.ast.hash }
    method pairlist($/) { make $<pair>».ast }
    method pair($/)     { make $<string>.ast => $<value>.ast }
    method array($/)    { make [$<value>».ast] }
    method string($/)   { make join '', $/.caps>>.value>>.ast }

    # TODO: make that
    # make +$/
    # once prefix:<+> is sufficiently polymorphic
    method value:sym<number>($/) { make eval $/ }
    method value:sym<string>($/) { make $<string>.ast }
    method value:sym<true>  ($/) { make Bool::True  }
    method value:sym<false> ($/) { make Bool::False }
    method value:sym<null>  ($/) { make Any }
```

```
    method value:sym<object>($/) { make $<object>.ast }
    method value:sym<array> ($/) { make $<array>.ast }

    method str($/)               { make ~$/ }

    method str_escape($/) {
        if $<xdigit> {
            make chr(:16($<xdigit>.join));
        } else {
            my %h = '\\' => "\\",
                    'n'  => "\n",
                    't'  => "\t",
                    'f'  => "\f",
                    'r'  => "\r";
            make %h{$/};
        }
    }
}

my $actions = JSON::Tiny::Actions.new();
JSON::Tiny::Grammar.parse($str, :$actions);
```

This example passes an actions object to the grammar's `parse` method. Whenever the grammar engine finishes parsing a regex, it calls a method on the actions object with the same name as the current regex. If no such method exists, the grammar engine moves along. If a method does exist, the grammar engine passes the current match object as a positional argument.

Each match object has a slot called `ast` (short for *abstract syntax tree*) for a payload object. This slot can hold a custom data structure that you create from the action methods. Calling `make $thing` in an action method sets the `ast` attribute of the current match object to `$thing`.

> An abstract syntax tree, or AST, is a data structure which represents the parsed version of the text. Your grammar describes the structure of the AST: its root element is the `TOP` node, which contain children of the allowed types and so on.

In the case of the JSON parser, the payload is the data structure that the JSON string represents. For each matching rule, the grammar engine calls an action method to populate the `ast` slot of the match object. This process transforms the match tree into a different tree--in this case, the actual JSON tree.

Although the rules and action methods live in different namespaces (and in a real-world project probably even in separate files), here they are adjacent to demonstrate their correspondence:

```
rule TOP        { ^ [ <object> | <array> ]$ }
method TOP($/)  { make $/.values.[0].ast }
```

The `TOP` rule has an alternation with two branches, `object` and `array`. Both have a named capture. `$/.values` returns a list of all captures, here either the `object` or the `array` capture.

The action method takes the AST attached to the match object of that sub capture, and promotes it as its own AST by calling `make`.

```
rule object        { '{' ~ '}' <pairlist>  }
method object($/)  { make $<pairlist>.ast.hash }
```

The reduction method for `object` extracts the AST of the `pairlist` submatch and turns it into a hash by calling its `hash` method.

```
rule pairlist       { [ <pair> ** [ \, ] ]? }
method pairlist($/) { make $<pair>».ast; }
```

The `pairlist` rule matches multiple comma-separated pairs. The reduction method calls the `.ast` method on each matched pair and installs the result list in its own AST.

```
rule pair       { <string> ':' <value> }
method pair($/) { make $<string>.ast => $<value>.ast }
```

A pair consists of a string key and a value, so the action method constructs a Perl 6 pair with the `=>` operator.

The other action methods work the same way. They transform the information they extract from the match object into native Perl 6 data structures, and call `make` to set those native structures as their own ASTs.

The action methods that belong to a proto token are parametric in the same way as the alternative:

```
token value:sym<null>        { <sym>    };
method value:sym<null>($/)   { make Any }

token value:sym<object>      { <object> };
method value:sym<object>($/) { make $<object>.ast }
```

When a `<value>` call matches, the action method with the same parametrization as the matching alternative executes.

Глава 11. Built-in types, operators and methods

Many operators work on a particular *type* of data. If the type of the operands differs from the type of the operand, Perl will make copies of the operands and convert them to the needed types. For example, `$a + $b` will convert a copy of both `$a` and `$b` to numbers (unless they are numbers already). This implicit conversion is called *coercion*.

Besides operators, other syntactic elements coerce their elements: `if` and `while` coerce to truth values (`Bool`), `for` views things as lists, and so on.

Numbers

Sometimes coercion is transparent. Perl 6 has several numeric types which can intermix freely--such as subtracting a floating point value from an integer, as `123 - 12.1e1`.

The most important types are:

Int

> `Int` objects store integer numbers of arbitrary size. If you write a literal that consists only of digits, such as `12`, it is an `Int`.

Num

> `Num` is the floating point type. It stores sign, mantissa, and exponent, each with a fixed width. Calculations involving `Num` numbers are usually quite fast, though subject to limited precision.
>
> Numbers in scientific notation such as `6.022e23` are of type `Num`.

Rat

> `Rat`, short for *rational*, stores fractional numbers without loss of precision. It does so by tracking its numerator and denominator as integers, so mathematical operations on `Rat`s with large components can become quite slow. For this reason, rationals with large denominators automatically degrade to `Num`.
>
> Writing a fractional value with a dot as the decimal separator, such as `3.14`, produces a `Rat`.

Complex

> `Complex` numbers have two parts: a real part and an imaginary part. If either part is `NaN`, then the entire number may possibly be `NaN`.
>
> Numbers in the form `a + bi`, where `bi` is the imaginary component, are of type `Complex`.

The following operators are available for all number types:

Таблица 11.1. Binary numeric operators

Operator	Description
`**`	Exponentiation: `$a**$b` is `$a` to the power of `$b`
`*`	multiplication
`/`	division
`div`	integer division
`+`	addition
`-`	subtraction

Таблица 11.2. Unary numeric operators

Operator	Description
`+`	conversion to number
`-`	negation

Most mathematical functions are available both as methods and functions, so you can write both `(-5).abs` and `abs(-5)`.

Таблица 11.3. Mathematical functions and methods

Method	Description
`abs`	absolute value
`sqrt`	square root
`log`	natural logarithm
`log10`	logarithm to base 10
`ceil`	rounding up to an integer
`floor`	rounding down to an integer
`round`	rounding to next integer
`sign`	-1 for negative, 0 for zero, 1 for positive values

The trigonometric functions `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sec`, `cosec`, `cotan`, `asec`, `acosec`, `acotan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `sech`, `cosech`, `cotanh`, `asech`, `acosech` and `acotanh` work in units of radians by default. You may specify the unit with an argument of `Degrees`, `Gradians` or `Circles`. For example, `180.sin(Degrees)` is approximately `0`.

Strings

Strings stored as `Str` are sequences of characters, independent of character encoding. The `Buf` type is available for storing binary data. The `encode` method converts a `Str` to `Buf`. `decode` goes the other direction.

The following operations are available for strings:

Таблица 11.4. Binary string operators

Operator	Description
~	concatenation: `'a' ~ 'b'` is `'ab'`
x	replication: `'a' x 2` is `'aa'`

Таблица 11.5. Unary string operators

Operator	Description
~	conversion to string: ~1 becomes `'1'`

Таблица 11.6. String methods/functions

Method/function	Description
`.chomp`	remove trailing newline
`.substr($start, $length)`	extract a part of the string. `$length` defaults to the rest of the string
`.chars`	number of characters in the string
`.uc`	upper case
`.lc`	lower case
`.ucfirst`	convert first character to upper case
`.lcfirst`	convert first character to lower case
`.capitalize`	convert the first character of each word to upper case, and all other characters to lower case

Bool

A Boolean value is either `True` or `False`. Any value can coerce to a boolean in boolean context. The rules for deciding if a value is true or false depend on the type of the value:

- Empty strings and `"0"` evaluate to `False`. All other strings evaluate to `True`.
- All numbers except zero evaluate to `True`.
- Container types such as lists and hashes evaluate to `False` if they are empty, and to `True` if they contain at least one value.

Constructs such as `if` automatically evaluate their conditions in boolean context. You can force an explicit boolean context by putting a `?` in front of an expression. The `!` prefix negates the boolean value.

```
my $num = 5;
```

```
# implicit boolean context
if $num { say "True" }

# explicit boolean context
my $bool    = ?$num;

# negated boolean context
my $not_num = !$num;
```

Глава 12. Формат Pod

Широко известный формат ведения документации в perl5 - POD (Plain Old Documentation) совсем недавно отпраздновал 15 лет. Вместе с новой версией perl6 готовится новый формат : Pod. Чем отличается perl 5 POD от Perl 6 Pod ?

Немного исторических дат, связанных с обоими форматами:

18 октября 1994 — В списке анонса perl 5.000 присутвует поддержка POD

```
18 October 1994:
It was a complete rewrite of Perl.
A few of the features and pitfalls are:
 ...
  * The documentation is much more extensive
    and perldoc along with pod is introduced.
  ..
```

9 апреля 2005 — S26: спецификация формата Pod для perl6. Автор - Damian Conway [http://en.wikipedia.org/wiki/Damian_Conway].

25 Apr 2007 — Первая редакция формата.

August 16, 2009 — S26 - The Next Generation (preview [http://www.nntp.perl.org/group/perl.perl6.language/2009/08/msg32352.html]).

31 Jul 2010 — Последняя редакция. Появились декларативные блоки.

В то время как существующий ныне POD означает Perl Old Documentation, спецификация s26 представляет новый формат следующим образом:

> ***Pod*** *- является эволюцией POD. В сравнении с POD, Perl 6 Pod более однороден, компактен и выразительнее. Pod также характеризуется описательной нотацией разметки, чем презентационной.*

Таким образом Pod избавился от слова "старый".

Структура Pod

Одна из особенностей формата Pod - его структура. На первый взгляд, есть некоторое внешнее сходство между документацией, написанной в формате Perl5 POD, и текстом в формате Perl6 Pod. Это потому, что различия лежат в основе синтаксической структуры обоих форматов.

Синтаксическая структура Pod

Основным элементом формата Pod (*как и в perl5 POD*) являются директивы, используемые для определения границ блоков Pod, описания конфигурационной информации (*=config*) и т.д. Каждая директива начинается с символа "равно" (=) в начале строки.

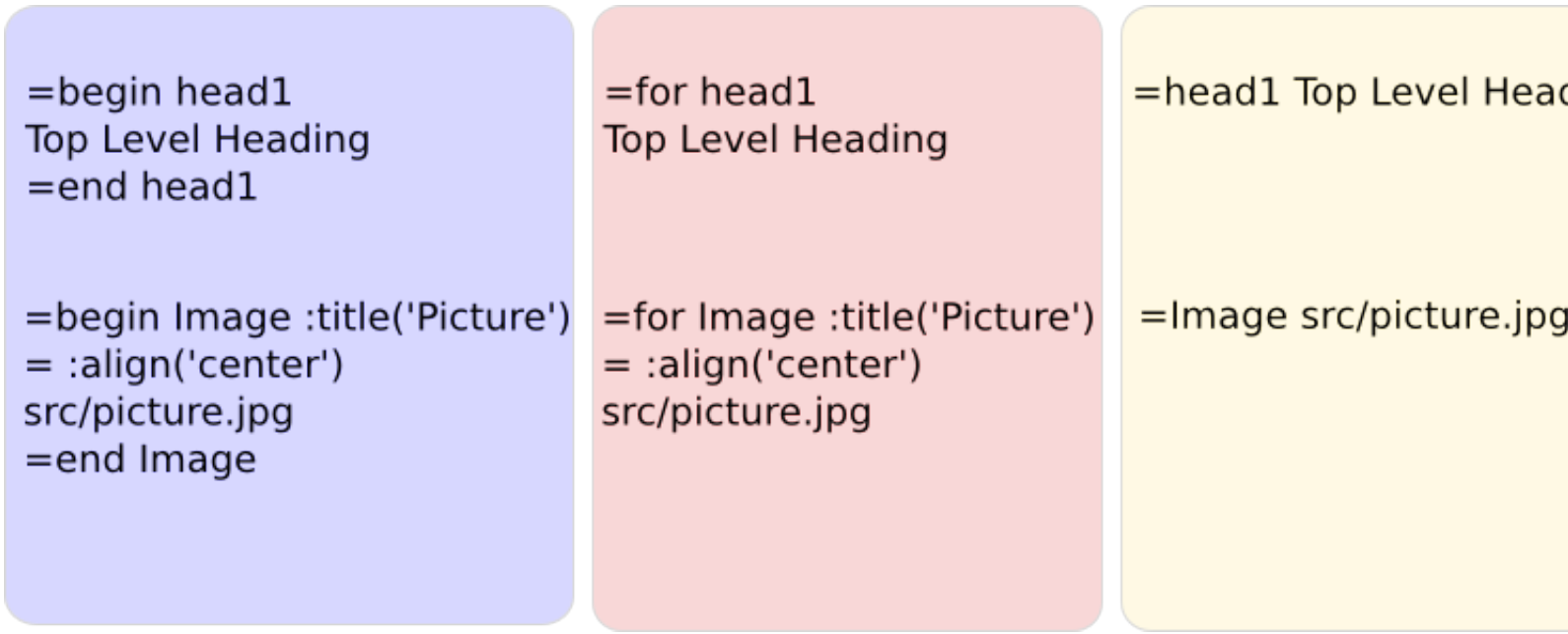
Примеры директив:

```
=config head2  :like<head1> :formatted<I>
=begin pod
=end pod
```

Содержимое документа состоит из одного или нескольких блоков. Каждый блок Pod может быть определен в виде трех равнозначных формах:

- Разграниченные блоки /Delimitedblocks
- Блоки-параграфы/Paragraph blocks
- Сокращенные блоки /Abbreviated blocks

Все три формы соответственно представлены на рисунке.



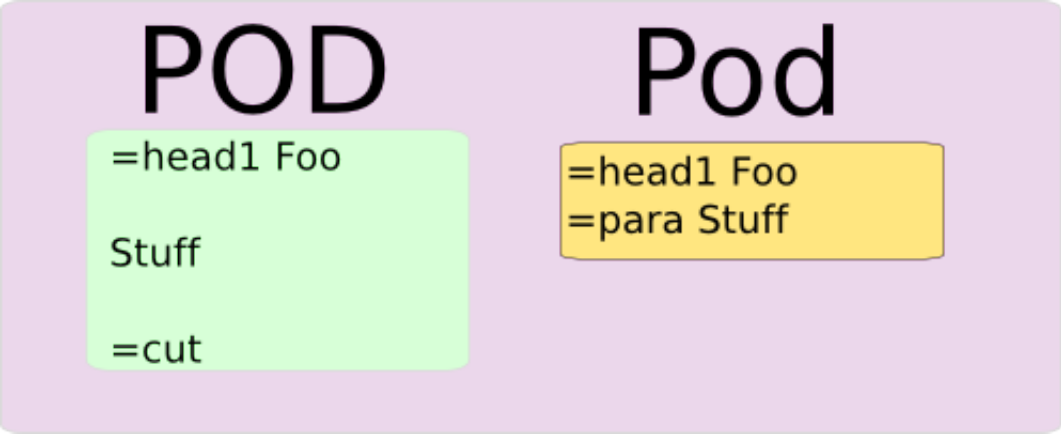
Каждая из форм имеет свои границы. Все содержимое документа, находящееся вне блоков Pod, определяется спецификацией как "молчаливый" материал. Этим материалом зачастую является исходный код программ, для документирования которого предназначен Pod.

В perl5 POD блок документации начинается с первой встреченной директивы и закачивается директивой =cut. Например:

```
=head1 test head

Some text

=cut
```

Таким образом получается, что структура Perl5 POD состоит из одного типа блока с указанными правилами определения границ, где директива =cut является неотъемлемой частью и служит признаком завершения блока. После этой директивы, обработчик (parser) Perl5 POD переключается в "молчаливый" режим пока не встретит следующий символ = в начале строки.

Именно изменения в определении границ блоков документации и являются тем фундаментальным различием обоих форматов.



Новые 3 формы определения блока Pod стали эволюционным развитием Perlpod Pod от POD (Plain Old Documentation).

Блоки Pod

Основной структурной единицей нового диалекта Pod [http://perlcabal.org/syn/S26.html] является блок документации. Он может быть представлен в виде 3 равнозначных форм:

- Разграниченные блоки / Delimited blocks
- Блоки-параграфы / Paragraph blocks
- Сокращенные блоки / Abbreviated blocks
- Блоки-декларатоpы / Declarator blocks

Разграниченные блоки / Delimited blocks

Разграниченные блоки имеют явно определенные границы. Для этого используются директивы *=begin* и *=end*, за каждой из которых следует имя типа блока (*typename*). Имена состоят из букв, цифр и символов подчеркивания, а начинаются с буквы или знака подчеркивания. Имена, состоящие целиком из символов нижнего (*=begin head1*) и верхнего *=begin SYNOPSIS* регистра, зарезервированы.

В строке с директивой *=begin* после имени блока следует *конфигурация данного блока*. Среди особенностей нового диалекта Pod - эта одна из самых замечательных. Конфигурация блока может использоваться в различных целях, в том числе и при создании *расширений для Pod*.

Конфигурационные параметры блока представлены в виде парной нотации в стиле Perl6 (SYNOPSIS 02 [http://perlcabal.org/syn/S02.html]).

Таблица 12.1. Парная нотация конфигурации блоков Pod

значение	формат определения	также...	также ..(*)
Boolean(true)	:key	:key(1)	key=>1

значение	формат определения	также...	также ..(*)
Boolean(false)	:!key	:key(0)	key=>0
String	:key<str>	:key('str')	key=>'str'
List	:key<1 2 3>	:key[1,2,3]	key=>[1,2,3]
Hash	:key{a=>1, b=>2}	-	key=>{a=>1, b=>2}

(*) - последняя форма не поддерживается в реализации Perl6::Pod [http://search.cpan.org/dist/Perl6-Pod/].

Если параметры блока не помещаются в одну строку, конфигурационный блок можно продолжить со следующей. В этом случае в начале строки ставиться символ = и пробел, далее конфигурационные параметры продолжаются.

Между директивами *=begin* и *=end* располагается *содержимое блока*. Сроки внутри блока могут содержать отступы, но они интерпретируются как блоки кода только в блоках *=pod*, *=item* *=code* и семантических блоках (например: *=METHOD*). То есть содержимое блока *=para* может отстоять от начала строки и не интерпретироваться при этом как код (verbitim paragraph в Perl5 POD).

Синтаксис блока выглядит следующим образом:

```
=begin BLOCK_TYPE  OPTIONAL CONFIG INFO
=                  OPTIONAL EXTRA CONFIG INFO
BLOCK CONTENTS
=end BLOCK_TYPE
```

Например:

```
=begin table  :caption<Table of Contents>
    Constants           1
    Variables           10
    Subroutines         33
    Everything else     57
=end table

=begin Name  :required
=            :width(50)
The applicant's full name
=end Name

=begin Contact  :optional
The applicant's contact details
=end Contact
```

Пустые строки между директивами, как это было в Perl5 POD не нужны; если они есть - то интерпретируются как часть содержимого блока. Кстати "пустыми" в Pod считаются также строки, содержащие только пробелы!

Блоки-параграфы / Paragraph blocks

Блоки параграфы начинаются с директивы *=for* и завершаются следующий директивой или пустой строкой (она не считается частью блока). После директивы *=for* следует имя блока и необязательные конфигурационные параметры.

Синтаксис этого типа блоков следующий:

```
=for BLOCK_TYPE  OPTIONAL CONFIG INFO
=                OPTIONAL EXTRA CONFIG INFO
BLOCK DATA
```

Примеры:

```
=for table  :caption<Table of Contents>
    Constants           1
    Variables           10
    Subroutines         33
    Everything else     57

=for Name  :required
=          :width(50)
The applicant's full name

=for Contact  :optional
The applicant's contact details
```

Сокращенные блоки / Abbreviated blocks

Сокращенные блоки начинаются с символа = за которым неразрывно следует имя блока. Продолжение строки интерпретируется как содержимое блока. Конфигурационных параметров в этой форме блока нет. Блок заканчивается перед следующей директивой Pod или пустой строкой (которая не ситается частью данных блока).

Синтаксис блока следующий:

```
=BLOCK_TYPE  BLOCK DATA
MORE BLOCK DATA
```

Пример:

```
=table
    Constants           1
    Variables           10
    Subroutines         33
    Everything else     57

=Name     The applicant's full name
=Contact  The applicant's contact details
```

Этот тип блока подходит для случаев, когда можно обойтись без конфигурирования блока. Иначе придется воспользоваться *=for* или *=begin/=end* директивами.

Блоки-деклараторы / Declarator blocks

Блоки-деклараторы особый тип блоков, который встроен в комментарии:

```
my $declared_thing;   #= Pod here until end of line
sub declared_thing () {   #=[ Pod here
                              until matching
```

```
                                  closing bracket
                                ]
        ...
}
```

Равнозначность стилевых блоков

Описанные выше типы блоков одинаково представлены во внутренней структуре документа. То есть если имя типа блока - параграф (*=para*), то он остается параграфом независимо от формы его описания.

Практически это означает, что приведенные ниже блоки:

```
=begin para
Text
=end para

=for para text

=para text
```

при конвертации в html будут преобразованы в один и тот же текст:

```
    <p>text</p>
```

Если тип блока таблица, то она останется ею в любом случае.

```
=begin table  :caption<Table of Contents>
   Constants         1
   Variables        10
   Subroutines       33
   Everything else    57

=end table
```

```
=for table  :caption<Table of Contents>
   Constants         1
   Variables        10
   Subroutines       33
   Everything else    57
```

```
=table
   Constants         1
   Variables        10
   Subroutines       33
   Everything else    57
```

Стандартные конфигурационные параметры

Pod резервирует несколько стандартных параметров для использования во встроенных типах блоков. Список этих параметров следующий:

:numbered Данный параметр указывает, что блок является нумерованным. Это свойство используется в заголовках (*=head1*, *=head2*) и списках (*=item*), но может быть указано для любого блока.

В случае произвольных блоков, стандарт передает интерпретацию данного свойства на усмотрение программе обрабатывающей этот блок.

Например:

```
Ягоды:
=for item :numbered
Клубника
=for item :numbered
Земляника
=for item :numbered
Черника
```

Будет выглядеть как :

Ягоды:

1. Клубника

2. Земляника

3. Черника

Примененное к заголовкам это свойство добавляет номер уровня.

:term Это свойство указывает на то , что данный список - список определений. Поэтому это свойство устонавливается для блоков *=item*.

:formatted Данный параметр дает указание интерпретировать содержимое блока, так словно оно обрамлено кодами форматирования.

Например вместо:

```
=begin para
B<I<
Warning: Do not immerse in water. Do not expose to bright light.
Do not feed after midnight.
>>
=end para
```

можно использовать:

```
=begin para :formatted<B I>
Warning: Do not immerse in water. Do not expose to bright light.
Do not feed after midnight.
=end para
```

Данные формы во внутреннем представлении почти эквивалентны. Единственное различие: во втором случае свойство *:formatted* остается в атрибутах объекта блока.

Коды форматирования, указанные в свойстве *:formatted*, дополняют уже примененные к блоку.

:like Замечательное свойство *:like* помогает навести порядок в параметрах блоков. Она указывает имена блоков, чьи свойства применить к текущему, тем самым снижая дублирование. Вполне подобное поведение можно назвать наследованием.

Параметр *:like* может быть применен к любому блоку, а также к директиве *=config*.

Пример:

```
=config head1 :numbered
=config head2  :like<head1> :formatted<I>
```

В этом примере, благодаря *:like*, блоки заголовков второго уровня *=head2* становятся нумерованными.

:allow Данное свойство разрешает использование внутри блока только указанные коды форматирования (*оригинальная спецификация ограничивала применение этого кода блоками =code*).

Вложенность блоков

Уровень вложенности в Pod - одна из составляющих его объектной модели документа. Вложенность блоков зачастую отмечается дополнительными отступами, но возможны и другие способы отображения: рамками, элементами сворачивания.

Любой блок может быть вложенным *(nested)*. Для этого достаточно указать атрибут блока `:nested`:

```
=begin para :nested
    We are all of us in the gutter,
    but some of us are looking at the stars!
=end para
```

Однако, указание атрибута вложенности для каждого блока быстро становится утомительным занятием, если таких блоков несколько или требуется несколько уровней вложенности:

```
=begin para :nested
    We are all of us in the gutter,
    but some of us are looking at the stars!
=end para
=begin para :nested(2)
        -- Oscar Wilde
=end para
```

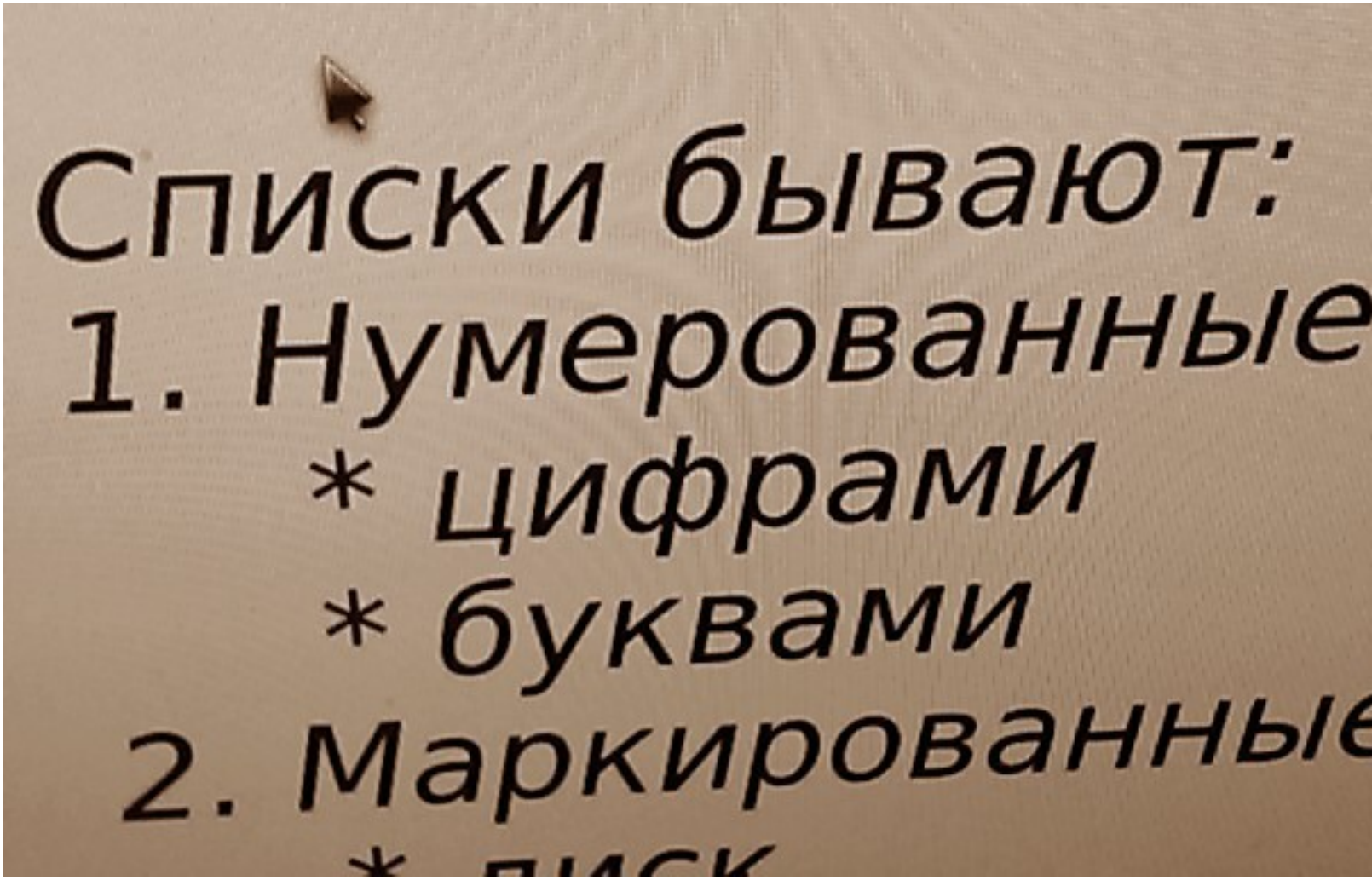
Формат Pod предоставляет блок `=nested`, который означает, что все его содержимое должно быть вложенным:

```
=begin nested
We are all of us in the gutter,
but some of us are looking at the stars!
    =begin nested
    -- Oscar Wilde
    =end nested
=end nested
```

Блоки вложенности `=nested` могут содержать любое количество блоков, включая неявные параграфы и блоки кода. Следует отметить, что физические отступы блоков не играют роли при определении их уровня вложенности. Предидущий пример может быть переписан с учетом этого следующим образом:

```
=begin nested
We are all of us in the gutter,
but some of us are looking at the stars!
=begin nested
-- Oscar Wilde
=end nested
=end nested
```

Списки



Списки в Pod представлены в виде групп, следующих друг за другом блоков `=item`. Каких либо специальных директив-"контейнеров" или разделителей для определения границ списков нет. Например:

```
The seven suspects are:

=item  Happy
=item  Dopey
=item  Sleepy
=item  Bashful
=item  Sneezy
=item  Grumpy
=item  Keyser Soze
```

Элементы списка имеют неявный уровень вложенности:

The seven suspects are:

- Happy

- Dopey
- Sleepy
- Bashful
- Sneezy
- Grumpy
- Keyser Soze

Списки могут быть многоуровневыми. Каждый уровень указывается с помощью блоков `=item1`, `=item2`, `=item3` и т.д. В этом смысле блок `=item` является синонимом `=item1`. Например:

```
=item1  Animal
=item2     Vertebrate
=item2     Invertebrate

=item1  Phase
=item2     Solid
=item2     Liquid
=item2     Gas
=item2     Chocolate
```

Результат следующий:

- Animal
  - Vertebrate
  - Invertebrate
- Phase
  - Solid
  - Liquid
  - Gas
  - Chocolate

Обработчики Pod должны выдавать предупреждающее сообщение в случаях, когда разность между уровнями вложенности соседствующих блоков больше 1. Например, если за блоком `=item1` следует блок `=item3`.

Блоки `=item` не могут содержать вложенные списки. Это значит, что даже элементы с более низким уровнем вложенности *не* могут присутствовать внутри `=item` более высокого уровня.

```
=comment НЕВЕРНО...
```

```
=begin item1          --------------
The choices are:                    |
=item2 Liberty        ==< Level 2   |==<  Level 1
=item2 Death          ==< Level 2   |
=item2 Beer           ==< Level 2   |
=end item1            --------------

=comment ПРАВИЛЬНО...
=begin item1          ---------------
The choices are:                     |==< Level 1
=end item1            ---------------
=item2 Liberty        ==================< Level 2
=item2 Death          ==================< Level 2
=item2 Beer           ==================< Level 2
```

Нумерованные списки

Нумерованный список состоит из элементов, имеющих конфигурационный параметр `:numbered`.

```
=for item1 :numbered
Visito

=for item2 :numbered
Veni

=for item2 :numbered
Vidi

=for item2 :numbered
Vici
```

Приведенный код преобразуется в следующего вида текст:

1. Visito

    1.1. Veni

    1.2. Vidi

    1.3. Vici

Схема нумерации целиком определяется средствами подготовки того или иного формата вывода. Поэтому возможен следующий вид нумерации:

1. Visito

    1a. Veni

    1b. Vidi

    1c. Vici

или даже :

A: Visito

(i) Veni

(ii) Vidi

(iii) Vici

Эквивалентным свойству `:numbered` в свойствах элемента, является указание # первым символом в тексте элемента списка.

```
=item1  # Visito
=item2    # Veni
=item2    # Vidi
=item2    # Vici
```

В случаях, когда требуется использовать символ # первым без интерпретации его как признака нумерованного списка, используется код форматирования `V<>`:

```
=item V<#> introduces a comment
```

или явное отрицание нумерации:

```
=for item :!numbered
# introduces a comment
```

Следующие друг за другом элементы первого уровня `=item1` нумеруются последовательно. Их нумерация начинается заново, если их последовательность прерывается каким либо блоком Pod. В следующем примере, список прерывается параграфом:

```
The options are:

=item1 # Liberty
=item1 # Death
=item1 # Beer

The tools are:

=item1 # Revolution
=item1 # Deep-fried peanut butter sandwich
=item1 # Keg
```

Результат будет следующим:

The options are:

1. Liberty

2. Death

3. Beer

The tools are:

1. Revolution

2. Deep-fried peanut butter sandwich

3. Keg

Нумерация вложенных элементов (`=item2`, `=item3`, `=item4`) сбрасывается каждый раз, когда встречается элемент более высокого уровня вложенности.

Чтобы продолжить нумерацию `=item1`, после разрыва списка блоком Pod, достаточно указать свойство `:continued`:

```
=for item1
# Retreat to remote Himalayan monastery

=for item1
# Learn the hidden mysteries of space and time

I<????>

=for item1 :continued
# Prophet!
```

Указанный код будет преобразован в следующий текст:

1. Retreat to remote Himalayan monastery

2. Learn the hidden mysteries of space and time

????

3. Prophet!

Маркированные списки

Список элементов без указанного свойства `:numbered` интерпретируется как маркированный *(unordered)* список. Элементы таких списков отмечаются маркерами, так называемыми буллитами *(bullit)*[1].

Так к примеру текст

```
=item1 Reading
=item2 Writing
```

[1]Маркер списка (буллит) - типографский знак, используемый для выделения элементов списка. Маркер списка [http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D0%BA%D0%B5%D1%80_%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D0%B0]

```
=item3 'Rithmetic
```

может выглядеть следующим образом:

- Reading

    — Writing

        ¤ 'Rithmetic

Как и в случае нумерованных списков, стиль маркеров различных уровней вложенности возлагается на программу преобразования в формат вывода.

Параграфы в элементах списков

Чтобы в составе элемента списка использовать несколько параграфов, используется разграниченная *(delimited)* форма блока `=item`.

```
Let's consider two common proverbs:

=begin item :numbered
I<The rain in Spain falls mainly on the plain.>

This is a common myth and an unconscionable slur on the Spanish
people, the majority of whom are extremely attractive.
=end item

=begin item :numbered
I<The early bird gets the worm.>

In deciding whether to become an early riser, it is worth
considering whether you would actually enjoy annelids
for breakfast.
=end item

As you can see, folk wisdom is often of dubious value.
```

Результат будет следующий:

Let's consider two common proverbs:

4. *The rain in Spain falls mainly on the plain.*

   This is a common myth and an unconscionable slur on the Spanish people, the majority of whom are extremely attractive.

5. *The early bird gets the worm.*

   In deciding whether to become an early riser, it is worth considering whether you would actually enjoy annelids for breakfast.

As you can see, folk wisdom is often of dubious value.

Списки определений

Для создания списка определений используется блок `=defn`. Данный блок идентичен блоку `=item` в том, что серия последовательных блоков `=defn` явно определяет список. Отличие заключается в том, что при преобразовании в HTML используются тэги `<DL>...</DL>` вместо `<UL>...</UL>`.

Первая непустая строка содержимого блока интерпретируется как термин, а оставшееся содержимое - как определение термина.

```
=defn  MAD
Affected with a high degree of intellectual independence.

=defn  MEEKNESS
Uncommon patience in planning a revenge that is worth while.

=defn
MORAL
Conforming to a local and mutable standard of right.
Having the quality of general expediency.
```

Как и другие, элементы списков определений могут быть пронумерованы. Для этого используется свойство `:numbered` или символ # в начале строки:

```
=for defn :numbered
SELFISH
Devoid of consideration for the selfishness of others.

=defn # SUCCESS
The one unpardonable sin against one's fellows.
```

Псевдонимы

С помощью директивы `=aliases` можно определить ограниченный лексической областью видимости псевдоним для: части Pod текста, определения (мета) объекта в коде или даже куска программного кода. Данные псевдонимы используются в тексте Pod благодаря коду форматирования `A<>`.

Директива `=alias` является такой же фундаментальной, как `=begin` или `=for`. Она не может быть представлена в виде разграниченного *(delimited)* или параграфного *(paragraph)* блока.

Существует два вида псевдонимов: псевдонимы для макросов и контекстуальные *(contextual)* псевдонимы. Для обоих этих видов существует лексическая область видимости, ограниченная текущим Pod блоком.

Псевдонимы для макросов

Данная форма определения псевдонима наиболее простая: требуются только два аргумента. Первый - идентификатор, который обычно состоит из символов в верхнем регистре (*хотя данное условие необязательно*). Следующий аргумент состоит из одной или нескольких строк текста для подстановки.

В результате создается макрос Perl 6 с лексической областью видимости, который может быть вызван в процессе обработки документации. Для этого идентификатор макроса помещается в код форматирования A<>. В результате код форматирования заменяется на результат вызова макроса,а проще говоря, на текст, указанный при определении псевдонима.

Замещаемый текст начинается с первого непробельного (non-whitespace) символа, следующего за идентификатором псевдонима и продолжается до конца строки. Возможно определение замещаемого текста виде нескольких строк. Для этого в начале каждой следующей замещаемой строки ставиться знак = (*с тем же отступом от начала строки как и =alias*) и затем по крайней мере один пробел. Каждая из дополнительный строк использует отступ слева такой же как и первая строка (с директивой =alias) замещаемого текста.

Например, для текста:

```
=alias PROGNAME    Earl Irradiatem Evermore
=alias VENDOR      4D Kingdoms
=alias TERMS_URLS  =item L<http://www.4dk.com/eie>
=                  =item L<http://www.4dk.co.uk/eie.io/>
=                  =item L<http://www.fordecay.ch/canttouchthis>

The use of A<PROGNAME> is subject to the terms and conditions
laid out by A<VENDOR>, as specified at:

    A<TERMS_URL>
```

будет получен следующий результат:

```
The use of Earl Irradiatem Evermore is subject to the terms and
conditions laid out by 4D Kingdoms Inc, as specified at:

    =item L<http://www.4dk.com/eie>
    =item L<http://www.4dk.co.uk/eie.io/>
    =item L<http://www.fordecay.ch/canttouchthis>
```

Преимущества использования псевдонимов очевидны. Они могут быть использованы многократно в документации и достаточно отредактировать текст в определении псевдонима, чтобы эти изменения стали актуальными во всех частях документа.

```
=alias PROGNAME    Count Krunchem Constantly
=alias VENDOR      Last Chance Receivers Intl
=alias TERMS_URLS  L<http://www.c11.com/generic_conditions>
```

Контекстуальные псевдонимы

Если директива =alias указана только с одним аргументом (идентификатором),то создается контекстуальный псевдоним. В данной форме за директивой =alias должен следовать программный код Perl 6 (а не Pod блок).

Единственный аргумент директивы используется в качестве имени псевдонима и следующая за ним часть программного кода Perl 6 сохраняется в качестве замещаемого текста (как результат вызова макроса).

Программный код, следующий за `=alias`, является исполняемым реальным кодом программы. Парсер Perl 6 позволяет использовать эту часть кода в качестве замещаемого текста для псевдонима. Таким образом разработчик может цитировать в документации необходимую часть программного кода без копирования.

Если код программы, следующий за одноаргументной формой директивы `=alias`, представляет собой блок кода, ограниченный скобками, то качестве текста для подстановки используется содержимое блока *без* скобок.

Например, для следующего кода :

```
# This is actual code...

sub hash_function ($key)
=alias HASHCODE
{
    my $hash = 0;
    for $key.split("") -> $char {
        $hash = $hash*33 + $char.ord;
    }
    return $hash;
}

=begin pod
An ancient (but fast) hashing algorithm is used:

    =begin code :allow<A>
    A<HASHCODE>
    =end code

=end pod
```

Результат будет следующим:

```
An ancient (but fast) hashing algorithm is used:

    my $hash = 0;
    for $key.split("") -> $char {
        $hash *= 33;
        $hash += $char.ord;
    }
    return $hash;
```

Если за директивой `=alias` *не* следует открывающая скобки, то ожидается декларатор (например: `my`, `class`, `sub` и т.д.). Определенный с помощью одного из деклараторов объект становиться значением (raead-only) псевдонима. Например:

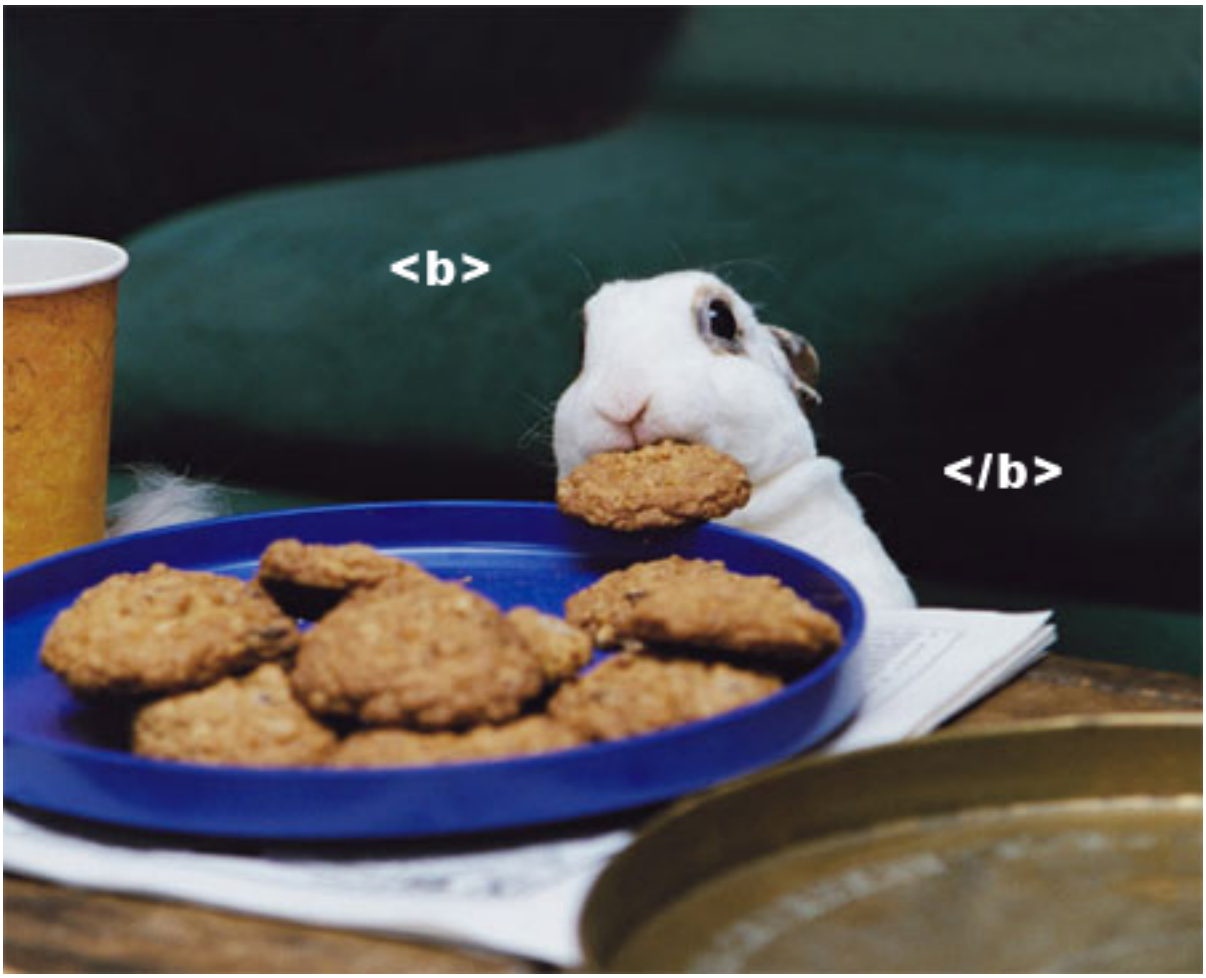
```
=alias CLASSNAME
class Database::Handle {
    =alias ATTR
    has IO $!handle;
```

```
        =alias OPEN
        my Bool method open ($filename?) {...}

        =alias DEFNAME
        constant Str DEFAULT_FILENAME = 'db.log';

        =for para
            Note that the A<OPEN.name> method of class A<CLASSNAME>
            stores the resulting low-level database handle
            in its private A<ATTR.name> attribute. By default,
            handles are opened to the file "A<DEFNAME>".

    }
```

Результат :

> Note that the `open` method of class `Database::Handle` stores the resulting low-level database handle in its private `$!handle` attribute. By default, handles are opened to the file "`db.log`".

Уровни значимости текста



Pod предоставляет три кода форматирования для отметки уровня значимости текста:

- Код `U<>` означает, что указанный текст является необычным или отличным от остального, что равнозначно *минимальному уровню значимости*. Отмеченный подобным образом текст обычно выделен подчеркиванием.
- Код форматирование `I<>` предназначен для выделения важного текста, что соответствует *основному уровню важности*. Содержимое этого уровня отображается курсивом или с помощью тэгов `<em>...<em/>`.
- Код `B<>` выделяет текст, который является базисным или фокусным для окружающего текста. Данным образом отмечается *фундаметальная значимость*. Подобный текст отображается жирным стилем или тэгами `<strong>...</strong>`.

Пример использования:

```
Текст может быть отмечен минимальным (подчеркнутым), ######## (курсивом) и #######
```

Блоки I/O

Pod [http://zag.ru/perl6-pod/S26.html] предусматривает специальные блоки для указания последовательности ввода и результатов вывода программ.

Это следующие блоки:

=input предварительно форматированный ввод с клавиатуры

=output экранный или файловый вывод результатов работы программы

Оба эти блока отображаются как есть, с сохранением форматирования и пробелов.

Подобно блокам `=code` , оба `=input` и `=output` блоки имеют неявный уровень вложения (*level of nesting, т.е. уровни вложения - предмет отдельного разговора, т.к. описаны они в специ-*

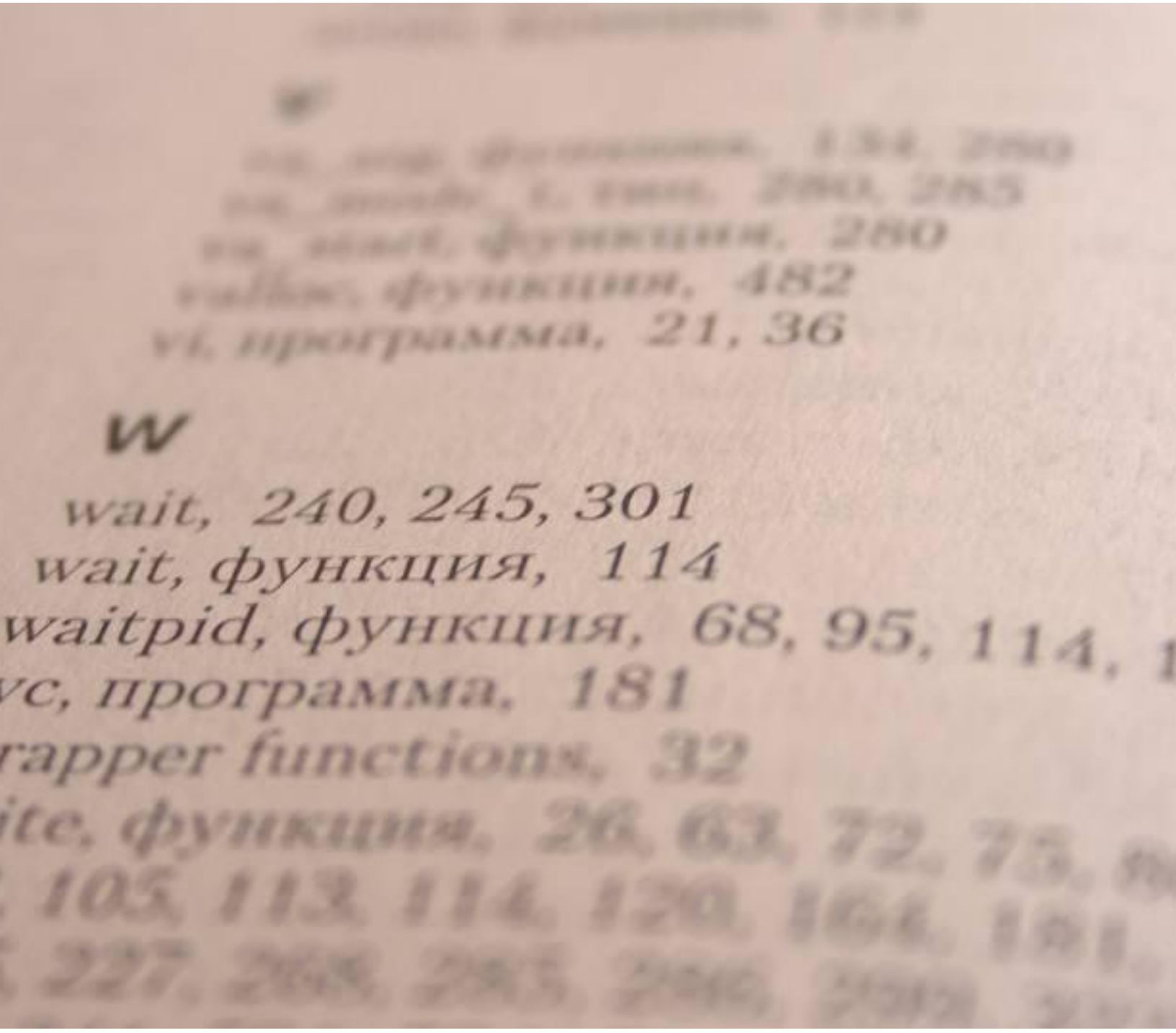
фикации вскользь и неопределенно). Блоки ввода-вывода, подобно блокам `=code`, отображаются с использованием шрифта фиксированной ширины (*fixed width font*), однако желательно, чтобы все три блока в документе отображались различными сочетаниями шрифт/ширина. Например : код - обычным шрифтом с засечками (*regular serifed*), ввод с клавиатуры - жирным sans-serif, а `=output` - обычным sans-serif.

В отличии от блока `=code`, оба блока допускают коды форматирования в их содержимом. В Pod имеются коды форматирования (K - ввод с клавиатуры и T - вывод на терминал) указывающие на ввод или вывод данных. Данная особенность привносит элемент интерактивности в документы, и делает возможным визуально демонстрировать процесс ввода данных и вывод результатов.

Пример демонстрации действий пользователя представлен ниже:

```
=begin output
  Name:    Baracus, B.A.
  Rank:    Sgt
  Serial:  1PTDF007

  Do you want additional personnel details? K<y>

  Height:  180cm/5'11"
  Weight:  104kg/230lb
  Age:     49

  Print? K<n>
=end output
```

Код форматирования X - индекс терминов



Содержимое кода X<> является элементом индекса. Текст внутри X отображается в итоговом тексте, а также используется как элемент индекса:

```
An X<array> is an ordered list of scalars indexed by number,
starting with 0. A X<hash> is an unordered collection of scalar
values indexed by their associated string key.
```

Возможно точное указание *индексируемого текста* и *элемента индекса*. Для этого используется вертикальная черта:

```
An X<array|arrays> is an ordered list of scalars indexed by number,
starting with 0. A X<hash|hashes> is an unordered collection of
scalar values indexed by their associated string key.
```

В таком случае, элемент индекса располагается после черты. Элементы индекса чувствительны к регистру. В примере "array" - индексируемый текст, а "arrays" - элемент индекса.

Для указания индексных уровней используется запятая:

```
An X<array|arrays, definition of> is an ordered list of scalars
indexed by number, starting with 0. A X<hash|hashes, definition of>
is an unordered collection of scalar values indexed by their
associated string key.
```

Можно указывать двое или больше элементов индекса для одного участка индексируемого текста. В качестве разделителя используется символ "точка с запятой":

```
A X<hash|hashes, definition of; associative arrays>
is an unordered collection of scalar values indexed by their
associated string key.
```

Индексируемый текст может быть пустым. Элемент индекса будет в таком случае "нулевой ширины":

```
X<|puns, deliberate>This is called the "Orcish Manoeuvre"
because you "OR" the "cache".
```

Это может оказаться полезным, кода необходимо привязать термин к абзацу или участку текста.

Код форматирования E - определение сущностей

Для вставки в Pod документ кодовой точки *(code point)* Unicode или ссылки на HTML5 символ, укажите необходимую сущность *(entity)*, используя код форматирования .

Если содержит число, оно интерпретируется как десятичное значение требуемой Unicode кодовой точки. Например:

```
  Perl 6 makes considerable use of « and ».
```

Можно также использовать явно двоичные, восьмеричные, десятичные и шестнадцатеричные числа (используя нотацию Perl 6 для указания формата представления):

```
Perl 6 makes considerable use of « and ».
Perl 6 makes considerable use of « and ».
Perl 6 makes considerable use of « and ».
Perl 6 makes considerable use of « and ».
```

Если содержимое отлично от числа, оно интерпретируется как имя символа Unicode (которое всегда в верхнем регистре) или именованная ссылка на символ HTML5. Например:

```
Perl 6 makes considerable use of #
 and #.
```

что эквивалентно:

```
Perl 6 makes considerable use of « and ».
```

Множество последовательно расположенных сущностей (в любом формате представления) могут быть указаны в одном коде , разделенных точкой с запятой:

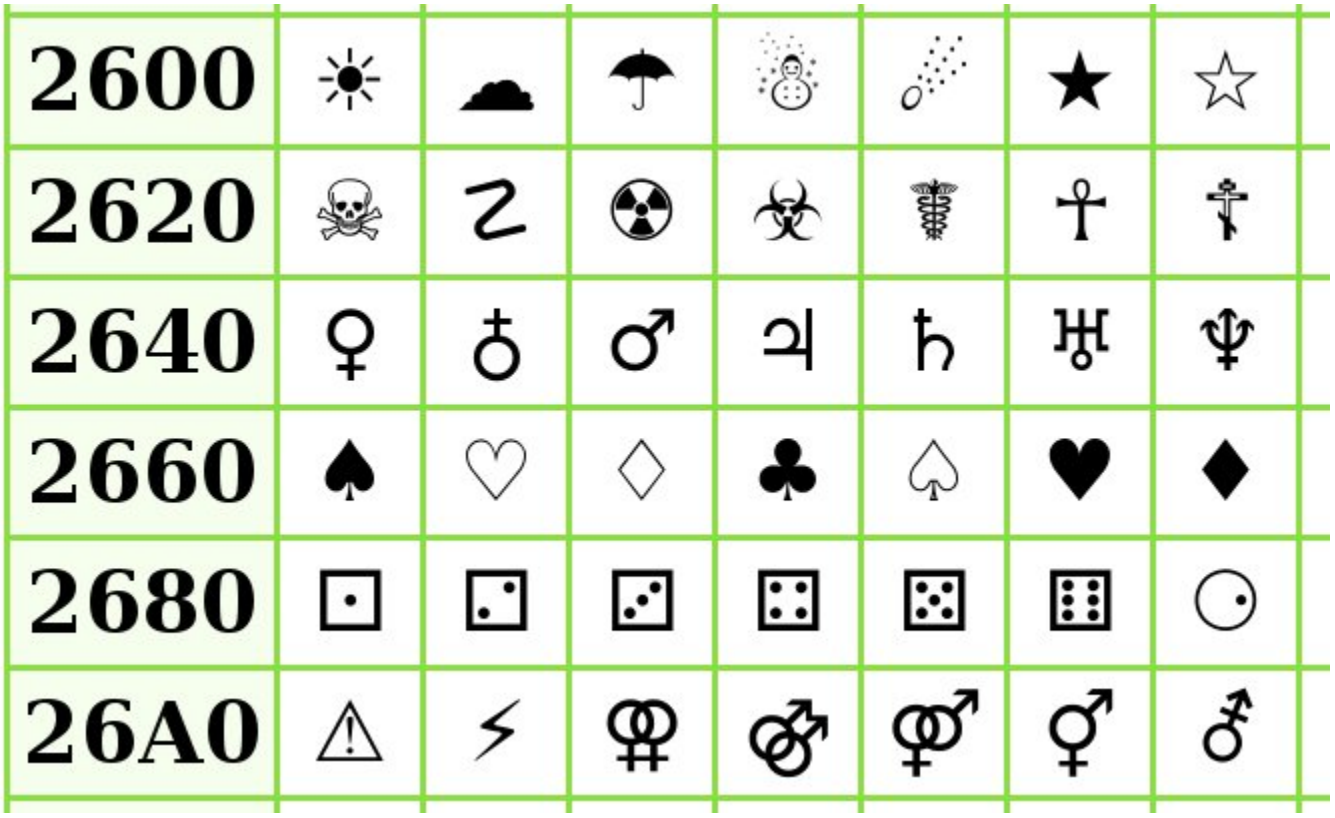
```
Perl 6 makes considerable use of #…».
```

Примеры

```
Get # I<(keyboard)> and type # I<(Euro)>.

Do : #,#,#,#,#.

Snow : # # #
```

- Get # *(keyboard)* and type # *(Euro)*.
- Do : #,#,#,#,#.
- Snow : ❄ ❅ ❆

В качестве источника я использовал следующие таблицы Unicode [http://www.tamasoft.co.jp/en/general-info/unicode.html].



Код форматирования N - примечания

Содержимое кода N<> является встроенным примечанием. Например:

```
Use a C<for> loop instead.N<The Perl 6 C<for> loop is far more
powerful than its Perl 5 predecessor.> Preferably with an explicit
iterator variable.
```

Трансляторы Pod [2] могут отображать содержимое примечаний разными способами: как сноски, как пояснения в конце книги или главы, как боковые панели с текстом, как всплывающие окна или подсказки, как разворачивающиеся элементы, и т.д. Однако они никогда не отображаются обычным текстом. Таким образом, предыдущий пример может быть отображен как:

```
Use a for loop instead.† Preferably with an explicit iterator
variable.
```

и далее:

```
Footnotes
† The Perl 6 for loop is far more powerful than its Perl 5
predecessor.
```

Код форматирования D - определения

Код форматирования D<> указывает, что содержащийся внутри текст является определением термина. При этом окружающий текст разъясняет данный термин. Данный код представляется собой строковый эквивалент блока =defn. Например:

```
There ensued a terrible moment of D<coyotus interruptus>: a brief
suspension of the effects of gravity, accompanied by a sudden
to-the-camera realisation of imminent downwards acceleration.
```

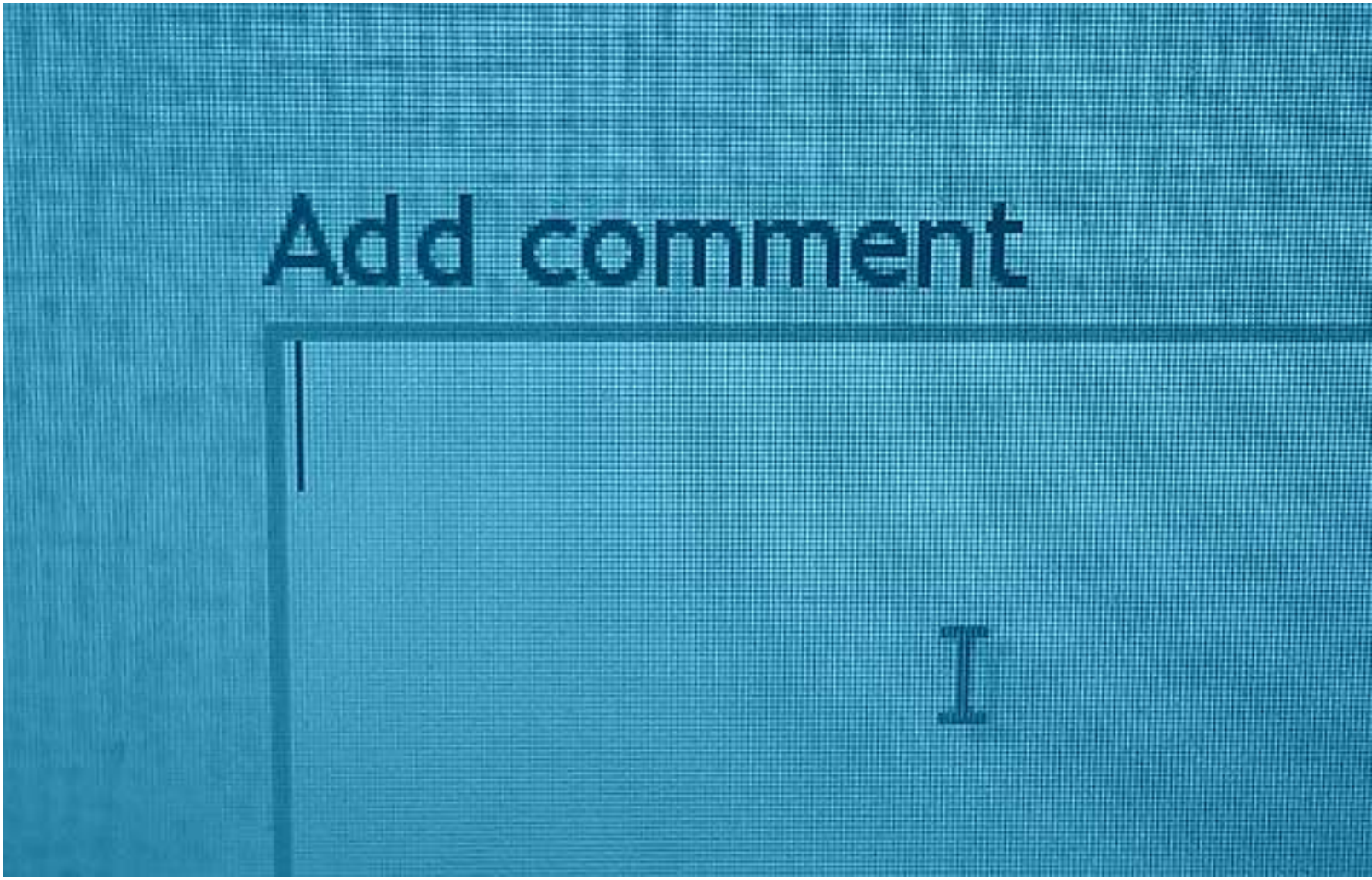
Определение может содержать синонимы. Они указываются после вертикальной черты и отделены друг от друга точкой с запятой.

```
A D<formatting code|formatting codes;formatters> provides a way
to add inline mark-up to a piece of text.
```

Определения обычно отображаются курсивом или отмечены тэгами <dfn>...</dfn>. На определения часто ссылаются из других частей гипертекстовых документов, где встречаются указанные термины (или любые из соответствующих синонимов).

[2] Трансляторы Pod - программы, преобразующие документы Pod в различные форматы. Например: *pod6xml*.

Код форматирования Z - комментарии



Код форматирования `Z<>` означает, что его содержимое является комментарием нулевой длины и не отображается. Например:

```
The "exeunt" command Z<Think about renaming this command?> is used
to quit all applications.
```

В формате Perl 5 POD код `Z<>` широко использовался для разбиения последовательности кодов разметки на составные части, чтобы избежать их интерпретации:

```
In Perl 5 POD, the Z<><> code was widely used to break up text
that would otherwise be considered mark-up.
```

Данный прием продолжает работать, однако, достичь результата сейчас легче благодаря "дословному" *(verbatim)* коду форматирования:

```
In Perl 5 POD, the  V<Z<>>  code was widely used to break up text
that would otherwise be considered mark-up.
```

Кроме того C<> также обрабатывает свое содержимое как "дословный" текст, что позволяет исключить необходимость в коде V<>:

```
In Perl 5 POD, the C<Z<>> code was widely used to break up text
that would otherwise be considered mark-up.
```

Код форматирования Z<> является эквивалентом блока =comment.

Комментарии как метки категорий

Большинство средств обработки Pod предоставляют механизм, позволяющий явно подключать или исключать отдельные блоки документации, если они соответствуют определенному критерию. Например, модуль экспорта документации *(renderer)* может быть проинформирован пропускать любой блок содержащий шаблон /CONFIDENTIAL/ *('КОНФИДЕЦИАЛЬНО')*. Подобный "невидимый маркер", может быть помещен внутри комментария Z<> в любом блоке и будет пропущен при обычной обработке. Например:

```
class Widget is Bauble
{
    has $.things; #= a collection of other stuff
    #={ Z<CONFIDENTIAL>
        This variable needs to be replaced for political reasons
    }
}
```

Код форматирования S - текст с неразрывными пробелами

Любой текст, заключенный в код S<> форматируется как обычно, сохраняя при этом пробельные символы (в том числе символ новой строки). Эти символы интерпретируются как неразрывные пробелы (кроме новой строки). Например:

```
The emergency signal is: S<
dot dot dot   dash dash dash   dot dot dot>.
```

Будет отформатирован следующим образом:

> The emergency signal is: dot dot dot dash dash dash dot dot dot.

вместо:

> The emergency signal is: dot dot dot dash dash dash dot dot dot.

Семантические блоки

Все блоки, имена которых состоят только из заглавных букв, зарегистрированы для стандартной документации, издательства и документирования программного кода. Так например, все стандартные компоненты документации по Perl или используемые в man страницах имеют зарезервированные аналоги имен в верхнем регистре.

Стандартные семантические блоки:

```
=NAME
=VERSION
=SYNOPSIS
=DESCRIPTION
=USAGE
=INTERFACE
=METHOD
=SUBROUTINE
=OPTION
=DIAGNOSTIC
=ERROR
=WARNING
=DEPENDENCY
=BUG
=SEEALSO
=ACKNOWLEDGEMENT
=AUTHOR
=COPYRIGHT
=DISCLAIMER
=LICENCE
=LICENSE
=TITLE
=SECTION
=CHAPTER
=APPENDIX
=TOC
=INDEX
=FOREWORD
=SUMMARY
```

Для указанных имен зарегистрированы соответствующие имена во множественном числе.

В модели документа данные блоки представлены заголовками первого уровня.

спецификаторы примеров

Pod предоставляет коды форматирования для указания примеров ввода, вывода, кода и мета синтаксиса:

- Код `T<>` предназначен для указания терминального вывода, т.е. текста выводимого программой. Данный текст отображается шрифтом фиксированной ширины или обрамляется тэгами `<samp>...</samp>`. Содержимое кода `T<>` всегда обрабатывается с сохранением пробелов (как если бы текст был обрамлен кодом `S<...>`). Код `T<>` является строковым эквивалентом блока `=output`.

- Код форматирования `K<>` указывает, что содержащийся внутри него текст, является клавиатурным вводом, т.е. некая последовательность, введенная пользователем. Такой текст отображается шрифтом фиксированной ширины (предпочтительно отличным от используемого для `T<>`) или выделяется тэгами `<kbd>...</kbd>`. Содержимое кода `K<>` всегда выводится с неразрывными пробелами. `K<>` является строковым эквивалентом блока `=input`.

- Содержимое кода `C<>` интерпретируется как программный код, т.е. текст, который может быть частью программы или спецификации. Данный текст обычно отображается шрифтом фиксированной ширины (желательно отличным от шрифтов кодов `T<>` или `K<>`)

или обрамляется тэгами <code>...</code>. Содержимое кода C<> транслируется в дословный *(verbatim)* текст с неразрывными пробелами. Код C<> является строковым эквивалентом блока =code.

Чтобы использовать коды форматирования внутри C<>, используется предварительное конфигурирование:

```
=begin para
=config C<> :allow<E I>
Perl 6 makes extensive use of the C<B<E<laquo>>> and C<B<E<raquo>>>
characters, for example, in a hash look-up:
C<%hashB<I<E<laquo>>>keyB<I<E<raquo>>>>
=end para
```

Чтобы использовать именованные символы () внутри *каждого* C<...> достаточно поместить вначале документа следующую строку:

```
=config C<> :allow<E>
```

- Код форматирования R<> используется для указания заменяемого элемента, маркера (placeholder) или метасинтаксической переменной. Данный текст обозначает элемент синтаксиса или спецификации, который в конечном итоге должен быть заменен на актуальное значение. Например:

```
The basic C<ln> command is: C<ln> R<source_file> R<target_file>
```

или:

```
Then enter your details at the prompt:

=for input
    Name: R<your surname>
      ID: R<your employee number>
    Pass: R<your 36-letter password>
```

Обычно заменяемые элементы отображаются наклонным шрифтом фиксированной ширины или обрамляются тэгами <var>...</var>. Гарнитура используемого шифта такая же как для кода C<>, за исключением случаев, когда код R<> находится внутри кодов K<> или T<> (или их эквивалентов: блоков =input или =output). Тогда используются шрифты соответствующих кодов.