Программирование на Perl 6

для любопытных

Оглавление

Краткое введение

Авторы языка Perl6 утверждают, что он охватывает несколько парадигм, являясь одновременно процедурным, объектно-ориентированным и функциональным. Кроме того, язык предлагает большой набор полезных и эффективных инструментов для анализа (parsing) текстовой информации. Впрочем, это же можно утверждать и в отношении многих других современных языков программирования. Очевидно, что Perl6 базируется на предыдущих версиях языка Perl. Вместе с тем утверждается, что это вообще-то новый язык со своим особым синтаксисом. Во всяком случае, Perl6 не имеет полной совместимости с предыдущей версией языка Perl5, а для освоения Perl6 даже не обязательно знакомство с предыдущими версиями.

Для работы требуется установить компилятор, именуемый Rakudo Perl6. Установочный файл можно свободно скачать с сайта языка. Файлы с исходным текстом на языке Perl6 должны иметь расширение ***.pl***, например, ***prob.pl*** (как и для предыдущих версий). Программу можно запустить из командной строки командой:

perl6 prob.pl

Допустимы также расширения ***p6*** и ***pl6***, которые позволяют отличать файлы с программами на Per6 от файлов с программами на других версиях Perl, которые тоже имеют расширение ***pl***.

Имеется интерактивный режим (Repl), вызываемый командой:

perl6

В командной строке появится приглашение ***(>)***, после чего можно вводить строки кода, которые будут немедленно выполняться с выдачей результата.

Я не сторонник длинных идентификаторов и считаю, что индивидуальные собственные имена надо применять только для многократно используемых и долго живущих переменных, функций и так далее. Если что-то существует только на нескольких строках кода, как это обычно бывает в примерах для изучения языка, идентификаторы должны быть как можно короче, лучше всего из одной буквы. Я везде буду придерживаться этого правила и, в частности, создаваемые функции почти всегда буду обозначать одной буквой f. Надеюсь, что читатель легко поймёт из контекста, где кончается жизнь одной функции с именем f и создаётся совсем другая функция всё с тем же именем f и на какую конкретно функцию я ссылаюсь в данный момент. Это же касается имён переменных, коллекций, и тому подобное.

1. Основы

1.1 Комментарии и строки кода

Одиночная строка комментариев начинается со знака **#** (как Ruby):

 # Это комментарий

Многострочный комментарий начинается с комбинации знаков **#`** и заключается в скобки, любые на выбор: ***(), [], {},*** <>:

#`<У лукоморья
дуб зелёный> - комментарий на двух строках.

Любое предложение (statement) должно заканчиваться точкой с запятой, даже если оно расположено на отдельной строке. Впрочем, на Repl точку с запятой на конце строки можно опускать. Блоки, тела функций, и тому подобное, заключаются в фигурные скобки и после закрывающей скобки точка с запятой не требуется.

Если выражение занимает несколько строк, на конце незаконченной строки надо ставить обратный слеш (\), который подавляет знак перевода строки:

***print($x**2 + ***
3 * $x);

В общем случае программа должна начинаться со строки

use v6;

Это для того, чтобы по ошибке не применить более раннюю версию Perl; при наличии этой строки в этом случае компилятор зафиксирует ошибку. Если подобный казус исключается, указанная строка не требуется.

1.2 Переменные

Все идентификаторы переменных (кстати, для всех версий Perl) должны начинаться со специального (служебного) знака (sigil): ***$, @, %, &,*** иногда ***::*** . Этот sigil указывает вид переменной: простая переменная (single), составная переменная (compound), подпрограмма (subrountine). После sigil идентификатор может содержать буквы и цифры, а также знак подчёркивания, тире и апостроф. Например, допустимы идентификаторы ***$my_name, @isn't, %double-click.***

Применение этих sigil в общем-то очень полезно, как мы увидим далее. С другой стороны необходимость ставить эти знаки перед каждым идентификатором увеличивает работу, а главное, ухудшает внешний вид кода, когда идентификаторов много от sigil пестрит в глазах. Но что делать, всегда выигрыш соседствует с потерями.

Но и это ещё не всё. При объявлении переменной требуется впереди ставить служебное слово ***my***, так называемый declarator (не ясно, как перевести, поэтому буду применять английское слово). Понимать это ***my*** надо так: он всегда должен присутствовать при объявлении новой переменной, коллекции и так далее; без него будет зафиксирована ошибка. Ну, а главное — это то, что переменная, со словом ***my*** доступна только в той области видимости, где она была объявлена, а также во внутренних областях. Кроме ***my*** применяются ещё и другие declarators такого вида и тогда область видимости будет определяться иначе. Более подробно об этом поговорим позже, а пока будем пользоваться только словом ***my***. (Мне кажется, что было бы

целесообразно sigil ***$*** и declarator ***my*** ввести «по умолчанию» и тогда программный код выглядел бы намного приятнее).

Итак, простая переменная без инициализации может быть объявлена так:

my $x; → (Any)

Так будет выглядеть объявление переменной ***$x*** в Repl. Интерпретатор возвращает в круглых скобках тип переменной. Стрелка (→) поставлена мной, здесь и везде далее я такой стрелкой буду заменять слова вроде «получим», «будет равно» и тому подобное. Repl этой стрелки не выводит.

Значит, в этом примере переменная ***$x*** получила неопределённый тип ***Any***. По желанию мы тип переменной можем указать, например:

my Int $y; → (Int)

Perl6 имеет обычный набор базовых типов. Нет необходимости здесь подробно их описывать, все особенности рассмотрим далее по ходу дела.

После объявления переменную можно инициировать значением указанного типа:

$y = 25;

Переменную ***$x***, имеющую тип Any можно инициировать значением любого типа:

$x = True;

$x = 3.5;

Тип переменной можно всегда изменить, объявив её заново.

Переменную можно объявить с одновременной инициализацией:

my $z = 2.75;

При этом тип не зафиксирован и переменную можно инициировать значением другого типа :

$z = "Hello";

Но можно тип задать:

my Rat $a = 2.75;

Теперь переменную ***$a*** уже нельзя инициировать значением другого типа, кроме ***Rat***, или надо объявлять её заново.

Отметим попутно, что Perl6 использует исключительно только кодировку UTF-8, а потому допустимо применение кириллицы, греческого шрифта и вообще любых символов. Например:

my $ю = 'Здравствуйте!';

print $ю; →Здравствуйте!

Всё прекрасно работает.

1.3 Числа

В Perl6 используется несколько типов числовых значений. Целые числа имеют тип ***Int.*** Числа с плавающей точкой представлены несколькими типами. Тип ***Rat*** представляет рациональные числа. Для чисел типа Rat имеются методы ***numerator*** и ***denominator*** (а ещё и метод ***nude***), позволющие получить числитель и знаменатель для представления в виде рациональной дроби. Фактически тип Rat обеспечивает вычисления без погрешности округления. Есть ещё типы чисел ***Num, Real*** и ещё целый ряд типов. Для работы с числами двойной длины, соответствующими типу Double в других языках, надо применять научную нотацию — ***0.253e0*** и это будет тип Num. Есть также возможность работы с комплексными числами - тип ***Complex***. Все встроенные математические функции возвращают результат двойной точности (тип Num). Имеются также два специальных значения, относящихся к типу Num: ***NaN*** и ***Inf***, мы познакомимся с ними по ходу дела. Весь набор базовых типов чисел в Perl6 кажется избыточным и для освоения всех нюансов требуется внимательно разобраться со всеми деталями. Все эти особенности, а также работу с комплексными числами здесь я рассматривать не буду, с этим не трудно разобраться самостоятельно.

Как видим, тип записывается с заглавной буквы (на самом деле все типы являются классами). Perl6 понимает и названия базовых типов (только базовых), с маленькой буквы: ***int, rat*** и так далее, но не будем здесь рассматривать эти детали.

В арифметических выражениях перевод чисел из одного типа в другой выполняется автоматически:

my Int $x = 5;
$x / 2; → 2.5

Допускается использовать в арифметических выражениях префиксный минус без дополнительных скобок:

2 + -5;→ -3

Оператор ***%*** возвращает остаток от деления:

7 % 2; → 1
7.5 % 2; → 1.5

Оператор возведения в степень - (********) :

2.5 ** 3.1; → 17.124347287269

Математические функции доступны по имени (без указания имени библиотеки):
sin(1.5); → 0.9974 - (часто я буду просто отбрасывать длинные ряды цифр)
log 3; → 1.0986 - скобки не обязательны

Знаки инкремента ***(++)*** и декремента ***(--)*** могут быть постфиксными и префиксными:
my Int $x = 5;
$x++; → 5 - возвращает аргумент
$x; → 6 - сама переменная ***$x*** изменилась
++$x; → 7 - возвращает результат инкремента
$x; → 7 - переменная изменилась

Допустима краткая форма:
$x += 5; → 12 - то же, что и ***$x = $x + 5;***
Такая форма допустима не только для арифметических операторов. Например, она допустима для оператора конкатенации ***(~)*** при работе со строками:
my Str $s = 'Hello';
$s ~= ' World'; → Hello World
Или для оператора возведения в степень:
my $x = 2.7;
$x**= 2; → 7.29

Perl способен автоматически трансформировать текст в число (а иногда и наоборот), если, конечно, выражение имеет смысл:
"3" ** 3; → 27
"2.7" ** "3.2"; → 24.00844

Для генерации случайных чисел (двойной длины) применяется встроенный метод ***rand***, позволяющий получать случайные числа в диапазоне от нуля до заданного числа:
my $x = 5.rand; → 4.87537994030983
my $x = 5.rand; → 0.403631583328298 - и так далее.
for 0 .. 4 { say 9.rand; } →
1.40645060386091
3.48666070359644
3.10811136698932
8.82565206313708
8.39242789051526

Для получения целых случайных чисел можно воспользоваться функцией изменения типа ***Int***:
for 0 .. 4 { say Int(9.rand); } → 0 8 7 1 1
Поскольку Int отбрасывает дробную часть числа без округления, то верхнее значение (9) мы никогда не получим. С учётом этого, получить случайные целые числа в заданном диапазоне, например от 4 до 10, можно так:
for 0 .. 4 { say 4 + Int(7.rand); } → 8 10 8 4 6
Впрочем, Perl6 имеет для этой цели специальный встроенный метод ***pick***:
for 0 .. 4 { say (4..10).pick; } → 8 6 10 8 10

Perl6 позволяет, конечно, работать также и с двоичными, восьмеричными и шестнадцатеричными числами.

1.4 Boolean

Переменные типа ***Bool*** могут принимать два значения: ***True*** и ***False.***
my Bool $x = False;
С помощью оператора ***(!)*** можно выполнять инверсию:
my $y = !$x; → True
$x; → False - при этом сама переменная ***$x*** не изменилась.

В условных выражениях все значения кроме нуля ***(0),*** пустой строки ***("")*** и пустых скобок ***(), {}, []*** трактуются, как ***True***.
Преобразование значения любого типа в значение типа ***Bool*** возможно с помощью оператора ***(?)***:
my $y = ?2.5; → True
my $z = ?""; → False
Есть также оператор ***so***, делающий то же самое:
my $x = so "hello"; → True
Кроме неопределённого значения (***Any***) есть ещё специальное значение ***Nil*** (ничто), применяемое в особых случаях. В условных выражениях ***Nil*** тоже трактуется, как ***False***:
my $a = Nil;
?$a; → False
Объявления типов тоже False:
?(Int); → False

Имеется обычный набор булевых операций: ***&&*** (синоним ***and***), || (синоним ***or***), ***!*** (синоним ***not***):

my $x = True;
my $y = False;
$x && $y; → False
$x || $y; → True

1.5 Строки

Строковые значения (тип ***Str***) имеют две разновидности: строка в двойных и строка в одинарных кавычках:

my $x = "Hello";
my $y = 'World';

Строка в двойных кавычках позволяет использовать интерполяцию и выполняет управляющие символы:

my $x = 2.5;
print("sin(2.5) = {sin $x}\n"); → sin(2.5) = 0.5984

Здесь мы применили оператор вывода на консоль ***print***. Для выполнения интерполяции достаточно интерполируемое выражение заключить в фигурные скобки. Кроме того, в примере использован управляющий символ перевода на новую строку - ***\n***. Для интерполяции одиночных переменных фигурные скобки можно опускать:

print("x = $x"); → x = 2.5

Обратный слеш (\) отключает интерполяцию и управляющие символы:

print("sin(2.5) = \{sin $x}\\n") → sin(2.5) = {sin 2.5}\n

При одинарных кавычках всё будет выведено «как есть»:

print('sin(2.5) = {sin $x}\n'); → sin(2.5) = {sin $x}\n

Отметим попутно, что в операторе ***print*** круглые скобки не обязательны (Perl6 позволяет опускать скобки довольно часто).

Вместо оператора ***print*** Perl6 позволяет использовать более универсальный оператор ***say***, обладающий дополнительными возможностями, с которыми мы ещё встретимся далее. В частности, оператор ***say*** всегда выполняет перевод на новую строку.

Для определения числа знаков в строке применяется метод ***chars***:

my $x = "Hello, World!";
$x.chars; → 13

Даже при использовании точечной нотации для вызова метода применима краткая форма:

$x .= chars; → 13 - то же самое, что и ***$x = $x.chars;***

Значение и тип самой переменной ***$x*** при этом изменятся.

Кроме типа Str есть ещё тип ***Stringy***, но пока не стоит вдаваться в эти детали.

Имеется много встроенных методов для работы с текстом. В частности, метод ***comb*** позволяет превратить текст в список, элементы которого представлены знаками текста и дальше с текстом можно работать, как со списком (массивом):

my $s = "голубые шарики";
my @a = $s.comb;
say @a[2..5]; → (л у б ы)

Здесь выражение ***@a[2..5]*** возвращает элементы списка с индексами в диапазоне ***2..5*** (о списках далее).

Метод ***uc*** переводит все знаки текста в верхний регистр, а метод ***lc***, соответственно — в нижний:

my $x = 'hello';
my $y = uc($x); → HELLO
my $z = lc($y); → hello

1.6 Списки, массивы

Списки и массивы в Perl6 представляют упорядоченные коллекции и во многом не отличаются друг от друга. Очень часто оба термина: список и массив можно применять к одному и тому же сорту коллекций, а операторы ***List*** и ***Array*** применяются на равных правах.

my @a is List; → () - создаётся пустой список
my @a is Array; → [] - создаётся пустой массив

Отличия между ними можно считать несущественными и далее по ходу дела я постараюсь отметить эту разницу. В некоторых руководствах по программированию используют только термин список, и я тоже буду поступать также в большинстве случаев.

В качестве sigil для списков использован знак **@**. При объявлении списка можно указать тип элементов:

my Int @x; → []

Получили пустой список ***@x***, элементы которого можно инициировать числами типа ***Int***:

@x = [1, 2, 3, 4, 5];

Если тип не указывать, то элементы могут быть разных типов:

my @y; → []
@y = [3, 0.75, True, "Bob"];

Можно объявление списка совместить с инициализацией элементов:
my @z = [2.3, 0.54, 0.75e6];
Квадратные скобки можно опустить, или использовать круглые. Если элементы имеют тип ***Str***, то имеется синтаксический сахар, позволяющий не вводить большое количество кавычек:
my @s = <Scala Ruby Fantom Perl6>; → [Scala Ruby Fantom Perl6]
То-есть, надо применить скобки (<>***)***. Можно при этом опустить и запятые, или оставить, если нравится.

Perl6 позволяет создавать списки с элементами разных типов:
my @s = 7, 3.56, 'aa', True; → [7 3.56 aa True]
При выводе на консоль всего списка можно применять обычный способ интерполяции с помощью фигурных скобок:
print "List @s = [{@s}]\n"; → List @s = [7 3.56 aa True]
Или с использованием квадратных скобок:
print "List @s = [@s[]]\n"; → List @s = [7 3.56 aa True]
При выводе элементы списка разделяются пробелами вместо запятых.

Для извлечения элемента по индексу принят обычный синтаксис:
@s[2]; → aa
Можно извлечь сразу несколько элементов:
my @s1 = @s[3, 0, 2];
@s1; → [True 7 aa]
При этом порядок индексов произвольный.

Элементы списка можно изменять:
@s[1] = "Marta";
@s; → [7 Marta aa True]
Также можно изменить сразу несколько элементов:
@s[3, 0, 1] = <Ruby Scala Groovy>;
@s; → [Scala Groovy aa Ruby]
В список можно добавлять новые элементы:
@s[5] = 77;
@s; → [Scala Groovy aa Ruby (Any) 77]
Здесь добавлен элемент с индексом ***5***, при этом компилятор добавил и пропущенный элемент с индексом ***4***, указав при выводе, что этот элемент имеет неопределённое значение (тип ***Any***).

Per6 позволяет извлечь элемент с несуществующим индексом, исключение не генерируется, а возвращается значение (Any):
@s[9]; → (Any)

Можно применять отрицательные индексы, но с дополнительным знаком ***(*)***:
my @a = 3, 4, 5, 6, 7;
@a[*-1]; → 7
@a[*-3]; → 5

Списки можно использовать для группового присваивания:
(my $x, my $y, my $z) = 9, 7, 5; - скобки должны быть круглые (а это значит, что здесь на самом деле список, а не массив).
$x; → 9
$y; → 7
$z; → 5

Так же можно выполнить обмен значениями двух переменных без ввода вспомогательной переменной:
($x, $z) = $z, $x;
$x; → 5
$z; → 9

Или воспользоваться встроенным методом ***reverse***:
(my $x, my $y) = 3, 5;
***($x, $y) .= reverse;*** - то же, что и ($x, $y) = ($x, $y).reverse
say ($x, $y); → (5 3)

Perl6 позволяет обмениваться элементами между списками. При этом могут быть самые разные комбинации:
my @s1 = 1, 2, 3, 4, 5;
my @s2 = <aa bb cc dd ee>;
my @i = 4, 1, 0;
@s1[@i] = @s2;
@s1; → [cc bb 3 4 aa]

Здесь мы в списке ***@i*** указали индексы изменяемых элементов списка ***@s1***. Из списка ***@s2*** элементы выбираются начиная с нулевого по порядку.

Встроенные методы ***head*** и ***tail*** позволяют извлечь первый и последний элемент списка, соответственно (в формате списка):
my @a = 5, 2, 8, 4;
@a.head; → (5)
@a.tail; → (4)

Круглые скобки здесь указывают на то, что методы возвращают не единичные значения, а списки с одним элементом. Методы ***head*** и ***tail***

могут, кроме того, принимать аргумент, определяющий, сколько элементов извлекается соответственно с начала или с конца списка:
my @a = 1, 2, 3, 4, 5;
@a.head(3); → (1 2 3)
@a.tail(3); → (3 4 5)

Метод ***first*** извлекает первый элемент, как значение:
@a.first; → 5
Есть ещё метод ***shift***, который тоже извлекает первый элемент, но при этом изменяется и сам исходный список:
my $x = @a.shift;
say $x; → 5
say @a; → [2 8 4]
Метод ***shift*** применяется при использовании списка в качестве стека.

Извлечь заданное число элементов вначале списка можно так:
@a[^2]; → (5 2)

Имеется много встроенных методов для работы со списками, в частности, метод ***elems*** возвращает размер списка (количество элементов):
say @a.elems; → 4
Или, синтаксический сахар:
say +@a; → 4
В дальнейшем мы рассмотрим многие из этих методов.

Для выполнения над списками операций, подобных суммированию всех элементов, можно применять такой синтаксис:
my @a = 1..5;
say [*] @a; → 120
say [+] @a; → 15
Операторы такого вида называются мета-операторами (reduce meta-operator). Perl6 предоставляет возможность создавать свои собственные мета-операторы, обсудим это далее. Имеются готовые для использования мета-операторы.

Конечно, списки могут быть вложенными, что позволяет, в частности, работать с многомерными матрицами.
my @a = [[1,2,3],[4,5,['a','b','c'],6]];
Скобки тут могут быть и круглыми, если нравится. Для извлечения элементов из вложенных списков применяются кратные индексы:
say @a[1][2][1]; → b

Встроенный метод ***flat*** позволяет распаковывать вложенные списки, но есть особенности его применения. Посмотрим на примере:
my @b = [[1,2,3],[4,5,6]];
my @c = @a.List.flat;
say @c; → [1 2 3 4 5 6]
Приходится дополнительно вставлять метод ***List***. Дело в том, что при инициализации создаётся массив (Array) @b, а метод flat применим только к спискам (List). Метод List предварительно трансформирует массив в список. Попробуем распаковать таким же образом массив @a:
my @b = @a.List.flat; → [1 2 3 4 5 [a b c] 6]
Значит, распаковываются только списки первого уровня вложенности. Чтобы распаковать полностью, надо проделать операцию ещё раз:
my @c = @b.List.flat; → [1 2 3 4 5 a b c 6]
На этих примерах видим, что между массивами и списками действительно есть разница.

Список можно создать также и при использовании знака ***$*** в качестве sigil:
my $a = (1,2,3,4,5);
$a[3]; → 4
$a.WHAT; → (List) - метод ***WHAT*** определяет тип объекта
Подобные вещи в Perl6 встречаются на каждом шагу, иногда это просто обескураживает.

Perl6 позволяет также использовать коллекции ***Seq*** и ***Set***:
my $a = Seq(1,2,3);
say $a; → (1 2 3) - это частный вид списков
my $b = Set(1,2,3,2);
say $b; → set(3, 1, 2) - эта коллекция не упорядочена и не имеет дублирующихся элементов.
Подробнее ***Seq*** и ***Set*** здесь не будем рассматривать.

1.7 Hash

Для коллекций этого вида пока нет одного общепринятого названия. Иначе hash ещё называют словом map, а в русском варианте — ассоциированным списком, отображением, словарём. Для краткости я везде буду пользоваться английским термином ***hash***. В качестве sigil для hash принят знак (***%***).

Элементы hash представлены парами ключ — значение, пары образуются с помощью знака ***(=>)***. Парой можно инициировать переменную с sigil $:

my $x = 'a' => 2;

Проверим тип переменной ***$x*** с помощью метода ***WHAT***:

$x.WHAT; → (Pair)

Значит, пара имеет тип ***Pair***.

Если в Repl объявить hash без инициализации, получим пустой hash:

my %h; → {}

Значит, коллекция hash ограничивается фигурными скобками. Объявление можно совместить с инициализацией пар конкретными значениями:

my %h = (1 => "Java", 2 => "Perl", 3 => "Python");

Как и для списков, при инициации скобки можно опустить, или применить квадратные скобки. Не следует использовать фигурные скобки, это хотя и сработает, но будет выдано предупреждение, его смысл пока не будем разбирать.

Hash ***%h*** имеет ключи типа ***Int*** и значения типа ***Str***. В общем случае тип ключей и значений может быть любым, допустимо также применение разных типов для элементов одного hash. Ключи в hash играют роль индекса в списках, по ключу можно извлечь значение или задать новое значение. Обычно ключи обозначают словом key, а значения — value. По смыслу в hash не может быть дублирующихся ключей, а значения могут быть любыми. Приведём пример hash с key-value разных типов:

my %h1 = ['a' => 'Java', 2 => 2.75, 0.3 => 'Python', 'hello' => True];

Для извлечения элемента по ключу используются фигурные скобки:

%h1{0.3}; → Python

Любое значение можно изменить и даже изменить его тип:

%h1{2} = False;

%h1; → {0.3 => Python, 2 => False, a => Java, hello => True}

Можно добавлять новые пары:

%h1{5} = 'world';

%h1; → {0.3 => Python, 2 => 1, 5 => world, a => 3, hello => 2}

Как и для списков, можно извлекать и изменять сразу группу значений:

%h1{2, 'hello', 'a'} = (1, 2, 3);

%h1; → {0.3 => Python, 2 => 1, a => 3, hello => 2}

Как и индексы для списков, ключи изменяемых пар можно задавать в виде списка, ну и так далее. Порядок следования пар в hash не фиксируется, поэтому можно наблюдать случайные перестановки пар.

Пусть переменная ***$x*** инициирована одной парой:

my $x = (5 => 'Foo');

Тогда с помощью методов ***key*** и ***value*** можно извлечь ключ и значение из пары, соответственно:

$x.key; → 5

$x.value; → Foo

Иногда, например, итераторы, обрабатывают hash последовательно пара за парой, и тогда методы ***key*** и ***value*** очень полезны. Подробнее это рассмотрим позже.

Метод ***keys*** возвращает список всех ключей hash:

my %h = [a => 'Java', 2 => 2.75, 0.3 => 'Python', hello => True];

%h.keys; → (a hello 0.3 2)

А метод ***values*** – список всех значений:

%h.values; → (Java True Python 2.75)

Как всегда, порядок произвольный.

Если ключи имеют тип ***Str***, кавычки для них можно опускать, они будут генерироваться автоматически (это уже использовано в предыдущем примере ***hello => True***):

my %h2 = one => 1, two => 2, three => 3;

При извлечении или изменении значений кавычки тоже можно опустить, если применить скобки (<>):

%h2<two>; → 2

Если и ключи и значения имеют тип Str, то, как и для списков, можно не ставить кавычек, запятых и связывающих знаков (=>), при использовании скобок (<>):

my %h3 = <k1 v1 k2 v2 k3 v3>; → {k1 => v1, k2 => v2, k3 => v3}

Есть ещё приём инициации пар с применением двоеточия, который применяется для случая, когда ключи имеют тип ***Str***:

my %h4 = (:aa(1), :bb(2), :cc(3)); → {aa => 1, bb => 2, cc => 3}

Позже мы применим этот способ для создания именованных аргументов у функций.

Очень удобный приём для создания hash с помощью двух списков, в которых представлены ключи и значения hash:

my @a = <aa bb cc dd>; - список ключей

my @b = 1, 2, 3, 4; - список значений
my %h; - создаём пустой hash %h
%h{@a} = @b; - инициируем элементы %h
say %h; → {aa => 1, bb => 2, cc => 3, dd => 4}

Оператор ***(,)*** - запятая, объединяет два hash в один:

```
my %h1 = 'a'=>1, 'b'=>2;
my %h2 = 'c'=>3, 'd'=>4;
my %h3 = %h1, %h2;
say %h3; → {a => 1, b => 2, c => 3, d => 4}
```

Ну и наконец, есть ещё особый синтаксис при использовании значений типа ***Bool***:

my %h5 = (:truey, :!falsey); → {falsey => False, truey => True}

Такой hash применяется в некоторых особых ситуациях.

1.8 Не изменяемость (immutable)

Все переменные (простые и составные), присутствующие в рассмотренных до этого примерах, были изменяемые (mutable). Во многих случаях предпочтительнее использовать не изменяемые (immutable) переменные. В некоторых языках программирования переменные immutable по умолчанию, а для преобразования их в mutable требуются дополнительные действия. В частности, такая практика применяется в функциональном программировании, использующем «чистые» (без побочных эффектов) функции. Использование не изменяемых переменных снижает риск непреднамеренных побочных эффектов.

В Perl6 для создания immutable переменной применяется оператор присваивания (***:=***) (вместо (=)):

```
my $x := 99;
```

Теперь попытка присвоить переменной ***$x*** новое значение приведёт к ошибке:

$x = 3; → Cannot assign to an immutable value

Не работает и оператор (***:=***), при этом текст ошибки будет другим:

$x := 7; → Cannot use bind operator with this left-hand side

Списки объявляются immutable таким же способом:

```
my @m := (1, 2, 3);
```

Попытка изменить хотя бы один элемент вызывает ошибку:

@m[0] = 55; → Cannot modify an immutable Int

В случае с immunable списками есть одна особенность — при инициализации элементов надо использовать круглые скобки (как в примере выше) или не применять никаких скобок:
my @m := 1, 2, 3;
Если же применить квадратные скобки, список будет mutable:
my @m := [1, 2, 3];
@m[1] = 77; → 77 - здесь ошибки нет.
@m; → [1 77 3]
Подобным же образом создаются и не изменяемые hash.

На самом деле оператор присваивания ***(:=)*** в отличие от оператора ***(=)***, не создаёт копию переменной, а просто закрепляет с этой переменной связь. Разберёмся с этим на примере:
my $x = 555;
my $y = $x;
say $y; → 555 - здесь $y является копией $x.
$x = 777; - изменим переменную $x.
say $y; → 555 - при этом переменная $y не изменилась.
Проделаем теперь всё это же с оператором (:=):
my $a = 555;
my $b := $a; - использовали оператор (:=)
say $b; → 555
$a = 777; - изменили переменную $a
say $b; → 777 - переменная $b изменилась, поскольку она постоянно привязана к переменной $x.
Естественно, что применяя оператор присваивания (:=) надо это обстоятельство постоянно иметь ввиду.

Есть также возможность создавать immutable переменные с помощью директивы constant, то-есть, создавать константы:
my constant $x = 7;
Попытка присвоения переменной ***$x*** другого значения вызовет ошибку.

Впоследствии мы ещё встретимся с применением immutable переменных.

1.9 Подпрограммы (subroutine)

Вообще говоря, subroutine по существу не отличаются от функций и в Perl этот термин используется, по-видимому, только по традиции. Можно, конечно, пользоваться и термином «процедура». В

дальнейшем будем в основном применять термин функция и реже - процедура, а название метод оставим для варианта в точечной нотации. Теперь часто применяют два термина — аргументы и параметры, чтобы различать список переменных функции и то, что ей передаётся в качестве значений этих переменных. Мне кажется, что применение двух терминов только прибавляет неопределённости, и лучше говорить аргументы и их значения. В основном я буду использовать только слово аргументы, полагаясь на то, что из контекста обычно ясно о чём именно идёт речь.

В качестве функции в Perl6 можно рассматривать любой блок, поскольку блоку можно присвоить имя. Например:

```
my $x =  {
        my $y = 3;
        $y * 7;
}
print $x(); → 21
```

Итак, объявленная таким образом переменная ***$x*** может использоваться, как процедура (функция), для этого достаточно при её вызове поставить пустые круглые скобки. Однако, такой процедуре нельзя передать параметры и, значит, полноценной функцией её признать нельзя. Можно, правда, поступить так:

```
my $t = 9;
my $x =  {
        my $y = 3;
        $y * $t;
}
print $x(); → 27
```

Объявленная вне блока «глобальная» переменная ***$t*** доступна внутри блока, но «локальная» переменная ***$y***, объявленная внутри блока, за пределами блока не видна.

При объявлении «настоящей» процедуры (функции) применяется ключевое слово ***sub***, а тело заключается в фигурные скобки:

```
sub f($x)  {
        print "Hello, {$x}\n";
}
```

Отступы в Perl6 не играют никакой роли, весь этот текст можно было бы записать в одной строке. Для вызова функции достаточно назвать её имя и передать аргументы:

f("World"); → Hello, World!
Аргументы можно не заключать в скобки, при их отсутствии пустые скобки ставить тоже не обязательно. Как видим, для идентификатора функции не используется sigil. Функция ***f*** может принимать аргументы любого типа:
f 3.5 → Hello, 3.5
Конечно, для аргументов можно и указать тип:
sub f(Str $x) { print "Hello, {$x}\n"; }
f 'Peter'; → Hello, Peter

Функции можно передавать необязательные (optimal) аргументы, для этого достаточно после их идентификаторов поставить знак (***?***). При вызове этим аргументам можно передать значения, как обычным, но можно не передавать и тогда они останутся неопределёнными (***Any***):
sub f($x?, $y?) { return [$x, $y]; }
Функция ***f*** принимает два необязательных аргумента ***$x?, $y?*** И возвращает их в формате списка:
say(f()); → [(Any) (Any)] - (здесь оператор ***print*** не справляется)
say(f(3, 5)); → [3 5] - оператор ***print*** выведет ***3 5***
Оператор ***return*** можно опускать — всегда возвращается последнее вычисленное значение в блоке.
sub f($x?, $y?) { [$x, $y]; } - будет то же самое.
Впрочем, часто использование return делает код более понятным. Бывает также, что требуется по условию прервать вычисления где-то в середине блока, и тогда директива return может быть необходима.
say(f(7)); → [7 (Any)]
say(f(Any, 9)); → [(Any) 9] -неопределённое значение можно задавать

Аргументам можно давать значения по умолчанию:
sub f($x = "World") { print("Hello, ", $x, "!\n"); }
f; → Hello, World!
f "Peter"; → Hello, Peter!
Следовательно, если при вызове функции аргумент не вводить, будет принято заданное значение по умолчанию; но можно аргумент ввести, и тогда будет использовано вводимое значение.

Аргументы могут быть именованными, для этого используется синтаксис hash:
```
sub f($x, :$y, :$z) {
    print $x + $y + $z;
```

```
}
f(1, z => 2, y => 3); → 6
```

Здесь ***$x*** – обычный аргумент, а ***$y*** и ***$z*** – именованные (перед sigil ставится двоеточие). Значения для именованных аргументов передаются в форме пар. При этом не требуется указывать sigil. Достоинство этого способа в том, что аргументы можно вводить в произвольном порядке по имени, что удобно, особенно при их большом количестве. При вызове функции можно применять ещё и такой синтаксис:

```
f(2.5, :y(0.5), :z(2.0)); → 5
```

Можно именованному аргументу одновременно присвоить ещё и значение по умолчанию:

```
sub f(:$x = 13) {
    say $x ** 2;
}
f; → 169
f x => 5; → 25
```

Функция может иметь не зафиксированное (неопределённое) количество аргументов. Для этого применяется список с дополнительным знаком (*****) впереди (иногда такие списки называют slurpy — параметрами от слова схватывать, заглатывать):

```
sub f($x, *@y) {
    print($x, ' | ', @y);
}
f(1, 2, 3, 4, 5); → 1 | 2 3 4 5
```

Здесь ***@y*** может принимать любое число значений, но располагаться эта переменная всегда должна в конце списка аргументов. При вызове функции все аргументы вводятся подряд, через запятую. В теле функции ***@y*** рассматривается, как список, что можно видеть по выведенному результату. Для большей ясности покажу, чем отличается этот вариант от применения просто списка:

```
sub f($x, @y) { print($x, ' | ', @y); }
f(5, [1,2,3]); → 5 | 1 2 3
```

В этом варианте функция принимает два аргумента: простую переменную ***$x*** и список ***@y*** и так они и должны вводиться при вызове. В остальном всё будет также.

Группу аргументов функции можно объединять в список и передавать их при вызове в формате списка:

```
my @m = 1, 2, 3;
sub f($x, $y, $z) {
        print("$x, $y, $z");
}
f(|@m); → 1, 2, 3
```

Значит, для этого надо поставить знак (|) перед списком. Можно, конечно, передавать аргументы и так:

```
f(|[5, 6, 7]); → 5, 6, 7
```

Напоминаю, что в операторе ***print("$x, $y, $z");*** использована интерполяция (для одиночных переменных фигурные скобки можно опускать) для того, чтобы вывелись запятые.

Аналогично можно применить hash для именованных аргументов:

```
sub f($x, :$y, :$z) { $x + $y + $z; }
```

Здесь один аргумент позиционный и два именованных. Используем hash:

```
my %h = (y => 7, z => 2);
say f(3, |%h); → 12
```

Значит, для распаковки hash достаточно поставить перед ним знак ***(|)***.

Аргументы могут иметь больше одного имени. Учитывая, что Perl6 позволяет применять для идентификаторов кириллицу, создадим функцию, которая может принимать именованные аргументы как на английском, так и на русском языке:

```
sub f(:color(:цвет($c)))  {
        say "color (цвет) $c";
}
f(color => "красный"); → color (цвет) красный
f(цвет => "синий"); → color (цвет) синий
```

Тут довольно замысловатый синтаксис, но техника, я думаю, простая и понятная.

Можно также принудительно задать тип возвращаемого функцией результата. Для этого применяется служебное слово ***returns*** (не return) перед блоком. Например, посмотрим на функцию, которая просто возвращает свой аргумент:

```
sub f($x) returns Str  { $x; }
```

Эта функция формально может принимать аргумент любого типа, но возвращать она должна только значение типа ***Str***.

```
say f 'Hello'; → Hello
```

здесь всё в порядке

say f 5; → сообщение об ошибке — функция не может иметь результат типа ***Int***.

Слово returns можно заменять такой стрелкой **-->** при следующем синтаксисе:

sub f($x, $y --> Int) { $x * $y; }
f 2,3; → 6

Функция f должна возвращать результат типа Int.

Как и в любом современном языке, в Per6 имеется много встроенных функций (методов). Большинство из них можно применять в точечной нотации.

my $x = "Scala";
$x.say; → Scala - say, как и ***print,*** можно использовать и в точечной нотации.
$x.uc; → SCALA - метод ***uc*** (от слов upper case – верхний регистр) переводит все знаки в тексте в верхний регистр. Аналогично:
$x.lc; → scala - метод ***lc*** (lower case) выполняет обратное действие.

Методы можно применять к объекту в форме цепочки, когда следующий метод применяется к результату предыдущего метода. Возьмём для примера список:

my @s = <ruby groovy java fantom>;

И применим к нему цепочку методов:

@s.sort.uc.say; → FANTOM GROOVY JAVA RUBY

Здесь метод ***sort*** выполняет сортировку списка. Этот пример показывает также, что в Perl6 многие методы применимы как к одиночным объектам, так и к коллекциям, выполняя роль итераторов. Об итераторах поговорим позже.

По умолчанию аргументы функций всегда immutable, то-есть, их нельзя изменять в теле функции:

sub f($x) { $x += 3; }
f 5; → Cannot assign to a readonly variable or a value (нельзя изменять переменную, имеющую статус «только для чтения»).

Для того, чтобы иметь возможность изменять аргумент, надо заменить его копией с помощью директивы ***is copy***:

sub f($x is copy) { $x += 3; say $x;}
my $y = 5;
f($y); → 8
say $y; → 5

При этом, как видим, переменная ***$y*** не изменила своего значения, поскольку функция использовала только её копию.

Имеется также возможность объявлять аргумент с директивой ***is rw,*** что означает «переменная доступна для чтения и записи». Посмотрим на примере:

```
sub f($x is rw) { $x += 3; say $x;}
my $y = 5;
f($y); → 8
say $y; → 8
```

Как видим, всё работает также, за исключением того, что переменная ***$y*** изменила своё значение, поскольку теперь копия не создавалась.

Посмотрим, как mutable аргумент можно использовать, например, в цикле ***while***:

```
sub f(int $n is rw) {
        while $n > 0 {
                print $n ~ " ";
                $n--;
        }
        say ' конец';
}
my int $y = 5;
f($y); → 5 4 3 2 1  конец
```

Здесь, кроме всего прочего, потребовалось обязательное объявление типа переменных ***$n*** и ***$y***, причём название типа должно быть с маленькой буквы. Разбор всех этих деталей не доставляет большого удовольствия.

1.10 Сигнатура и самоанализ функций

В Perl6 иемется возможность проанализировать список аргументов функции и узнать свойства этих аргументов, такие, как необязательный (optional), именованный (named), неопределённое количество (slurpy), тип (type). Сам список аргументов вместе с их свойствами называется сигнатурой (signature). Давайте разберёмся со всем этим на простом примере. Создадим функцию, переводящую градусы в радианы:

```
sub f(Int $x, Real :$y = pi) { $y * $x/180; }
```

Функция ***f*** имеет два аргумента: позиционный — целое число ***$x***, и необязательный (на это указывает двоеточие) именованный — число типа Real ***$y***. Функцию можно вызвать так:
say f(90); → 1.5707963267949
или так (без математического смысла):
say f(90, y => 5); → 2.5

Метод ***signature*** возвращает сигнатуру функции (список аргументов в своём исходном виде):
say &f.signature; → (Int $x, Real :$y = 3.14159265358979e0)
Метод ***params*** возвращает список аргументов.
```
my @p = &f.signature.params;
say @p; → [Int $x Real $y = 3.14159265358979e0]
```
Внешне результаты похожи, но обратите внимание на отсутствие запятой во втором случае. С помощью метода ***elems*** можно узнать количество аргументов в списке:
say @p.elems; → 2
С помощью цикла ***for*** и встроенных методов ***name, type, named, slurpy*** и ***optional*** можно узнать все свойства аргументов:
```
for @p {
    say "Name: ", .name;
    say " Type: ", .type;
    say " named? ", .named ?? 'yes' !! 'no';
    say " slurpy? ", .slurpy ?? 'yes' !! 'no';
    say " optional? ", .optional ?? 'yes' !! 'no';
}
```
В результате получим такие данные:
```
Name: $x
Type: (Int)
named? no
slurpy? no
optional? No

Name: $y
Type: (Real)
named? yes
slurpy? no
optional? Yes
```

Обратите внимание на способ применения методов в цикле for – здесь переменная ***$_*** используется неявно; вообще-то можно было бы написать:

say "Name: ", $_.name;

и так далее, что означало бы то же самое.

1.11 Функции, как значения

Имя функции можно легко изменить. Для этого надо объявить новую переменную с sigil ***&*** и выполнить присваивание:

```
sub f($x) { $x ** 2; }
my &f1 = &f;
```

Перед изменяемым названием процедуры тоже надо поставить знак ***&***. Теперь функцию ***f1*** можно использовать точно также, как и исходную ***f***:

f1(5); → 25

Этот пример показывает, что функции в Perl6 можно рассматривать, как значения и, в частности, инициировать ими переменные. В дальнейшем мы рассмотрим это свойство подробнее и увидим, какие важные следствия оно может иметь.

В дальнейшем мы также подробно рассмотрим анонимные функции и их использование. Пока покажу только, как анонимную функцию можно применить для создания процедуры:

```
my &f = sub { say "Анонимная функция !"; }
```

Здесь справа от знака равенства — анонимная функция без аргументов. Анонимная функция объявляется также, как и именованная, только не указывается имя. Выполнив присваивание вновь объявленной переменной ***&f***, получаем полноценную процедуру ***f***:

f; → Анонимная функция !

Анонимная функция может иметь аргументы, синтаксис обычный:

```
my &f = sub ($x) { $x ** 2; }
f(3); → 9
```

Для создаваемой процедуры ***f*** при этом объявлять аргументы не требуется.

1.12 Ввод вывод

Perl6 имеет целый набор операторов вывода на консоль: print, printf, sprintf, say. Мы уже использовали операторы print и say.

Оператор форматированного вывода ***printf*** (а также ***sprintf***) позволяет оформить выводимую информацию в подходящем виде:

my $y = 34.050078;

printf("%15.4f\n", $y); → 34.0501

Здесь 15 определяет общее количество позиций, отведённых для результата, а четвёрка — число выводимых знаков после десятичной точки. При этом выполняется округление. Буква ***f*** указывает, что выводится результат типа ***Rat***. Выводимое число прижимается к правому краю выделенной области. Можно прижать число и к левому краю, для чего достаточно поставить знак минус:

printf("%-15.4f\n", $y); → 34.0501

Имеется также краткая форма оператора форматированного вывода — метод ***fmt***, который можно использовать с любым оператором вывода - say, print, printf, sprintf:

my $x = 35;

say $x.fmt('%7d'); → 35

Можно пробелы впереди числа заменить на нули:

say $x.fmt('%07d'); → 0000035

Метод fmt очень удобен для вывода коллекций:

my @a = 1, 2, 3;

say @a.fmt('%07d'); → 0000001 0000002 0000003

Дополнительно можно указать нужный разделитель между элементами:

say @a.fmt('%07d',' / '); → 0000001 / 0000002 / 0000003

say @a.fmt('%7d',', '); → 1, 2, 3

Пример вывода для hash:

say { Яблоки => 50, Апельсины => 100 }.fmt('%s цена %d рублей');

Результат будет иметь вид:

Яблоки цена 50 рублей

Апельсины цена 100 рублей

Метод fmt имеет и многие другие возможности, нет смысла рассматривать их всех здесь.

Добавим ещё, что имеется оператор вывода на консоль ***dd***, который автоматически выводит название переменной и её тип:

my $x = "Hello";

dd $x; → Str $x = "Hello"

С оператором dd также можно применять метод fmt.

Для чтения из файла на диске применяются операторы: ***open*** для открытия файла, ***lines*** для чтения и ***close*** для закрытия файла. Пусть требуется прочитать файл с названием ***prob.pl***. Открываем файл:
my $f = open("prob.pl", :r);
Значит, создаём вспомогательную переменную ***$f***, которая будет содержать информацию об открытом файле, конкретное её содержание можно посмотреть командой ***say $f;***. Имя файла может быть относительным, как у нас, или полным. Параметр ***:r*** указывает, что файл открывается для чтения. С помощью метода ***lines*** читаем файл в список:
my @a = $f.lines;
Список ***@a*** будет содержать весь текст файла, каждый элемент списка содержит одну строку текста:
say @a[0]; → sub f($x) {
После чтения файл автоматически закрывается и чтобы выполнить повторное чтение его надо открыть снова.

Метод lines может принимать аргумент, определяющий количество прочитанных строк:
my @a = $f.lines(2);
Теперь список ***@a*** будет содержать только две первые строки файла. При этом файл не закрывается и можно повторить команду чтения для нужного количества следующих строк. При прочтении последней строки файл опять закроется. Если чтение выполняли не до конца файла, его надо закрыть командой
$f.close; → True
Если всё нормально, метод вернёт значение True.
Вместо метода ***lines*** можно использовать метод ***get***, читающий очередную строку файла:
my @a = $f.get.split(' '); → [sub f($x) {}]
Здесь метод split использован для разбивки строки на элементы, а аргумент метода определяет, по какому символу требуется разбить строку (у нас по пробелу).
say @a[1]; → f($x)

Есть ещё метод ***slurp-res***t (разнообразие в Perl6 бесконечно), читающий весь файл единой строковой переменной:
my $d = $f.slurp-rest; - получим весь текст файла как одну строку. После этой команды файл надо закрывать.

Для записи в файл его надо открыть с оператором ***:w***:

my $f = open("prob.pl", :w);
Запись в файл выполняется любой командой вывода, например, ***say***:
$f.say("line to be written to the file"); → True
$f.close;
Аргумент метода ***say*** будет выведен в файл; если всё нормально, возвращается значение True. Затем файл надо закрыть. В этом случае старое содержимое файла будет стёрто. Есть возможность выводить информацию в файл построчно

Кроме рассмотренных приёмов есть ещё множество способов выполнять чтение и запись в файлы, рассматривать все их нет нужды, всё это достаточно просто.

2. Ветвления и циклы

2.1 Условный оператор if

Синтаксис условного оператора ***if*** ничем не отличается от принятого в большинстве других языков и в общем случае имеет вид:
if (условное выражение) {блок} else {блок}
Условное выражение можно заключать в круглые скобки (не обязательно), а блок имеет обычный вид — в фигурных скобках.
my $x = 5;
if ($x > 3) { print 'aa' } else { print 'bb' } → aa
Чтобы создать цепь проверок, применяется оператор ***elsif***:
if ($x > 7) { print 'aa' } elsif ($x > 4) { print 'bb' } else { print 'cc' } → bb

Ветвь ***else*** не обязательна:
$x === "Hello";
if ($x === "Hello") { print "Yes"; } → Yes
if ($x === "World") { print "Yes"; } → ()
В этом случае, если условие даёт ***Fals***e, условный оператор возвращает ***()***; напоминаю, что пустые скобки трактуются, как ***False***:
?(); → False

Обычно условный оператор ***if*** в Perl6 не возвращает результата и используется только для получения побочного эффекта. Однако, есть возможность заставить его возвращать результат. Для этой цели применяется дополнительный оператор ***do*** при следующем синтаксисе:

my $x = 2;
my $y = do if ($x <= 3) { $x ** 2 } else { $x ** 3 } → 4
$x = 5;
my $y = do if ($x <= 3) { $x ** 2 } else { $x ** 3 } → 125

Поскольку в Perl6 любое значение можно трактовать, как булево, на месте условного выражения может стоять любое значение, переменная или выражение:

if $x * 2 { print 'aa' } else { print 'bb' } → aa
if $x * 0 { print 'aa' } else { print 'bb' } → bb

Во втором случае выражение возвращает ***0***, который трактуется, как ***False***.

Есть также postfix версия, когда оператор ***if*** с условием находятся в конце предложения:

s***ay "Hello" if 3 >= 2; → Hello***
say "Hello" if 3 >= 5; → ()

Если условие не выполнено, здесь тоже возвращается ***()***. Как видим, в этом варианте фигурные скобки для блока можно опустить, хотя можно и поставить:

{ say "Hello" } if 3 >= 2; → Hello

Кроме if имеется ещё оператор unless, который работает «наоборот»:

unless (3 > 7) { say 'aa' }; → aa
unless (3 < 7) { say 'aa' }; → ()

Этот оператор не имеет ветви else. Unless также может быть postfix:

say 'aa' unless (3 > 7); → aa

Как и в других языках, имеется тернарный условный оператор, использующий знаки ***(??)*** и ***(!!)***:

(3 > 2) ?? say 'aa' !! say 'bb'; → aa
(3 < 2) ?? say 'aa' !! say 'bb'; → bb

Этот оператор способен возвращать значение без дополнительного do:

my $x = 2.5;
my $y = ($x < 8) ?? $x * 2 !! $x * 3;
$y; → 5
$y = ($x > 8) ?? $x * 2 !! $x * 3;
$y; → 7.5

Этот вариант не требует заключать блоки в фигурные скобки. Очевидно, что краткая форма условного оператора очень удобна, и потому популярна у программистов.

2.2 Переключатель

Часто данную конструкцию называют «сопоставление с образцом» (pattern matching). В большинстве языков программирования переключатель реализуется с помощью ключевых слов switch-case. По-видимому, вследствие постоянного стремления авторов Perl быть оригинальными, они использовали слова given-when. Поскольку в Perl6 на месте образца может быть также и условное выражение, то слово when может быть действительно здесь более уместно (в значении «как только»). Для разнообразия я приведу пример в форме законченной программы:

```
sub f($x)  {
        given $x  {
                when $x < 5 {print "$x < 5\n";}
                when $x ~~ "5" {print "$x = 5\n";}
                when $x > 5 {print "$x > 5\n";}
        }
}
print "Введите число: ";
my $x = get;
f($x);
print "Нажмите любую клавишу";
my $xxx = get;
```

Здесь стандартная функция ***get*** позволяет вводить строку текста с клавиатуры. Отметим, что такой простой и эффектный инструмент для ввода с клавиатуры в других языках встречается не часто.

В Perl6 не требуется принудительно переводить строки в числа, это преобразование (если есть нужда в нём) выполняется автоматически.

Во втором ***when*** использован знак сравнения **(~~)**, используемый в основном для сопоставления с образцом. Впрочем, здесь можно применить и привычный знак ***(===)***. Эту строку также можно было бы заменить и на такую:

```
when "5" {print "$x = 5\n";}
```

И в этом случае выполнялось бы точно то же самое сравнение ***($x ~~ 5)***. Этот вариант как раз и представляет pattern matching.

Кстати, объявление функции может находиться в любом месте, компилятору это безразлично. Так, наш пример мы могли бы представить и в таком виде:

```
print "Введите число: ";
my $x = get;
f($x);
print "Нажмите любую клавишу";
my $xxx = get;
sub f($x)  {
        given $x  {
                when $x < 5 {print "$x < 5\n";}
                when $x ~~ "5" {print "$x = 5\n";}
                when $x > 5 {print "$x > 5\n";}
        }
}
```

В блоке оператора ***given*** может использоваться специальная переменная (topic variable), имеющая фиксированный идентификатор ***($_)***. Этой переменной передаётся значение аргумента оператора ***given***, в примере это ***$x***. Таким образом, переменную $_ можно использовать вместо ***$x***, например, так:

```
given $x  {
        when $_ < 5 {print "$_ < 5\n";}
        when 5 {print "$_ = 5\n";}
        when $_ > 5 {print "$_ > 5\n";}
}
```

Переменные, подобные topic variable применяются и в других языках (обычно имеют другие обозначения). Как увидим далее, в Perl6 эта переменная используется и в других конструкциях и с большей пользой, чем в нашем примере.

Во второй строке when вместо ***"5"*** я поставил ***5***, Perl6 выполняет преобразования типов автоматически и часто об этом не надо заботиться. Третья строка when выполняет избыточную проверку, поскольку вариант ***$x > 5*** здесь уже безальтернативный. Можно в этом случае применить директиву ***default***:

```
default { print "$x > 5\n";}
```

Кроме того, слово ***default*** можно вообще опустить, а при этом ещё и фигурные скобки становятся не обязательными. Наконец, есть ещё один вариант:

when * {print "$x > 5\n";}
Знак звёздочки здесь трактуется, как «любое значение».

2.3 Циклы

Оператор ***loop*** организует бесконечный цикл:

```
loop {
        say "Бесконечный цикл!";
        last;
}
```

Директива ***last*** (аналог ***brea***k в других языках) осуществляет немедленный выход из цикла. Если эту строчку закомментировать, цикл будет повторяться бесконечно с выводом заданной фразы.

В Perl6 тот же оператор ***loop*** организует цикл, аналогичный циклу ***for*** в других языках:

```
loop (my $i = 0; $i < 5; $i++) {
        print "$i, ";
}
```

Получим: ***0, 1, 2, 3, 4,***

Директива ***next*** (аналог ***continue***) принимает условие, по которому происходит прерывание текущей итерации и переход на начало новой:

```
loop (my $i = 0; $i < 5; $i++) {
        next if $i == 3;
        print "$i, ";
}
```

В этом случае результат будет ***0, 1, 2, 4,*** то-есть, число 3 не будет выведено.

В Perl6 есть и цикл ***for***, применяемый при работе с коллекциями. Для цикла ***for*** принят специальный синтаксис:

```
my @a = 'one', 'two', 'three';
for @a -> $v {
        print "\$v = $v\n";
}
```

Получим:

$v = one
$v = two
$v = three

Специальный знак стрелки ***(->)*** надо понимать так, что переменной ***$v*** в цикле последовательно передаются значения элементов списка ***@a.***

В блоке цикла ***for*** тоже доступна уже знакомая нам специальная переменная ***$_***, которой передаются значения элементов списка автоматически. При этом её использование позволяет сделать код более лаконичным и компактным:

```
for @a {
        say "\$_ = $_";
}
```

Получим такой же результат:

```
$_ = one
$_ = two
$_ = three
```

Значит, использование ***$_*** избавляет от необходимости применять знак стрелки.

Отметим попутно, что оператор вывода ***say*** (как впрочем и ***print***, а также и большинство встроенных функций) может использоваться, как метод (в точечной нотации):

```
my $x = 'Hello';
$x.say; → Hello
$x.print; → Hello
```

С использованием этого способа пример будет иметь вид:

```
for @a {
        $_.say;
}
```

Ну и наконец, при таком применении оператора ***say*** можно вообще не указывать переменную ***$_***, она будет подразумеваться по умолчанию:

```
for @a {
        .say;
}
```

Пришли к варианту, в котором сокращать уже действительно больше нечего.

Кроме ***last*** и ***next*** есть ещё директива ***redo***, позволяющая повторить текущую итерацию ещё раз с тем же значением переменной ***$_***. Применение ***redo*** увидим далее.

Есть ещё такая удобная форма цикла ***for***, позволяющая повторять вызов функции:

```
my $x = 0;
sub f {
  print "{$x++}, ";
}
f for 1 .. 6; → 0, 1, 2, 3, 4, 5,
```

Здесь число итераций задаётся рангом.

2.4 Операторы для сравнения значений

В Perl6 операторы сравнения значений для чисел и для текстовых значений различны. Оператор ***(==)*** проверяет числа на равенство, а оператор ***(!=)*** - на неравенство:

```
3 == 4; → False
3 != 4; → True
```

Для проверки текстовых значений на равенство применяется оператор ***eq***, а на неравенство - ***ne***:

```
my $x = 'perl';
my $y = 'Ruby';
$x eq $y; → False
$x ne $y; → True
```

Перед этими операторами может стоять знак ***(!)***, меняющий их смысл на противоположный:

```
$x !eq $y; → True
$x !ne $y; → False
```

Кроме этих операторов можно применять ещё ***(===), (!==), (eqv), (!eqv),*** проверяющих на каноническое (или глубокое) равенство. В частности, ***eqv*** можно использовать для коллекций:

```
my %c = (1 => 'one', 2 => 'two');
my %d = (1 => 'one', 3 => 'three');
%c eqv %d; → False
%c eqv %c; → True
```

Поскольку Perl6 может автоматически преобразовывать числа в текст и наоборот, если это допустимо по смыслу, то в большинстве случаев все операторы сравнения работают для чисел, текста и коллекций, но результат может оказаться неожиданным, и это надо иметь ввиду.

Как мы уже видели, оператор ***(~~)*** применяется в конструкции given-when. Этот же оператор применяется и во многих других

специальных случаях. Например, с помощью ***(~~)*** можно проверить наличие в hash заданного ключа:

my %x = <one один two два three три>;
say 'two' ~~ %x; → True
say 'five' ~~ %x; → False

Вот ещё один, более замысловатый пример:

sub f($x) { $x == 3; }
say 3 ~~ &f; → True

Процедура ***f*** принимает один аргумент и возвращает значение типа Bool (просто сравнивает свой аргумент с заданным значением). Выражение ***3 ~~ &f;*** проверяет, что вернёт процедура при заданном значении аргумента (слева от ~~).

say 5 ~~ &f; → False

Перед идентификатором процедуры ставится знак ***&***.

Оператор ***(~~)*** позволяет также проверять тип переменной:

my $x = 'Hello';
$x ~~ Str; → True
$x ~~ Real; → False

Наконец, оператор ***(~~)*** используется в такой важной области, как анализ текста с помощью регулярных выражений. Но это отдельная тема и пока не будем её касаться.

Операторам сравнения **<, <=, >, >=** для чисел соответствуют операторы ***lt, le, gt, ge*** – для текстовых значений.

'perl' lt 'ruby'; → True

Таким образом, Perl6 предлагает множество вариантов для сравнения значений. Кажется, что многовариантность не только в этом случае, но всюду, где это возможно, была одной из целей создателей языка. Не знаю, какая польза от такого подхода. Во всяком случае лишнюю путаницу и вероятность ошибок такой стиль несомненно увеличивает. Часто операторы (==), (===), (~~) могут применяться с одинаковым результатом, но между ними есть отличия, разбираться в их тонкостях здесь у меня нет желания.

2.5 Ранг (range)

Ещё один вид коллекций представляют ранги — последовательности целых чисел. Для создания ранга принят следующий синтаксис:

my $x = 3 .. 7;

Ранг ***$x*** представляет последовательность чисел ***3, 4, 5, 6, 7***
Метод ***head*** позволяет извлечь первый элемент ранга:
$x.head; → (3)
А метод ***tail*** – последний:
$x.tail; → (7)
Элементы ранга можно извлекать, как из списка:
$x[2..4]; → (5 6 7)

С помощью знака ***(^)*** можно исключить первый или последний элемент из ранга (или оба сразу):
my $y = 3 ^..^ 7;
Ранг ***$y*** представляет коллекцию ***4, 5, 6***:
$y.head; → (4)
$y.tail; → (6)

С помощью рангов можно создавать списки:
my @a = 3 .. 7; → [3 4 5 6 7]
В следующих примерах для наглядности будем пользоваться списками. Если левая граница ранга представлена ***0***, то такой ранг можно создать так:
my @a = ^5; → [0 1 2 3 4]

С помощью оператора ***(…)*** (три точки) можно создавать ранги с заданным шагом изменения элементов:
my @b = 3, 5 ... 9; → [3 5 7 9]
Здесь первые два числа задают значение первого элемента и шаг (у нас он равен ***2***), а третье число — конец ранга. Не обязательно третье число должно совпадать с концом ранга, принимается ближайшее меньшее:
my @b = 3, 5 ... 10; → [3 5 7 9]

С помощью знаков ***(*>)*** можно конец ранга сделать равным ближайшему большему числу:
my @c = 3, 9 ... *>30; → [3 9 15 21 27 33]
Ранги можно использовать и для индексов при извлечении элементов из списка:
@c[2 .. 4]; → (15 21 27)
Здесь извлекли элементы с индексами от ***2*** до ***4***.
@c[3 .. *] → (21 27 33)
Знак ***(*)*** вместо индекса позволяет извлечь элементы с заданного места до конца списка.

Ранги можно использовать в цикле ***for***, например:

my $x = 5;
for ^$x {print "$_, ";} → 0, 1, 2, 3, 4,
Напоминаю, что ***^$x*** - это ранг, в частности:
my @a = ^$x; → [0 1 2 3 4]
В теле цикла использована topic variable ***$_***.

2.6 Ленивые вычисления

В Perl6 реализован механизм ленивых вычислений; а именно, выражения не вычисляются до тех пор, пока их результат не будет использован для чего-либо. Посмотрим на эффект ленивых вычислений на примере бесконечных рангов и списков. В Perl6 имеется возможность использовать специальное численное значение, обозначаемое ***Inf*** и соответствующее бесконечности. Значение ***Inf*** имеет тип ***Num*** (подробнее о типе Num поговорим позже), и этим значением можно инициировать переменные:
my Num $x = Inf;
Переменная ***$x*** может быть использована в арифметических выражениях:
1 / $x; → 0
5 * Inf; → Inf
Inf / 120; → Inf
При делении любого числа на ***Inf*** получим ***0***, а умножение или деление ***Inf*** на любое число даёт ***Inf***.

С помощью ***Inf*** можно создать бесконечный ранг:
my $r = 0 .. Inf;
Если бы при такой инициации переменной ***$r*** стали вычисляться все элементы бесконечного ранга, создалась бы аварийная ситуация. Тут могли бы быть разные варианты, но скорее всего произошло бы «зависание» компьютера, поскольку целые числа не ограничены по величине. Ничего этого не происходит потому, что элементы не вычисляются до тех пор, мы не попытаемся их использовать. Например, мы можем извлечь любые элементы ранга ***$r***:
$r[20..30]; → (20 21 22 23 24 25 26 27 28 29 30)

С помощью бесконечного ранга можно создавать бесконечные списки, в отношении которых тоже действует правило ленивых вычислений:
my @a = 3 .. Inf;
@a[125]; → 128

Значит, при работе с рангами и списками никогда не вычисляется больше элементов, чем требуется. Интересно, что получится, если попробовать извлечь последний элемент бесконечного списка:
$a.tail; → Cannot tail a lazy list
Компилятор протестует.

Бесконечный ранг можно создавать и так:
my @a = 0 .. *; - то же самое, что и ***my @a = 0 .. Inf;***
@a[3]; → ***3***
Если попытаться извлечь из бесконечного списка ***@a*** элементы, начиная с некоторого индекса и до конца списка, все эти элементы станут вычисляться и компьютер зависнет:
say @a[25 .. *]; - лучше этого не делать!
Можно вместо слова Inf применять общепринятое обозначение ∞, если компьютер это позволяет:
my @a = 1..∞;

Константы в Perl6 могут представлять коллекции и даже бесконечные списки (и, значит, константы тоже ленивые):
constant c = 5, 15 ... *; → ***(...)***
say c[^5]; → ***(5 15 25 35 45)***

3. Функции и «синтаксический сахар»

Perl6 имеет множество встроенных функций и методов, реализующие разнообразные возможности, иногда довольно экзотические. Выше мы уже видели много примеров «синтаксического сахара», в Perl6 это обычная практика и использование подобных приёмов позволяет писать программы необычного вида. Посмотрим на примеры.

3.1 Распаковка (деструкция)

Название области видимости (declarator, пока у нас только ***my***) может употребляться в форме своеобразной функции. Приведём примеры.

При объявлении переменных можно использовать необязательные скобки:
my ($x); - то же, что и ***my $x;***
Здесь после ***my*** обязателен пробел.

Скобки необходимы, если объявляется сразу несколько переменных:
my ($x, $y, $z); → ((Any) (Any) (Any))
Можно задать тип переменных:
my Real ($x, $y, $z); → ((Real) (Real) (Real))
Как обычно, объявление можно совместить с инициацией:
my Real ($x, $y, $z) = 3.4, 0.75, 99;
say $y; → 0.75

При инициации можно использовать списки:
my @a = <Ruby Scala Fantom>;
my ($x, $y, $z) = @a;
say $z; → Fantom
Если некоторые переменные не нужны, их идентификаторы можно заменить знаком ***$***:
my ($, $y, $) = @a;
say $y; → Scala

Переменные могут быть любыми, например, это могут быть списки:
my (@a, @b); → ([] [])
Могут быть переменные разных сортов:
my (@a, @b, $x); → ([] [] (Any))
Можно тут же задать и типы:
my (Str @a, Bool @b, Int $x); → ([] [] (Int))

Знак ***(*)*** позволяет работать с произвольным числом переменных:
my ($x, *@a) = 1, 2, 3, 4, 5, 6; → (1 [2 3 4 5 6])
say $x; → 1
say @a; → [2 3 4 5 6]

Когда аргументом процедуры является список, элементы списка можно сделать именованными с тем, чтобы в теле функции их можно было использовать по имени (а не с помощью вызова по индексу). При этом применяется следующий синтаксис:

```
my @s = 5, 9;
sub f(@a [$x, $y]) {
  say "\$x = $x, \$y = $y, @a = {@a}.";
}
f(@s); → $x = 5, $y = 9, @a = 5 9.
```

Функция ***f*** принимает список ***@a***, а в квадратных скобках перечисляются элементы этого списка с присваиванием им

идентификаторов ***$x*** и ***$y*** (напоминаю, что обратный слэш подавляет интерполяцию, а также вместо ***{@a}*** можно было бы написать ***@a[]***).

Если сам список ***@a*** не используется в теле процедуры, его идентификатор можно не указывать, то-есть применить анонимный список (вместо идентификатора один sigil **@**):

```
my @s = 3, 4;
sub f(@ [$x, $y]) {
  $x ** 2 + $y ** 2;
}
say "{@s[0]}**2 + {@s[1]}**2 = {f(@s)}"; → 3**2 + 4**2 = 25
```

С помощью анонимного списка можно задавать и произвольное количество аргументов (как мы уже делали ранее для именованных списков):

```
my @s = 3, 4, 5, 6, 7;
sub f(@ [$x, *@y]) {
  say "$x, @y[]"
}
f(@s); → 3, 4 5 6 7
```

Здесь анонимный список имеет два аргумента: простая переменная ***$x*** и список произвольной длины ***@y***. При этом все данные вводятся одним списком.

Подобным же образом можно работать и с hash. Например, можно также применять анонимные hash:

```
sub f(% (:Миша($age1), :Катя($age2))) {
  say "Миша $age1 лет, Катя $age2 лет";
}
f({ Катя => 17, Миша => 20}); → Миша 20 лет, Катя 17 лет
```

В этом случае ключи в теле процедуры не используются, а значения доступны, как простые переменные. При вызове процедуры эти переменные вводятся, как именованные и, в частности, могут вводиться в произвольном порядке. Можно также hash сформировать заранее:

```
my %h = Катя => 17, Миша => 20;
```

f(%h); - получим то же самое.

Цикл for в Perl6 не возвращает последнего вычисленного значения (возвращает Nil), но, как и для условных выражений if, можно с помощью директивы ***do*** заставить цикл возвращать результат:

```
sub f($n)  {
```

```
        do for ^$n -> $x {
            $x**3;
        }
}
my @a = f(5);
say @a; → [0 1 8 27 64]
```

Значит, цикл ***do for*** возвращает последнее вычисленное значение на каждой итерации.

3.2 Анонимные функции (λ-функции)

Ранее мы уже познакомились с анонимными функциями, создаваемыми с использованием ключевого слова sub. В Perl6 можно создавать и применять анонимные функции многими способами. Такие функции играют важную роль в функциональном программировании, где их обычно называют λ-функциями. В некоторых ситуациях λ-функции также называют closure и для краткости в дальнейшем будем чаще использовать этот термин, в конце-концов вовсе не в терминах дело. Рассмотрим здесь эти closure более детально.

Простейшим представителем анонимной функции является блок. В Perl6 блоком можно инициировать простую переменную, которая затем может быть использована, как функция или метод:

```
my $x = 5;
my $g = { $x ** 3 }
```

В таком варианте блок может содержать только предварительно объявленную глобальную переменную и нет возможности ввести локальную переменную, как аргумент функции ***$g***. Чтобы выполнить ***$g***, надо использовать круглые скобки, как для функции без аргументов:

```
$g(); → 125
```

Впрочем, аргумент можно и указать, но он не будет использован:

```
$g(7); → 125
```

$g можно использовать и как метод, в точечной нотации, при этом могут быть разные варианты с одинаковым результатом:

```
().$g; → 125
.$g; → 125
7.$g; → 125
```

Дело в том, что как и во всех других вариантах использования блока, в блоке доступна topic variable ***($_)*** и при указании аргумента он передаётся этой переменной. Ясно, что ***$_*** можно использовать в блоке:

```
my $x = 5;
my $g = { $x * $_ }
say $g(3) → 15
```

Как видим, всё работает при простой переменной ***$g***, то-есть, с применением знака ***$***. Но выше мы уже встречали, что для функций есть и свой sigil - ***(&);*** замена ***$*** на ***&*** в данных примерах ничего не меняет. Условимся в дальнейшем применять всегда знак ***(&)***.

Для того, чтобы ввести локальную переменную в блоке, необходимо применять вариант со стрелкой ***(->)***, или так называемый "pointy block":

```
my &g = -> $x { sqrt($x) }
```

Теперь ***&g*** можно применять, как функцию с аргументом:

```
&g(5); → 2.2360
```

Или, как метод:

```
5.&g; → 2.2360
```

Или, если есть дробная часть числа:

```
6.37.&g; → 2.5238
```

Тут оказались две точки, но компилятор это понимает правильно.

Можно и не вводить имя для функции ***(&g)***, а использовать "pointy block" непосредственно, передавая ему аргумент:

```
-> $x { sqrt($x) }(5); → 2.2360
```

Аргументов у pointy block может быть больше одного:

```
my &g = -> $t, $s { $t + $s}
&g(2, 3); → 5
```

Для применения ***&g*** с несколькими аргументами в точечной нотации придётся использовать список:

```
my &g = -> [$t, $s] { $t + $s }
[2, 3].&g; → 5
```

Интересно, что вариант вызова ***&g(2, 3)***; в данном случае не работает. Как обычно, вместо квадратных скобок могут быть круглые.

Если аргумент один можно воспользоваться topic variable ***($_)***:

```
my &g = { $_ ** 3}
&g(3); → 27
5.&g; → 125
```

Можно, конечно, поступать и так:

my &g = -> $t { $t + $_}

Но теперь переменную ***$_*** надо инициировать заранее:

$_ = 7;
&g(3); → 10
3.&g; → 10

Над closure можно выполнять разнообразные манипуляции. Например, они могут быть элементами списка:

```
my &g1 = { $_ ** 2 }
my &g2 = { $_ ** 3 }
my &g3 = { sqrt($_) }
my @a = (&g1, &g2, &g3);
sub f($x) {
        do for @a -> &g {
                &g($x);
        }
}
my @b = f(5);
say @b; → [25 125 2.2360]
```

Здесь в цикле ***do for*** переменной ***&g*** последовательно передаются closure ***&g1, &g2, &g3***, которые и выполняются с аргументом ***$x***, получаемом процедурой ***f*** при её вызове. В результате формируется список результатов. Интересно, а можно ли вместо ***&g*** воспользоваться переменной ***$_***?

```
my &g1 = { $_ ** 2 }
my &g2 = { $_ ** 3 }
my &g3 = { sqrt($_) }
my @a = (&g1, &g2, &g3);
sub f($x) {
        do for @a {
                $_($x);
        }
}
my @b = f(9);
say @b; → [81 729 3]
```

Всё прекрасно работает!

Пожалуй, самый интересный и мощный механизм — это способность одних функций принимать в качестве аргументов и

возвращать в качестве результатов другие функции. Такой способностью обладает большинство современных языков программирования, но в Perl6 это делается наиболее просто и элегантно. Рассмотрим простейший пример:

```
my &g = { $_ ** 2 }
sub f(&r, $x)  {
        say &r($x);
}
f(&g, 7); → 49
```

Здесь функция ***f*** принимает два аргумента: функцию ***&r*** и переменную ***$x***. В теле функции ***f*** выполняется принятая функция-аргумент. Можно, конечно, не вводить имя ***&g***, а передать функции ***f*** непосредственно closure:

```
sub f(&r, $x)  {
        say &r($x);
}
f({ $_ ** 2 }, 7); → 49
```

Теперь пример возвращения функции, как результата:

```
sub f($x)  {
        my &g = -> $t { do if ($x>0) {$t ** 2} else {$t ** 3} }
        &g;
}
say f(-3)(2); → 8
say f(3)(5); → 25
```

Выражение ***f(-3)(2);*** надо понимать так, что сначала выполняется функция ***f*** с аргументом ***-3***, которая возвращает результат в виде функции ***&g***, которая затем и выполняется с аргументом ***2***.

3.3 Захватывания (capture)

Функции в качестве аргументов для других функций применяются в технике, называемой «захватывание». Список аргументов функции превращается в захватывание с помощью знака ***(\)***. Функции можно передавать несколько таких захватываний, например, в цикле, получая разные результаты. Лучше всего разобраться со всем этим на конкретном примере:

```
sub f($x, $y, :&g) {
        &g($x, $y);
}
```

```
my @z = \(9, 3, g => { say $^a + $^b }),
        \(6, 7, g => { say $^a * $^b });
for @z -> $t { f(|$t); } → 12 42
```

Головная функция ***f*** принимает два позиционных аргумента: ***$x*** и ***$y*** и один именованный - :***&g***. Sigil ***&*** указывает, что этот аргумент сам является функцией (впрочем, здесь всё будет работать и при использовании sigil ***$*** вместо ***&***). В теле функции ***f*** аргумент ***&g*** вызывается, как функция с аргументами ***$x*** и ***$y***.

Список ***@z*** содержит два элемента, которые и представляют захватывания, представляющие, в свою очередь, список аргументов функции ***f*** со знаком ***(\)*** впереди. В цикле for функция f вызывается циклически и на каждой итерации ей передаются захватывания из списка @z. Для упаковки списка в один элемент использован знак ***(|)***. Ранее мы уже применяли этот знак для распаковки списков и hash.

Захватывания можно применять и для других целей, например, в том случае, когда головная функция передаёт свои аргументы нескольким другим функциям. Давайте посмотрим на примере:

```
my $z = \(x => 3, y => 4);
sub g1(:$x, :$y) { $x + $y; }
sub g2(:$x, :$y) { $x * $y; }
sub f($t) {
    g1(|$t) + g2(|$t);
}
say f($z); → 19
```

Сначала переменную ***$z*** мы инициировали захватыванием, содержащим два именованных аргумента $x и $y. Затем объявили две функции ***g1*** и ***g2***, имеющих в своём списке аргументов те же две именованных переменных. Наконец, мы объявили головную функцию ***f*** с одним аргументом ***$t***, представляющим захватывание, а в теле этой функции вызываются функции ***g1*** и ***g2***, которым передаётся аргумент - захватывание функции ***f***. Для распаковки захватывания опять использован знак ***(|)***. Конечно, всё это можно реализовать и другими способами, но мне кажется, что применение таких приёмов привносит в код упорядоченность и ясность.

Perl6 имеет много и других подобных трюков. Вряд ли целесообразно рассматривать всех их здесь, а, как советуют опытные программисты, лучше осваивать постепенно по мере необходимости в процессе реального программирования.

3.4 Итератор map

Без итераторов не обходится ни один современный язык и Perl6 не является исключением. Итератор ***map*** принимает два параметра: closure (или именованную функцию) и коллекцию, а затем применяет эту функцию к каждому элементу коллекции и результат возвращает в виде списка. Рассмотрим пример:

```
my &g = -> $x { exp($x) }
my @a = (1, 2, 3);
my @b = map(&g, @a);
say @b; → [2.7182 7.3890 20.0855]
```

Здесь функция ***&g*** принимает элементы списка ***@a*** в качестве аргументов, а результатами мы инициировали список ***@b***. Можно не применять имя для функции, а передавать итератору closure:

```
my @b = map(-> $x {exp($x)}, @a);
```

Результат будет тот же.

Можно, конечно, и так:

```
my @b = map({exp($_)}, @a);
```

Вместо списка ***@a*** могут быть коллекции других видов. Тот же результат получим, применив ранг:

```
my &g = -> $x { exp($x) }
my @b = map(&g, 1 .. 3);
say @b; → [2.7182 7.3890 20.0855]
```

Итератор ***map*** работает и с hash:

```
my &g = -> $x { $x.value ** 2 }
my %h = ('a'=>1, 'b'=>2, 'c'=>3);
my @b = map(&g, %h);
say @b; → [1 9 4]
```

- (порядок в hash не фиксируется)

Здесь функция ***&g*** получает пары из ***%h*** а метод ***value*** извлекает из пары значение. Метод ***key*** извлекает ключи:

```
my &g = -> $x { $x.key }
my %h = ('a'=>1, 'b'=>2, 'c'=>3);
my @b = map(&g, %h);
say @b; → [a c b]
```

Итератор ***map*** может использовать и анонимные функции, созданные с помощью ключевого слова ***sub***:

```
my @a = (7, 4, 6, 3, 1, 8, 9);
my @b = map(sub ($x) {do if $x%2 {$x} }, @a);
```

say @b; → [7 3 1 9]

Здесь оператор ***do if*** возвращает только те элементы списка, для которых остаток от деления на ***2*** не равен нулю, так как такое значение трактуется, как ***True***. Значит, эта программа отфильтровывает нечётные числа из списка ***@a***. Для того, чтобы получить чётные числа, достаточно перед выражением ***$x%2*** поставить знак отрицания ***(!)***:

```
my @a = (7, 4, 6, 3, 1, 8, 9);
my @b = map(sub ($x) {do if !($x%2) {$x} }, @a);
say @b; → [4 6 8]
```

Естественно, что анонимной функции можно предварительно присвоить имя:

```
my &g = sub ($x) { do if ($x%2) {$x} }
my @a = (7, 4, 6, 3, 1, 8, 9);
my @b = map(&g, @a);
say @b; → [7 3 1 9]
```

Однако, нельзя в этой ситуации применить именованную функцию ***f***:

```
sub f ($x) { do if ($x%2) {$x} }
```

Кстати, Perl6 имеет операцию ***(%%)***, возвращающую ***True***, если числа делятся на цело, и ***False*** – если с остатком. Таким образом, для фильтрации чётных чисел можно было также написать:

```
my &g = sub ($x) { do if ($x %% 2) {$x} }
```

Приведём ещё пример подсчёта, сколько раз в списке встречается элемент с заданным значением:

```
my @a = (7, 4, 6, 3, 6, 8, 6);
sub f(@s, $x) {
        my $i = 0;
        map({ do if $_ == $x {$i++} }, @s);
        say $i
}
f(@a, 6); → 3
```

Вместо переменной ***$_*** есть ещё один способ передачи значения в блок, в котором применяется знак звёздочки ***(*)***:

```
my @a = (7, 4, 6, 3, 9);
my @b = map( * + 3, @a); → [10 7 9 6 12]
```

Это тоже самое, что и

```
my @b = map( {$_ + 3}, @a ); → [10 7 9 6 12]
```

В этом примере к каждому элементу списка прибавляется ***3***. Можно выполнять и другие арифметические операции, например умножение:
my @b = map(* * 3, @a); → [21 12 18 9 27]
Операцию со звёздочкой можно повторять:
my @b = map(*+3 *+3, @a); → [16 13 15 12 18]

С помощью этого приёма можно создавать своеобразные функции:

```
my $x = 5;
say (*+3/2)($x); → 6.5
```

Значит, если где-то в выражении (скобки круглые) поставить звёздочку, то она может рассматриваться, как аргумент, который затем выражению можно передать, как функции. Аргументов может быть несколько, достаточно поставить несколько звёздочек:

```
my $y = 2;
say ((*+3)/*)($x, $y); → 4
```

Здесь первая звёздочка замещается на аргумент ***$x***, а вторая — на ***$y***.

Созданной таким способом функции можно присвоить имя:

```
my &g = ((* - 2)/(* * 3));
say &g($x, $y); → 0.5
```

Есть ещё одна краткая форма для closure. Рассмотрим на примере. Сначала применим closure в обычном виде:

```
my &g = -> $x, $y { $x + $y }
&g(3, 5); → 8
```

Для того, чтобы не писать обозначения переменных ***$x, $y*** перед блоком, можно применить такой синтаксис:

```
my &g = { $^x + $^y }
&g(4, 7); → 11
```

Этот же приём работает и с анонимной функцией, созданной с помощью ключевого слова sub:

```
my &f = sub ($x, $y) { $x **2 + $y **2 }
&f(3, 4); → 25
```

Это можно заменить на:

```
my &f = sub { $^x **2 + $^y **2 }
&f(3, 4); → 25
```

Конечно, такие closure можно применять для итератора map:

```
my @a = 1 .. 4;
my @b = map({ $^x ** 2}, @a); → [1 4 9 16]
```

Иногда в closure для итератора ***ma***р может быть несколько переменных. Посмотрим, как это работает:
my @c = map({$^x + $^y + 3}, @a); → [6 10]
На первой итерации переменная ***$x*** получает значение первого элемента списка ***@a***, то-есть ***1***, а переменная ***$y*** – значение второго элемента, то-есть ***2***. Соответственно, на второй итерации ***$x*** равно ***3***, а ***$y*** равно ***4***. Таким образом, в результирующем списке всего два элемента. Кстати, если в списке ***@a*** будет нечётное число элементов, компилятор зафиксирует ошибку, поскольку для работы closure не хватит значений. Если в closure будет три переменных, число элементов в списке должно быть кратно 3, и так далее. Вообще, выражение:
{ $^b / $^a }
эквивалентно такому closure:
-> $a, $b { $b / $a }

3.5 All, any

Если перед коллекцией поставить слово (модификатор) ***all***, получим структуру, автоматически организующую циклы. Посмотрим на примерах, как её можно применять.
my $x = all(5, 2, 8, 4, 3, 9);
Каков же тип переменной ***$x***?
$***x.WHAT; → (Junction)*** - (переводится «соединение»)
Применим ***$x*** в каком-нибудь цикле, например:
my $y = 5 > $x; → all(False, True, False, True, True, False)
На каждой итерации элемент списка проверяется на условие (***< 5***).
Как видим, результат ***$y*** тоже имеет тип ***Junction***.

Таким образом, модификатор all позволяет автоматически создавать циклы, в которых над элементами списка можно выполнять нужные действия. Фактически all тоже является итератором с предельно простым синтаксисом. Ещё один пример:
my $y = $x ** 2;
say $y; → all(25, 4, 64, 16, 9, 81)

Чтобы иметь возможность извлечь элементы из переменной ***$y***, имеющей тип Junction, надо превратить её в список. Для этого можно воспользоваться тем, что такие структуры применимы в цикле ***for***:
my @a = [];
for $y { @a.push($_); }

say @a; → [25 4 64 16 9 81]
Здесь в цикле for к пустому списку добавляются элементы из структуры $y с помощью метода push. Можно было бы всё сделать сразу и не вводить переменную $y:
my @a = [];
for $x**2 { @a.push($_); }
say @a; → [25 4 64 16 9 81]

В примере всего лишь элементы возводятся в квадрат, но можно использовать и более сложные выражения. Например, можно организовывать вложенные циклы:
my $x = all(1, 2, 3);
my $y = $x + $x; → all(all(2, 3, 4), all(3, 4, 5), all(4, 5, 6))
Нетрудно разобраться, что тут вычисляется.

Модификатор all можно с таким же успехом применять и к коллекциям других видов, например, к hash. Примеры нетрудно придумать. Существуют и другие виды подобных модификаторов, организующих Junction, например модификатор ***any***. Посмотрим на такой пример:
my @a =3,2,7,3,4,3;
my $b = do if $x == any @a -> $i { $i; }
say $b; → any(True, False, False, True, False, True)

Отметим ещё, что следующие два определения эквивалентны:
$x = any(1, 2, 3);
$x = 1 | 2 | 3;

3.6 Оператор for в роли итератора

Итератор можно создать с помощью цикла for, используя такой синтаксис:
.m() for @a
Здесь ***m*** – некоторый метод, а ***@a*** – заданный список. В такой форме оператор for вызывает метод m для каждого элемента списка @a. Посмотрим на конкретный пример:
my @a = 1, 2, 3, 4, 5;
my @b = [];
my &f = { $_ ** 2; }
push @b, .&f() for @a;
say @b; → [1 4 9 16 25]

В роли метода использована функция ***&f***, созданная с помощью closure. С помощью метода ***push*** мы заталкиваем результат очередной итерации в пустой первоначально список ***@b***.

Оператор for можно совместить с оператором вывода print или say:

my @a = <москва ленинград киев донецк>;
.say for @a; →
москва
ленинград
киев
донецк

Сделаем первые буквы заглавными:

.say for map {.tc}, @a;

Сначала применяем итератор ***map***, который с помощью встроенного метода ***tc*** делает первые буквы элементов массива ***@a*** заглавными. Напоминаю, что блок ***{.tc}*** эквивалент ***{$_.tc}***; в таком варианте переменная ***$_*** подразумевается по умолчанию.

Perl6 имеет и другие итераторы, например:

my @a = 1,2,3,4,5; → 15
my $s = reduce -> $a, $b { $a + $b }, @a; → 15

Итератор ***reduce*** принимает closure и список. В closure указывается, какие действия надо выполнить над элементами списка. Кстати, для краткости можно использовать так называемые placeholder при следующем синтаксисе:

my $s = reduce { $^a + $^b }, @a; → 15

Эти placeholder ($^a и $^b) трактуются, как элементы списка, порядок их следования определяется порядком букв (a и b) в алфавите.

my $s = reduce { $^a * $^b }, @a; → 120

Итератор ***grep*** позволяет фильтровать элементы списка по заданному условию:

my @evens = grep { $_ %% 2 }, @a; → [2 4]

На примере итераторов видим, что в Perl6 любое действие можно выполнить многими способами, иногда даже кажется, что количество таких способов прямо-таки бесконечно.

3.7 Каррирование функций (уменьшение их арности)

Каррирование — это когда, например, функция двух переменных (её арность равна 2) трансформируется в функцию одного переменного (с арностью, равной 1). По смыслу приём каррирования

очень прост: значение одного переменного (а может быть и нескольких для функций многих переменных) фиксируется, то-есть этому переменному присваивается значение. В Perl6 каррирование выполняется легко и естественным образом. Покажем на примере:

sub f($x, $y) { $x * $y }

Здесь ***f*** – функция двух переменных.

f(2, 3); → 6

Присвоим оному аргументу конкретное значение, например, пусть переменная ***$y*** будет равна ***5***. Для получившейся при этом функции одного переменного примем идентификатор ***g***. Тогда каррирование можно выполнить, например, так:

s***ub g($x) { f($x, 5) }***

g(3); → 15

Вместо ***$x*** можно, конечно, использовать любой другой идентификатор:

sub g($a) { f($a, 5) }

g 3; → 15 - скобки, как обычно, можно опускать.

Каррирование можно выполнить и другими способами. Можно, в частности, использовать для этого closure:

my &g = -> $t { f($t, 5) }

g 4; → 20

Каррировать можно встроенные функции и методы; в частности, арифметические операции:

sub f($x) { $x + 7 }

f 2; → 9

Эта функция ***f*** всегда будет прибавлять число ***7*** к своему аргументу.

Применим closure:

my &f = { $_ +7 }

f 4; → 11

Тут мы использовали topic variable, что всегда делает код компактнее.

Ещё проще в такой ситуации использовать знак ***(*)*** в значении «всякий», «любой» (обычно называют placeeholder):

my $x = * / 2;

На самом деле ***$x*** в этом случае можно использовать, как функцию одного переменного:

$x(4); → 2

$x(7); → 3.5

В такой ситуации аргумент должен быть в круглых скобках, указывающих, что переменная используется в роли функции. Но можно опустить и sigil и скобки:

x 9; → 4.5

Можно, конечно, использовать sigil &, что в данном случае ничего не меняет:

my &x = */2;

Для каррирования функций есть также специальный метод ***assuming***. Покажем на примере:

```
sub f($x, $y) { $x ** $y; }
my &g = &f.assuming(2);
say g(3); → 8
```

К функци двух переменных f применяется метод assuming, который принимает один аргумент, понижая арность функции. Из результата (8) видно, что метод assuming фиксирует первый аргумент функции. Метод assuming может принимать аргументы в количестве больше одного, например:

```
sub f($x, $y, $z) { $x ** $y + $z; }
my &g = &f.assuming(2,3);
say g(4);
```

Фиксируемые аргументы можно сделать именованными:

```
sub f($x, $y, :$z) { $x ** $y + $z; }
my &g = &f.assuming(z => 5);
say g(2, 3); → 13
```

Каррировать можно и встроенные функции. Например функция ***substr*** принимает три аргумента: заданную строку текста, номер начальной позиции для извлекаемой подстроки и количество символов в подстроке. С помощью метода assuming фиксируем первые два аргумента. Теперь изменяя третий аргумент можем извлекать разные подстроки:

```
my $s = "Жили-были старик со старухой";
my &f = &substr.assuming($s, 0);
say &f($_) for 9, 16, 19, 28; →
    Жили-были
    Жили-были старик
    Жили-были старик со
   Жили-были старик со старухой
```

Незамысловатый приём каррирования полезен и часто применяется в функциональном программировании.

4. О типах

Perl6 имеет динамическую типизацию — тип переменных выводится из контекста на этапе runtime. Выше уже указывалось, что тип переменных можно также задавать принудительно и тогда тип контролируется на этапе компиляции, как в языках со статической типизацией. Кроме базовых типов, с которыми мы имели дело до сих пор, Perl6 позволяет создавать свои собственные. Основные средства для создания типов class и role, мы с этими понятиями познакомимся позже, когда будем рассматривать объектно-ориентированное программирование. Есть также и другие средства создания и модификации типов, о которых поговорим сейчас.

4.1 Подтипы

Оператор ***subset*** (с двумя дополнительными ключевыми словами: ***of*** и ***where***) позволяет для любого типа создавать подтипы (subtype), применяемые для дополнительных проверок. Например, для типа целых чисел ***Int*** можно создать подтип чётных чисел, который можно назвать, скажем, ***Even***:

subset Even of Int where * %% 2;

Знак звёздочки здесь означает любое значение, а операция ***(%%)*** возвращает True, если деление без остатка; мы с этим синтаксисом уже знакомы.

Инициируем переменную чётным числом:

my $x = 6;

Теперь проверим с помощью операции ***(~~)*** тип переменной ***$x***:

$x ~~ Even; → True

Да, переменная ***$x*** имеет тип ***Even***. А относится ли она к типу ***Int***?

$x ~~ Int; → True

Да, относится, поскольку тип ***Even*** является подтипом ***Int***. Возьмём теперь нечётное число:

my $y = 7;

Проверим:

$y ~~ Even; → False

Переменная ***$y*** не относится к подтипу ***Even***.

Названия подтипов могут быть любыми, в том числе и на кириллице:

subset Чёт of Int where * %% 2;

Указывать подтип можно, например, для аргументов функций, как обычно:

```
sub f(Even $x)  {
      say "Чётное число = $x";
}
```

f(4); → Чётное число = 4

Нечётное число не может быть аргументом функции ***f***:

f(5); → Constraint type check failed for parameter '$x'

Компилятор выдаёт сообщение об ошибке.

Метод ***WHAT*** (почему-то его имя пишется заглавными буквами) позволяет узнать тип объекта:

```
my $x = 'Ruby';
$x.WHAT; → (Str)
```

4.2 Multi – функции

Как мы уже знаем, тип аргумента функции можно не указывать и тогда он считается неопределённым ***(Any)*** и функция может принимать значение любого типа. Можно тип указать, и тогда функция может принимать только значение этого типа:

```
sub f(Str $a) { say "String: $a"; }
f('Hello'); → String: Hello
f(2.5);  - ошибка
```

Иногда бывает необходимо, чтобы функция могла принимать значения разных (двух, трёх и больше) типов или разное количество аргументов. Такие функции можно создавать с помощью ключевого слова ***multi***. Посмотрим на конкретный пример:

```
multi f(Real $x)  {
      my $y = $x ** 2;
      say "Результат число: $y";
}
multi f(Str $x)  {
      if $x ~~ / <[aeiou]> / {
            say "Слово '$x' содержит гласные";
      } else { say "Гласных в слове '$x' нет"; }
```

```
}
multi f(Int $x, Int $y)  {
        say "Сумма $x + $y равна {$x + $y}";
}
f(5); → Результат число: 25
f('Hello'); → Слово 'Hello' содержит гласные
f(3, 9); → Сумма 3 + 9 равна 12
f('CSV'); → Гласных в слове 'CSV' нет
```

С ключевым словом ***multi*** можно функцию с одним и тем же именем создавать несколько раз, изменяя тип и количество аргументов. Прелесть такой техники в том, что передавая этой функции разные аргументы мы фактически выполняем разные функции с разными результатами. Выбор нужной функции происходит автоматически в результате анализа числа и типа аргументов. В этом примере во второй multi — функции ***f*** использован анализ текста с помощью регулярного выражения, проверяющего, присутствуют ли в тексте гласные буквы. О регулярных выражениях надо говорить отдельно.

Multi-функции могут также принимать аргументы с дополнительными условиями. Для условия применяется ключевое слово ***where***:

```
multi g(Int $x where * %% 2)  {
        say $x/2;
}
g(8); → 4
g(7);   - ошибка
```

Согласно условию функция ***g*** может принимать в качестве аргумента только целые чётные числа. Если попытаться передать такой функции аргумент, не соответствующий заданному условию, генерируется ошибка. Условия могут быть самыми разными. Ещё пример:

```
multi g(Int $x where 10 .. 50)  {
        say sqrt($x);
}
g(21); → 4.5825
```

В этом примере передаваемое значение должно попадать в диапазон чисел ***10 .. 50***. Поскольку имя функции одно и тоже, то нужный вариант будет выбираться по анализу аргумента. Что получится, если

задать число, удовлетворяющее обоим условиям то-есть чётное и из нужного интервала:

g(30); → 15

В такой ситуации будет выполнена функция, объявленная первой, в чём можно убедиться, переставив функции местами. Число ***21*** попадает в интервал, но не является чётным, а потому здесь неопределённости нет.

g(5); - ошибка, число не соответствует ни одному из двух условий.

Если добавить ещё третью функцию, не имеющую условия для аргумента:

```
multi g($x)  {
      say "Число не соответствует ни одному условию ";
}
```

то она будет выполняться, когда не удовлетворяются оба условия и, значит, ошибка генерироваться не будет.

Иногда в условии можно вообще не использовать имя аргумента, например, при проверке типа:

```
subset Even of Int where * %% 2;
multi f(Even) {"Число чётное";}
multi f($) {"Число нечётное";}
say f(6); → Число чётное
say f(7); → Число нечётное
```

В этом примере вместо аргумента указан только проверяемый тип числа, тем не менее, функция принимает число и если оно чётное, выдаёт результат (в теле функции невозможно использовать само число, поскольку нет его имени). В альтернативном варианте вместо аргумента можно поставить только знак ***$***, который здесь будет трактоваться, как любое число.

5. Объектно-ориентированное программирование

5.1 Классы

В основном, классы в Perl6 по своей структуре соответствуют классам в других языках, например, в Ruby. Существенное отличие имеется только в синтаксисе. Как обычно, в общем случае класс содержит поля (в документации Perl6 называют атрибутами, но не в названии суть) и методы. Поля (переменные) объявляются с обязательным служебным словом (declarator) ***has,*** которое в классе

заменяет привычное уже нам слово ***my***. Таким образом, мы встретились с ещё одним сортом этих declarator, дальше увидим и другие. Только при использовании ***has*** поля класса имеют свойства, указанные далее. Вместо ***has*** в классе на тех же правах можно применить и ***my***, и часто получим тот же результат, но иногда свойства переменных с разными declarator будут разными. Разбираться со всеми нюансами этих различий — скучное занятие. Методы в классе объявляются со служебным словом ***method***.

Есть три разновидности переменных (если использовано слово ***has***). Во-первых это переменные, которые по принятой в других языках терминологии можно назвать переменными экземпляра. Эти переменные обозначаются, как обычно, только между sigil $ и идентификатором ставится точка ***($.x)***, и доступны только по чтению (они immutable). Во-вторых, могут применяться переменные, которые доступны, как по чтению, так и по записи (mutable). Обозначаются они также, но с добавлением служебных слов ***is rw*** (сокращение от read – write). И в третьих, могут быть переменные, аналогичные переменным класса, которые можно инициировать и изменять только в теле самого класса. Для этих переменных sigil $ отделяется от идентификатора восклицательным знаком ***($!x)***. Можно ещё считать, что переменные второго вида имеют модификацию public, а третьего — private. (Эти точки и восклицательные знаки называют ещё вторыми sigil, или twigil).

Теперь можно рассмотреть конкретный пример класса:

```
class Proba  {
        has $.x;
        has Int $.y is rw;
        has Int $!z = 7;
        method f($k)  {
                $.y -= 3;
                $!z -= 2;
                $.x + $.y + $!z + $k;
                }
}
my $p = Proba.new(x => 5);
$p.y = 10;
say $p.f(3);  → 20
```

Класс ***Proba*** имеет три поля всех трёх разновидностей, описанных выше, и один метод. Как видим, в методе ***f*** поля ***$.y*** и ***$!z*** доступны по чтению и по записи. Экземпляр класса (объект) ***$p*** создаётся с помощью встроенного метода ***new***, обычно его называют конструктором, принимающего аргументы, которые являются именованными и передаются в форме пары с именем без sigil ***(x => 5)***. При необходимости можно создать в классе свой собственный конструктор, назвав его ***new*** (тогда встроенный метод ***new*** будет переопределён), или использовав какое-нибудь другое имя. Пример на эту тему рассмотрим позже.

Ранее уже упоминалось, что класс создаёт новый тип данных. Посмотрим, какой тип имеет переменная ***$p***:

say $p.WHAT; → (Proba)

Значит, экземпляры класса имеют тип, совпадающий с именем класса.

Переменная вида ***is rw*** (у нас ***$.y***) доступна по записи и по чтению при этом указывать sigil не требуется: ***($p.y)***.

Попробуем создать ещё один объект ***$p1***. Применим краткую форму для вызова метода ***new***, которая в данной ситуации будет иметь такой вид:

my Proba $p1 .= new(x => 7); (хотя, что тут краткое?)

$p1.y = 2;

say $p1.f(4) → 15

Результат равен ***15***, из чего следует, что приватная переменная ***$!z*** по-прежнему имела значение, равное **7,** а переменная ***$.y*** должна инициироваться снова, значение, полученное ею в предыдущем объекте не сохраняется.

Имеется возможность заставить конструктор выдавать указание о необходимости инициации поля, если это не было сделано. Для этой цели применяется директива ***is required***. Посмотрим на простейшем примере:

class A { has $.a is required };

В классе ***A*** имеется единственное поле (атрибут) ***$.a***. Добавка директивы is required позволяет конструктору выдавать нужное предупреждение:

A.new; → The attribute '$!a' is required, but you did not provide a value for it.

Конструктор говорит, что атрибуту ***$!a*** требуется задать значение.

A.new(a=>3); - в этом варианте будет всё в порядке.

Здесь можно ещё добавить и свой собственный текст предупреждения:

class B { has $.a is required("Необходимо задать значение для a") };
B.new; → The attribute '$!a' is required because Необходимо задать значение для a, but you did not provide a value for it.

Класс, как обычно, кроме полей и методов может содержать в своём теле произвольный код: объявления переменных, выражения, функции и их вызовы и так далее. В других языках ООП (во всех без исключения) этот свободный код в классе выполняется только в момент создания экземпляра класса (объекта). Сколько объектов будет создано, столько раз и будет выполнен этот внутренний код. Но всё не так происходит в Perl6. Здесь код в теле класса выполняется сразу при запуске программы, как если бы он был за пределами класса. И наоборот, при создании объекта внутренний код не выполняется. Посмотрим, как это выглядит на примере:

```
my $x = 3;
class A  {
        method f($y) {
                say $x * $y;
        }
        my $y = 5;
        sub f() {
                say $x * $y;
        }
        f(); → 15
        say 'Hello';          → Hello
}
my $p = A.new;
$p.f(7); → 21
```

В теле класса ***A*** есть метод и функция. Я применил для них одно и то же имя ***f***, что не имеет значения, поскольку это разные вещи. Функцию ***f*** можно вызвать только внутри класса, снаружи она не видна. Метод ***f*** можно выполнить только после создания объекта, которому этот метод и будет принадлежать. В теле класса есть также оператор вывода слова ***Hello*** и это слово будет выведено сразу при запуске программы. При создании объекта с помощью ***new*** ни функция ***f***, ни оператор вывода не выполняются. Отметим ещё, что внешняя переменная ***$x*** доступна внутри класса.

5.2 Наследование (inheritance)

Наследование в Perl6 программируется с использованием служебного слова ***is***. В общем, всё также, как и в других языках, но, как обычно, с некоторыми особенностями. Разберёмся на примере:

```
class A {
  has $.x;
  submethod ni {
    say $!x ** 2;
  }
  method f { $.x * 3 }
  method bar { $.x * 5 }
}
class B is A {
  method foo {
    say $.x;
  }
  method bar { $.x * 10 }
}
my $p = B.new(x => 5);
say $p.f; → 15
$p.foo; → 5
say $p.bar; → 50
my $p1 = A.new(x => 6);
$p1.ni; → 36
```

Объявление класса ***B*** в таком виде: ***class B is A { … }*** означает, что класс ***B*** является дочерним, а класс ***A*** родительским (класс ***B*** наследует классу ***A***). В дочернем классе доступны все поля и методы родительского класса. Действительно, в экземпляре ***$p*** класса ***B*** доступен метод ***f*** класса ***A***: ***say $p.f; → 15***. Метод ***foo*** класса ***B*** выводит на терминал значение переменной (поля) ***$.x*** класса ***A***, и, значит, поле ***$.x*** класса ***A*** доступно в классе ***B***. Далее, в классе ***B*** объявлен метод ***bar*** при том, что в родительском классе ***A*** тоже есть метод с тем же именем ***bar***. Никакого конфликта нет, метод ***bar*** класса ***B*** маскирует (переопределяет, перегружает) метод ***bar*** из родительского класса. Ну, и наконец, в классе ***A*** есть метод ***ni***, объявленный со служебным словом ***submethod***. Методы, объявленные с этим служебным словом не наследуются, и, значит,

метод ***ni*** не доступен в объекте ***$p*** дочернего класса ***B***, хотя он работает как обычный метод в родительском классе ***A***, что я и продемонстрировал, создав экземпляр ***$p1*** класса ***A***. Видимо, встречаются ситуации, когда желательно иметь в классе методы, недоступные для наследования. Попытка вызова такого метода в дочернем классе генерирует сообщение об ошибке.

В Perl6 возможно множественное наследование. Для того, чтобы указать, что класс ***C*** наследует классам ***A*** и ***B***, применяется такой синтаксис: ***class C is A is B { … }***:

```
class A {
    method m1 {say "Это класс A"}
}
class B {
    method m2 {say "Это класс B"}
}
class C is A is B {
    method m3 {say "Это класс C"}
}
my $p = C.new();
$p.m1; → Это класс A
$p.m2; → Это класс B
$p.m3; → Это класс C
```

Отметим, что при этом класс ***B*** не наследует классу ***A***, то-есть, если создать экземпляр:

```
my $p1 = B.new();
```

то вызов:

```
$p1.m1;
```

приведёт к ошибке — метод ***m1*** недоступен в классе ***B***.

5.3 Role

Очевидно, что механизм наследования предназначен для возможности повторного использования ранее созданного кода. Как и все другие современные языки, Perl6 имеет и другие способы повторного использования. К тому же, например, множественное наследование может создавать некоторые проблемы (больше в теории, чем на практике). В частности, в Perl6 существует такая вещь, как ***role***. Фактически это то же самое, что обычно называют миксинами (mixin), трейтами (trait), интерфейсами (interface). (Кажется, что

авторы Perl изначально задались целью ни в коем случае не применять привычные термины. Даже странно, что они всё-таки используют такие слова, как class, method и тому подобное, а не придумали что-нибудь оригинальное и в этих случаях).

Применение role создаёт более простой и надёжный способ повторного использования — код из этих role просто подмешивается к коду создаваемого класса. Для такого подмешивания принято служебное слово ***does***. Сам role объявляется точно также, как и класс. Давайте вернёмся к примеру с классом Proba и весь код из этого класса переместим в role:

```
role R  {
        has $.x;
        has Int $.y is rw;
        has Int $!z = 7;
        method f($k)  {
                $.y -= 3;
                $!z -= 2;
                $.x + $.y + $!z + $k;
                }
}
class Proba does R  {
        method g {
                say ($.x);
        }
}
my $p = Proba.new(x => 5);
$p.y = 10;
say $p.f(3); → 20
$p.g; → 5
```

Как видим, класс ***Proba*** с подмешанным role ***R*** ничем не отличается от класса ***Proba*** в прошлом примере. Здесь я ввёл в класс ***Proba*** метод ***g***, который выводит на консоль значение переменной экземпляра ***$.x***, в классе Proba вообще не объявленной, она объявлена только в role ***R***. Значит, весь код из ***R*** просто вставляется в код класса и происходит это на этапе компиляции, иначе зафиксировалась бы ошибка применения не объявленной переменной ***$.x***. Да и передача параметра методу ***new*** при создании объекта ***$p*** происходит так, как будто переменная ***$.x*** объявлена в классе ***Proba***. Итак, применение

role, в противоположность несколько загадочному наследованию классов, сводится к простой механической вставке кода, который мы желаем повторно использовать. Что может быть проще? При этом можно подмешивать к классу сколько угодно role, без риска получить проблемы, связанные с множественным наследованием.

Посмотрим ещё, как будет вести себя программа, если и в классе и в подмешиваемом к нему role объявляется метод с одним и тем же именем :

```
role R  {
        has $.x;
        method f  {
                say $.x**2;
                }
}
class Proba does R  {
        method f {
                say $.x**3;
        }
}
my $p = Proba.new(x => 5);
$p.f;  → 125
```

Значит, здесь был выполнен тот метод ***f***, который объявлен в классе, а не в role ***R***. Значит, при подмешивании одноименные методы не переопределяются, видимо, они просто игнорируются.

Когда role подмешивается к классу, то он будет подмешан ко всем объектам, создаваемым с помощью этого класса. Но можно role подмешивать с помощью того же слова does и к отдельным объектам уже после их создания. Тогда класс будет способен создавать другие объекты без подмешанного role. Изменим рассмотренный пример:

```
role R   {
        has Int $.y is rw;
        has Int $!z = 7;
        method f($k)  {
                $.y -= 3;
                $!z -= 2;
                $.x + $.y + $!z + $k;
                }
}
```

```
class Proba  {
        has $.x;
        method g {
                say ($.x);
        }
}
my $p = Proba.new(x => 5);
$p does R;
$p.y = 10;
say $p.f(3); → 20
$p.g; → 5
```

Всё работает также, только объявление переменной ***$.x*** пришлось перенести из role в класс, так как на момент создания объекта ***$p*** код из role ещё не подмешан и конструктор не может инициировать переменную ***$.x***. Можно, конечно, написать и в одной строке:

```
my $p = Proba.new(x => 5) does R;
```

Role не только похож на класс, его можно использовать, как самостоятельный класс. В частности можно создавать экземпляры role и работать с ними точно также, как с экземплярами класса:

```
role R  {
        has $.x;
        method f  {
                say $.x**2;
                }
}
my $p = R.new(x => 7);
say $p.WHAT; → (R)
$p.f; → 49
```

Экземпляры role также имеют тип, совпадающий с именем role. Эта способность role к использованию как класса отличает его от миксинов, трейтов, интерфейсов в других языках.

5.4 Self

В теле класса можно обращаться к переменным и методам экземпляра класса с помощью служебного слова ***self***. Это слово указывает на принадлежность того, что за ним стоит, тому объекту (invocant) , для которого вызывается переменная или метод.

Рассмотрим пример класса ***Point***, представляющего точку на плоскости:

```
class Point {
   has Real $.x;
   has Real $.y;
   method c {  return (self.x, self.y) }
   method r { (self.x ** 2 + self.y ** 2) ** 0.5 }
   method pc {
      my $ro = self.r;
      my $fi = atan2 self.y, self.x;
      return $ro, $fi;
   }
}
my $p =  Point.new( x => 4, y => 3 );
say $p.c; → (4 3)
say $p.r; → 5
say $p.pc; → (5 0.643501108793284)
```

В классе Point имеется две переменных экземпляра ***$.x*** и ***$.y***, представляющих координаты точки на плоскости в прямоугольной системе координат, и три метода экземпляра: метод ***c*** возвращает координаты точки, метод ***r*** вычисляет радиус-вектор точки и метод ***pc*** вычисляет координаты точки в полярной системе координат. В методе ***pc*** используется метод ***r***, который вызывается с использованием слова ***self***. Математическая функция ***atan2*** принимает два аргумента — длины катетов, это аналог функции ***atan self.y/self.x*** . Слово self нельзя только применить при объявлении переменных, то-есть недопустим вариант:

```
has Real self.x;
has Real self.y;
```

В остальном слово ***self*** в этом контексте можно считать просто синонимом sigil ***$***, то-есть класс ***Point*** может иметь и такой вид:

```
class Point {
   has Real $.x;
   has Real $.y;
   method c {  return ($.x, $.y) }
   method r { ($.x ** 2 + $.y ** 2) ** 0.5 }
   method pc {
      my $ro = $.r;
```

```
        my $fi = atan2 $.y, $.x;
        return $ro, $fi;
    }
}
my $p =  Point.new( x => 4, y => 3 );
say $p.c; → (4 3)
say $p.r; → 5
say $p.pc; → (5 0.643501108793284)
```

Возможно, что в каких-то случаях этот термин self и необходим, но чаще всего его можно заменить на $, раз это одно и то же.

5.5 Применение объектов одного класса в качестве атрибутов другого класса

Поскольку экземпляры класса являются переменными с типом, созданным классом, то такие переменные могут использоваться в других классах на равных правах с полями базовых типов. Для иллюстрации рассмотрим такой примитивный пример. Создадим класс ***Rectangle***, представляющий прямоугольники со сторонами, параллельными осям координат. Пусть эти прямоугольники определяются их шириной, высотой и координатами левого нижнего угла. И пусть этот класс позволяет находить координаты верхнего левого и верхнего правого угла. Класс ***Rectangle*** может выглядеть, например, так:

```
class Point {
    has Real $.x;
    has Real $.y;
}
class Rectangle  {
        has Real $.w;
        has Real $.h;
        has Point $.ln;
        method lv  {
                return($.ln.x, $.ln.y + $.h);
        }
        method pv  {
                my $.p = Point.new(x => $.ln.x + $.w, y => $.ln.y + $.h);
                return $.p;
        }
```

```
}
my $t = Point.new(x => 1, y => 2);
my $r = Rectangle.new(w => 5, h => 3, ln => $t);
say $r.lv; → (1 5)
say $r.pv; → Point.new(x => 6, y => 5)
```

Для краткости я в классе Point оставил только объявление полей. В классе Rectangle поле ***$.ln***, определяющее координаты левого нижнего угла прямоугольника, имеет тип ***Point***, соответственно, конструктору надо передавать объект этого класса. Метод ***lv*** возвращает координаты левого верхнего угла, а метод ***pv*** возвращает объект класса ***Point***, представляющий координаты правого верхнего угла. Обращаю внимание на то, что при передаче методу ***say*** объекта на терминал выводится сообщение, представляющее конструктор этого объекта вместе с переданными ему параметрами, которые в нашем случае являются координатами точки.

5.6 Метод Delegation

В некоторых языках используется механизм, получивший название delegat. В Perl6 также доступна подобная техника с помощью приёма, называемого в документации метод **Delegation**. Применение наследования делает доступными все методы родительского класса или role. Delegation же позволяет отобрать (делегировать) из одного класса для использования в другом классе только нужные методы . Для такого отбора используется служебное слово ***handles***. Сам механизм делегирования методов очень прост (в отличие от других языков). Давайте посмотрим на простейшем примере:

```
class A  {
        has $.x;
        method m1 { $.x; }
        method m2 { $.x**2; }
        method m3 { $.x**3; }
        method m4 { $.x**4; }
}
class B  {
        has $.m is rw handles ["m1", "m2", "m4"];
}
my $p = B.new;
$p.m = A.new(x => 3);
```

say $p.m1; → 3
say $p.m2; → 9
say $p.m4; → 81

Здесь в классе ***B*** с помощью слова ***handles*** делегируется часть методов из класса ***A***. Названия этих методов перечисляются в качестве элементов списка в текстовом формате. При этом создаётся переменная ***$m***, которая затем может быть инициирована объектом класса ***A***. Чтобы инициация была возможна, надо эту переменную сделать mutable, то-есть объявить как ***is rw***. В результате этих манипуляций в объекте ***$p*** класса ***B*** становятся доступны делегированные методы ***m1, m2, m4***. Попытка вызова метода ***m3*** вызовет ошибку, поскольку он отсутствует в списке делегатов. (Кстати, в краткой форме текстового списка строка делегирования может выглядеть так:
has $.m is rw handles <m1 m2 m4>;)

5.7 Конструкторы

При создании экземпляра класса мы используем метод new, который является встроенным конструктором и всегда предлагается классу по умолчанию. При желании конструктор можно изменить, для чего применяется встроенный метод ***bless***, которому достаточно только передать аргументы в нужном виде. Например, можно создать конструктор, который будет принимать не именованные аргументы (то-есть не в форме пар, как это было во всех примерах ранее). Новый конструктор можно назвать тем же именем new, что приведёт к переопределению встроенного метода new:

```
class Point {
   has $.x;
   has $.y;
   method new ($a, $b) {
      $.bless(x => $a, y => $b);
   }
   method coor  {
      return ($.x, $.y)
   }
};
my $p = Point.new(3, 5);
say $_ for $p.coor; → 3 5
```

Значит, достаточно было только передать методу ***bless*** аргументы в форме именованных, инициируя их аргументами метода ***new***. В следующем примере принудительно укажем тип аргументов, передаваемых конструктору, которому, к тому же, дадим новое имя ***con***:

```
class Point {
   has $.x;
   has $.y;
   method con (Int $a, Int $b) {
      $.bless(x => $a, y => $b);
   }
   method coor{
      return ($.x, $.y)
   }
};
my $p = Point.con(3, 5);
say $_ for $p.coor; → 3 5
```

Теперь конструктор ***con*** может принимать только целые числа и в не именованном формате (то-есть, в позиционном).

5.8 Применение функций в качестве атрибутов класса

Функции в Perl6 тоже представляют объекты, которые имеют тип ***(Sub)***, это можно проверить с помощью метода WHAT (применяется такой синтаксис: ***say &f.WHAT;***). Следовательно, функции тоже можно использовать в качестве атрибутов (полей) класса. Пример подобного использования функции может быть, например, таким:

```
class Prob  {
        has $.x;
        has $.y;
        has &.f;
        method m { return &!f($.x) + $.y; }
}
sub g($t) { $t ** 2; }
my $p = Prob.new(x => 7, y => 51, f => &g);
say $p.m; → 100
sub g1($t) { sqrt($t); }
my $p1 = Prob.new(x => 8, y => 2.5, f => &g1);
say $p1.m; → 5.32842712474619
```

В классе Prob объявлены три поля: два из них — переменные базового типа, а третье — функция (имеет тип (Sub)). Конструктор инициирует это поле, соответственно, функцией ***g***. При использовании поля ***&.f*** в методе ***m***, оно должно быть приватным, то-есть, иметь twigil (!) - ***&!f($.x)***. При этом, поля базовых типов допускают оба варианта — точка и восклицательный знак.

Поскольку тип полей ***$x*** и ***$***y явно не указан, можно также использовать, например, функцию ***g1***, с аргументом типа ***Real***.

Кстати, если в программе объявить переменную вида:

my ≫

то она будет иметь тип ***(Callable)***:

s***ay &gg.WHAT; → (Callable)***

То-есть, это что-то такое, что можно вызывать, как функцию или метод. Видимо, тип ***(Sub)*** является подтипом для ***(Callable)***.

Эффектнее всего передавать конструктору closure:

```
class Prob  {
        has $.x;
        has $.y;
        has &.f;
        method m { return &!f($.x) + $!y; }
}
my $p = Prob.new(x => 3, y => 4, f => -> $t { $t **3; });
say $p.m; → 31
```

Тут можно было бы также применить topic variable $_:

```
f => { $_ ** 3; }
```

6. Разные трюки Perl6

6.1 Создание своих собственных операторов и переопределение существующих

Perl6 позволяет создавать свои собственные операторы (мета-операторы, о которых уже упоминалось ранее), присваивая им произвольные имена. Например, создадим оператор ***(^)***, умножающий свой операнд на 2:

```
sub prefix:<^> (Int $x) { $x * 2; }
say ^7; → 14
```

Итак, здесь применяется такой синтаксис: ключевое слово sub, слово ***prefix*** (определяет префиксный оператор), двоеточие, идентификатор оператора в угловых скобках (тут недопустимы пробелы), операнд с необязательным указателем типа, блок, определяющий работу оператора. Операнд можно заключать в скобки:

Say ^(9); → 18

Заменив слово ***prefix*** на ***postfix***, получим постфиксный оператор:

```
sub postfix:<^> (Int $x) { $x * 2; }
say 8^; → 16
```

Можно применять и в точечной нотации:

```
say 12.^; → 24
```

Можно создавать и инфиксные операторы:

```
sub infix:<^> ($x, $y) {$x * $y;}
say 2 ^ 3; → 6
```

Посмотрим теперь на более содержательный пример — создадим оператор, позволяющий вычислять факториал числа ***n*** с использованием обычного математического синтаксиса — ***n!***

```
sub postfix:<!> (Int $n where $n >= 0) { [*] 2..$n; }
say 5!; → 120
```

При объявлении операнда, с помощью ключевого слова ***where*** введено дополнительное условие, проверяющие операнд на не отрицательность. Нарушение этого условия вызовет исключение. Сам факториал вычисляется путём перемножения элементов соответствующего ранга. Вычислим факториал нуля:

say 0!; → 1

Как же получается эта единица? Дело в том, что выражение ***[*] 2..$n***; вычисляет произведение элементов, начиная с исходного значения, принятого равным ***1***. Сам ранг ***2..0*** не имеет элементов и в результате остаётся одна только исходная единица. Также вычисляется и ***1!*** Если заменить ***(*)*** на ***(+)***, получим соответствующую сумму, но только ранг надо заменить на ***1..$n***.

Для того, чтобы переопределить любой оператор, в том числе и базовый, достаточно использовать его идентификатор:

```
sub prefix:<+> (($x, $y)) { $x * $y; }
+(2,3); → 6
```

Итак, теперь оператор ***(+)*** стал выполнять умножение. Двойные скобки ***(($x, $y))*** нужны потому, что операнд здесь всегда заключается в скобки, а операнд в этом примере представляет список.

Если применять ключевое слово ***multi***, то можно создавать универсальные операторы, определяющие свои действия, например, по типу операнда:

```
multi sub prefix:<+> ((Int $x, Int $y)) { $x + $y; }
multi sub prefix:<+> ((Str $x, Str $y)) { $x ~ $y; }
+(3,4); → 7
+('aa','bb'); → aabb
```

Теперь оператор ***(+)*** или суммирует два числа, или выполняет конкатенацию двух текстовых значений в зависимости от типа переданных ему данных.

Приведу ещё пример оператора, объединяющего два или более hash в один. Применим идентификатор ⊕:

```
multi sub infix:<⊕> (%a, %b) {
      return {%a, %b};
}
my %q1 = jan => 1, feb => 2, mar => 3;
my %q2 = apr => 4, may => 5, jun => 6;
my %first_half = %q1 ⊕ %q2;
say %first_half; → {apr => 4, feb => 2, jan => 1, jun => 6, mar => 3,
may => 5}
my %q3 = jul => 7, aug => 8, sep => 9;
my %q4 = oct => 10, nov => 11, dec => 12;
my %year = [⊕] %q1, %q2, %q3, %q4;
say %year; → {apr => 4, aug => 8, dec => 12, feb => 2, jan => 1, jul =>
7, jun => 6, mar => 3, may => 5, nov => 11, oct => 10, sep => 9}
```

Здесь интересно то, что созданный нами инфиксный оператор ⊕ можно использовать в нотации ***[⊕] @a***, где @a – список с элементами, представляющими hash, которые надо объединить.

Впрочем, тот же результат можно получить и так:

```
my %year = [,] %q1, %q2, %q3, %q4;
```

6.2 Оператор ранга и ленивые списки

Ранее мы уже видели применение бесконечных ленивых последовательностей. Добавим ещё некоторые детали.

Оператор три точки ***(…)*** создаёт ленивые последовательности:

my $r := (0 ... 200);
Последовательность $r имеет тип Seq:
$r.WHAT; → (Seq)
Знак присваивания ***(:=)*** позволяет получать immutable последовательности. Но если инициировать этой последовательностью список (при этом требуется применить метод ***cache***), он не будет ленивым:
my @a = $r.cache;
То же самое получим и при использовании ленивого ранга непосредственно:
my @a = (0 ... 200);

Для получения ленивого списка применяется оператор ***lazy***:
my @a = lazy 1 ... 200;
say @a; → [...]
say @a.elems; → Cannot .elems a lazy list
Любой элемент с индексом от 0 до 199 генерируется при его использовании:
say @a[125]; → 126
При этом сам список остаётся по прежнему пустым:
say @a; → […]
Но если мы попытаемся использовать элемент с индексом за пределами ранга, то-есть 200 или более, то получим результат (Any) и при этом будут созданы все элементы списка:
say @a[210]; → (Any)
say @a; - будут выведены все элементы от 1 до 200.

Задавая два первых элемента, можем получать разные арифметические последовательности:
my @od = (1, 3 ... 15); → [1 3 5 7 9 11 13 15]
my @ev = (0, 2 ... 14); → [0 2 4 6 8 10 12 14]
my @a = (0, 5 ... 20); → [0 5 10 15 20]

Подобным же образом, задавая три первых элемента, можем получать геометрические последовательности:
my @a = (1, 3, 9 ... 300); → [1 3 9 27 81 243]
Члены не обязательно должны быть целыми:
my @a = (1.0, 3.2, 10.24 ... 300); → [1 3.2 10.24 32.768 104.8576]
Последовательности могут быть и обратными:
my @a = (12, 10 ... 0); → [12 10 8 6 4 2 0]

Арифметические и геометрические последовательности могут быть бесконечными и тогда они всегда ленивые:

my @a = (1, 3, 9 ... Inf); → […]

say @a[36]; → 150094635296999121

Есть и более мощный способ получения последовательностей, при котором задаётся не следующий элемент, а блок, определяющий способ получения последующих членов. Например, ряд нечётных чисел можно получить так:

my @a = (1, { $_ +2 } ... 11); → [1 3 5 7 9 11]

Такой способ позволяет создавать более сложные ряды, чем арифметические или геометрические последовательности. Например, можно получить функцию, вычисляющую факториалы чисел:

```
my $x;
my @fact = $x = 1, {$_ * $x++} ... *;
say @fact[0..7]; → (1 1 2 6 24 120 720 5040)
```

Поскольку задана бесконечная последовательность, то список ***@fact*** ленивый и факториалы вычисляются только при их использовании. На самом деле ***@fact*** - полноценная функция. Есть возможность представить её в другом виде:

```
my @fact = 1, { state $x = 1; $_ * $x++} ... *;
```

Здесь служебное слово ***state*** является особым декларатором, при котором переменная ***$x*** инициируется единицей только один раз, а затем в цикле всегда сохраняется для неё предыдущее значение.

Впрочем, можно поступить ещё и так:

```
my $x;
sub fact { $x = 1, {$_ * $x++} ... *};
fact[1..7]; → (1 2 6 24 120 720 5040)
```

А так можно создать функцию для чисел Фибоначчи:

```
my @fibo = 0, 1, -> $a, $b { $a + $b } ... *;
say @fibo[^8]; → (0 1 1 2 3 5 8 13)
```

Метод вычисления следующего числа здесь задан с помощью closure. Ранее уже упоминался приём использования placeholder в виде звёздочки (*). Используя placeholder, можно получить более краткую форму:

```
my @fibo = 0, 1, * + * ... *;
```

say @fibo[^8]; → (0 1 1 2 3 5 8 13)

Для непосвящённого такой код может показаться полной абракадаброй.

6.3 Модули

Perl6, как и любой другой язык программирования, позволяет создавать свои собственные библиотеки модулей для повторного использования кода. Техника создания нового модуля очень проста. Для объявления модуля применяется служебное слово ***module***, после которого указывается имя модуля с заглавной буквы. Далее располагается тело модуля в фигурных скобках. Например:

```
module Mod1  {
        class A {
                method f($x) {
                        say($x * $x);
                }
        }
        our sub f($x) { sin($x) }
        our &f1 = -> $x { cos($x) }
        our $z = 777;
print "Нажмите любую клавишу";
my $xx = get;
}
```

Этот код следует поместить в файл с именем ***Mod1.pm***, то-есть, название файла должно совпадать с названием модуля и иметь расширитель ***.pm***.

Модуль Mod1 содержит класс, две функции, созданные разными способами и переменную. Использовать модуль можно, например, так. Создадим такую головную программу в файле ***rab.pl***:

```
use lib ".";
use Mod1;
```

my $p = Mod1::A.new;
$p.f(9); → 81
say Mod1::f(1.5); → 0.997494...
say Mod1::f1(1.5); → 0.0707372...
say Mod1::{'$z'}; → 777

Строка ***use lib ".";*** указывает, что модуль будет отыскиваться в текущей директории. Следовательно, перед вызовом модуля надо перейти в его директорию. Строка ***use Mod1;*** загружает модуль и, кстати, выполняет в нём код, который может быть выполнен. В нашем примере при выполнении строки use Mod1; будет выведен на терминал текст ***«Нажмите любую клавишу»*** и программа остановится в ожидании этого нажатия. После этого классы, функции и переменные из модуля будут доступны в вызывающей программе.

Есть, конечно, некоторые особенности в коде модуля. К объявлению классов здесь не предъявляется никаких дополнительных требований. Но функции и переменные должны быть объявлены с declarator ***our*** вместо ***my***. Declarator ***our*** определяет неограниченную область видимости, в то время, как ***my*** задаёт область видимости только в пределах текущего блока. Бывают и другие ситуации, когда применяется declarator our; не будем эту тему рассматривать подробнее. Наконец, при вызове переменных модуля в головной программе применяется особый синтаксис, а именно идентификатор переменной заключается в кавычки и располагается в фигурных скобках: ***{'$z'}***.

Как обычно, Perl6 предоставляет и другие способы создания модулей.

7. Немного о функциональном программировании

Функциональный стиль программирования предполагает использование одних только чистых функций, то-есть функций без побочных эффектов. К побочным эффектам могут относиться,

например, промежуточные (или вспомогательные) переменные. Если мы запрограммируем функцию, используя такую переменную, то кроме получения интересующего нас результата будем иметь ещё и эту переменную с каким-то остаточным значением. Такие остаточные значения являются мусором, загромождающим память компьютера. Современные языки, включая и Perl6 имеют специальные мусорщики для удаления ставших ненужными данных.

На самом деле ни один язык программирования не позволяет полностью исключить побочные эффекты. Например, побочным эффектом является вывод данных на внешний носитель. Если чистая на первый взгляд функция использует какую-нибудь библиотечную функцию, то часто мы не знаем, имеет или нет посторонняя функция побочные эффекты, и это значит, что мы не можем гарантировать, что наша функция действительно чистая.

Имеется два основных приёма, обычно применяемых при программировании чистых функций. Во-первых это использование только неизменяемых (immutable) переменных, которые гарантируют, что после выхода из функции, состояние памяти осталось таким, каким оно было до работы функции. Во-вторых это организация циклов только с использованием рекурсии, которая не требует применения вспомогательных переменных (параметров цикла). Язык Perl6 позволяет использовать эти приёмы в полной мере, причём immutable обеспечивается автоматически, поскольку аргументы функций не могут быть изменены в их теле. Рассмотрим несколько примеров чистых функций. Обычно начинают с функции вычисления факториала числа. Не будем нарушать традицию:

```
sub fact($n where $n >= 0)  {
        given $n  {
                when !$n { 1 }
                default { $n * fact($n - 1) }
        }
}
my $n = 5;
say "$n! = " ~ fact $n;  → 5! = 120
```

Для аргумента функции ***fact*** мы ввели условие проверки на не отрицательность, нарушение которого вызовет исключение. Сам

алгоритм функции построен на переключателе ***given – when*** и на рекурсии для организации цикла. Если аргумент ***$n*** равен нулю, результат приравнивается единице и происходит выход из функции (напоминаю, что ***!0*** соответствует значению ***True***). Если аргумент больше нуля, значение ***$n*** умножается на рекурсивный вызов функции ***fact*** с аргументом, уменьшенным на единицу. Таким образом строится цепочка произведений:

n * (n-1) * (n-2) … 1

При этом нет никаких промежуточных переменных и значит функция ***fact*** чистая. Ну, а оператор ***say*** конечно имеет побочный эффект, но тут уж ничего не поделаешь.

Как обычно, Perl6 позволяет сделать всё это и другими способами. Например, всё то же самое можно получить, применив функцию ***multi*** вместо переключателя ***given – when***:

```
multi sub fact(0) { 1 };
multi sub fact(Int $n where $n > 0) {
   $n * fact $n - 1;
}
say fact 0;   → 1
say fact 10;  → 3628800
```

Perl6 позволяет написать для факториала числа совсем короткую программу:

```
sub fact($n) {[*] [1..$n]}
say fact(5); → 120
```

Но теперь мы не можем быть уверенными, что эта функция чистая, поскольку не знаем, представляет ли чистую функцию мета-оператор ***[*]***, использованный здесь.

Посмотрим ещё на пример с использованием списка:

```
sub f(@a, $s is copy = 0)  {
        given @a  {
                when [] { $s }
                { $s += @a.shift; f(@a, $s) }
        }
}
```

say f([1,2,3,4,5]); → 15

Здесь вычисляется сумма элементов списка с использованием переменной- накопителя ***$s***. Обычно такую функцию считают чистой, хотя строго говоря, результат не просто возвращается функцией, а накапливается в ***$s***, изменяя эту переменную по ходу дела. То-есть, в принципе здесь есть побочный эффект. Хотя, конечно, изменяется здесь не сама переменная ***$s***, а только её копия. А вот пример такой же функции, но уже действительно без побочных эффектов:

```
sub f(@a)  {
        given @a  {
                when [] { 0 }
                { @a[0] + f(@a.tail(@a.elems - 1)) }
        }
}
say f([1,2,3,4,5]); → 15
```

Можно, конечно, и так:

```
sub f(@a)  {
        given @a  {
                when [] { 0 }
                { @a.shift + f(@a) }
        }
}
say f([1,2,3,4,5]); → 15
```

Практически все современные языки позволяют программировать в функциональном стиле. Но мне кажется, что если уж у вас появилось желание изучить функциональный стиль и программировать таким способом, то лучше использовать чисто функциональный язык Hascell, в котором всё приспособлено для этой цели. Не будем здесь рассматривать эту тему более подробно.

Заключение

Те возможности языка Perl6, которые мы вкратце рассмотрели на данный момент, разумеется не исчерпывают тему. Язык программирования Perl изначально предназначался в основном для анализа (parsing) текста. Версия языка Perl6 обладает достаточно универсальными возможностями, ни в чём не уступающими, а в некоторых аспектах и превосходящими, возможности большинства современных языков программирования. Вместе с тем, технике анализа текста с применением регулярных выражений (regex) и в Perl6 также уделяется много внимания. В частности, имеется возможность использовать такой оригинальный и мощный механизм, как Grammar. Тем не менее, я считаю, что в целом тема regex достаточно рутинная и легко может быть изучена самостоятельно, а потому здесь я её не рассматриваю. Так же не затронуты мною не менее рутинное программирование исключений, веб-программирование и некоторые другие вопросы. Считаю, что для первоначального знакомства с языком достаточно овладеть рассмотренным здесь материалом.