Содержание

1. Базовый синтаксис

Про язык Perl существует миф, что это write-only язык. На самом деле, если текст понимает интерпретатор, то сможет прочесть и человек. Цель этой лекции — научить читать исходные тексты практически любой программы.

Основная идея, лозунг Perl — есть более одного способа сделать что-либо в этом языке (*TIMTOWTDI*, *There's More Than One Way To Do It*). При написании программы стоит следовать пути удобочитаемости ее текста. К сожалению, не все придерживаются этого принципа, и зачастую прочесть программу может лишь интерпретатор Perl: *the only thing can parse Perl (the language) is perl (the binary).*

1.1. Условия и циклы

Одной из простейших конструкций является *блок* — ограниченный фигурными скобками код. Внутри блока размещаются несколько раздельных *statement*:

```
{
statement;
statement;
...
}
```

При этом блок ødo не является блоком, а является единичным *statement*. В отличие от блока, то, что написано в фигурных скобках *do*, возвращает свое значение:

```
do { ... } | { ... }

$value = do { ... }

$value = { ... };
```

Блок используется в существующих конструкциях для создания отдельных областей видимости.

1.2. Управляющие функции

Для управления циклами существует управляющая конструкция *next*. В языке Си есть аналог *continue*. При этом *next* можно вызвать в любом месте тела цикла, при этом исполнение перенесется в начало блока *continue* цикла. Если такого блока нет, то исполнение перенесется в конец тела и перейдет на следующую итерацию:

```
for my $item ( @items ) {
        my $success = prepare($item);

        unless ($success) {
                next;
        }

        process($item);
} continue {
        # next переходит сюда
        postcheck($item);
}
```

Команда *last* позволяет выйти из цикла и прекратить его исполнение без выполнения блока *continue*. В следующем примере показан цикл *for*, но на его месте может быть любой цикл — *while*, *until* и другие:

```
for my $item ( @items ) {
        my $success = prepare($item);

        unless ($success) {
                last;
        }

        process($item);
} continue {
        postcheck($item);
}
# last переходит сюда
```

В языке Си аналогом *last* является оператор *break*.

Команда *redo* начинает итерацию цикла — переходит в начало цикла без изменения переменной состояния. Если просто написать внутри цикла *redo*, то получится бесконечный цикл, так как он не изменяет никакие переменные. В некоторой степени это аналог оператора *goto* в Си:

```
for my $item ( @items ) {
        # redo переходит сюда
        my $success = prepare($item);

        unless ($success) {
                redo;
        }
```

```
        process($item);
} continue {
        postcheck($item);
}
```

Еще одна особенность блока заключается в том, что любой пустой блок идентичен циклу с одной итерацией. В таком блоке работают вызовы *redo*, *last*, *next*. Оператор *redo* идет в начало блока, *last* выходит из блока, а *next* переходит в конец блока:

```
{
        # redo
        stmt;
        if (...) { next; }

        stmt;
        if (...) { last; }

        stmt;
if (...) { redo; }

        stmt;
        # next
}
# last
```

В языке Perl существует оператор выбора *given/when* (аналог *switch/case* из Си). Внутри — не обычный *case*, а довольно сложный оператор *smart match*. Внутри *when* можно писать разные выражения:

```
use feature 'switch'; # v5.10+

given ( EXPR ) {
        when ( undef )    { ... }
        when ( "str" )    { ... }
        when ( 42 )       { ... }
        when ( [4,8,15] ) { ... }
        when ( /regex/ )  { ... }
        when ( \&sub )    { ... }
        when ( $_ > 42 )  { ... }
        default           { ... }
}
```

В отличие от других языков, здесь нет ключевого слова *break* — *break* автоматически подразумевается внутри каждого условия *when*. Для «проваливания» в другой *when* используется *continue*.

В языке Perl есть определенные функции и операторы, которые имеют множественное поведение. Один из примеров — *goto*, используемый для перехода. Данный оператор является классическим, и, как и в других языках, выполняет переход на метку:

```
goto LABEL;

LABEL1:
        say "state 1";
        goto LABEL2;
LABEL2:
        say "state 2";
        goto LABEL1;

state 1
```

```
state 2
state 1
state 2
...
```

Метка записывается статически, она объявлена в программе.

Существует также устаревший синтаксис, использовать который не рекомендуется. Здесь в качестве параметра *goto* можно передать строку, а имя строки будет воспринято как метка:

```
goto EXPR; # DEPRECATED

{
EVEN:
        say "even";
        last;
ODD:
        say "odd";
        last;
}

goto(
        ("EVEN","ODD")[ int(rand 10) % 2 ]
);
```

Хвостовой вызов позволяет вызвать тело другой функции на месте этой функции с теми аргументами, которые есть сейчас (*goto &*). В следующем примере показано классическое написание функции вычисления чисел Фибоначчи, которая делает рекурсивный вызов:

```
goto &NAME;
goto &$var;

sub fib {
        return 0 if $_[0] == 0;
        return 1 if $_[0] == 1;
        return _fib($_[0]-2,0,1);
}

sub _fib { my ($n,$x,$y) = @_;
        if ($n) {
                return _fib( $n-1, $y, $x+$y );
        }
        else {
                return $x+$y;
        }
}
```

Известная проблема с рекурсией заключается в том, что у нее растет стек. Если же заменить строку вызова функции на выражение, показанное в следующем примере, то эта функция будет выполнена непосредственно на данном месте (без создания отдельного *stack frame*):

```
goto &NAME;
goto &$var;

sub fib {
        return 0 if $_[0] == 0;
        return 1 if $_[0] == 1;
        return _fib($_[0]-2,0,1);
}
```

```
sub _fib { my ($n,$x,$y) = @_;
        if ($n) {
                @_ = ( $n-1, $y, $x+$y ); goto &_fib;
        }
        else {
                return $x+$y;
        }
}
```

Аналогичным образом можно написать расчет факториала. Функция также написана в рекурсивном стиле, но при этом она никогда не вызовет переполнение стека.

```
goto &NAME;
goto &$var;

sub fac {
        my $n = shift;
        return _fac($n,1);
}

sub _fac {
        my ($n,$acc) = @_;
        return $acc if $n == 0;
        @_ = ($n-1,$n*$acc);
        goto &_fac;
}
```

1.3. Постфиксная нотация

Помимо классического написания условий и циклов, в языке Perl возможна *постфиксная нотация*. Она подразумевает, что *statement* выполнится независимо от того, выполняется условие или нет:

```
STMT if EXPR;

STMT unless EXPR;

STMT while EXPR;

STMT until EXPR;

STMT for LIST;

STMT when EXPR;
```

Постфиксная нотация используется, когда *statement* в приложении важнее чем то, что идет после него. Так можно обратить внимание того, кто читает программу, на то, что именно сейчас выполняется. Постфиксная нотация может использоваться внутри *given/when*.

Так как *do* является *statement*, то можно после него написать любой из блоков (*while*, *until*, *for*, *if*):

```
do {
        ...;
} while ( EXPR );

do {
        ...;
} until ( EXPR );
```

```
do {
        ...;
} for ( LIST );
```

Это выглядит как постфиксный цикл, но у него есть определенная особенность: не работают все функции управления циклом, так как эти операторы работают внутри блока.

2. Переменные

2.1. Основные типы

Любой идентификатор переменной записывается следующим образом. Значком, который называется *Sigil*, может быть один из пяти символов: $, @, %, &, * (см. рис. 1). Символ $ означает скалярный контекст или скалярную переменную, символ @ — либо массив, либо списковый контекст. Символ % относится к хэшу, символ & — к блоку кода (к указателю на функцию). Символ * означает специальный тип *glob*.

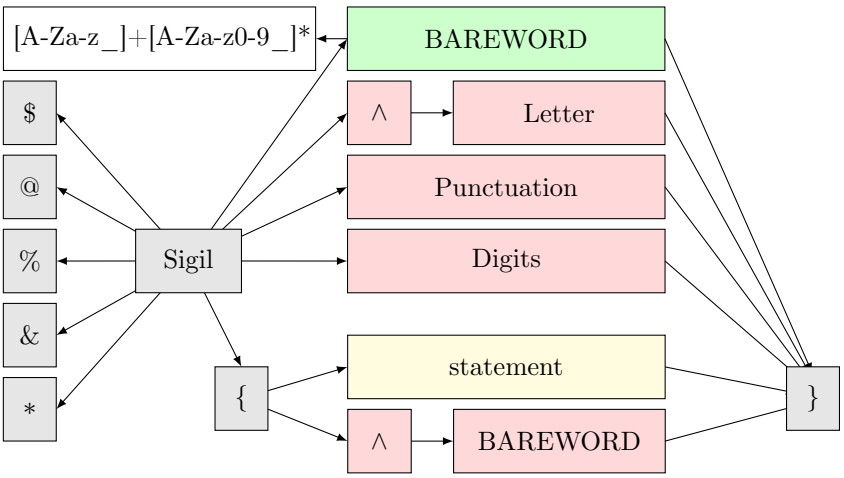

Рис. 1: Правила составления идентификаторов переменных в Perl

После *Sigil* в классическом случае будет идти переменная *Bareword* — любое сочетание букв, которое начинается с большой, маленькой буквы или с подчеркивания и продолжается тем же набором плюс цифры.

Вокруг любой конструкции, идущей после *Sigil*, можно поставить фигурные скобки. Кроме *Bareword*, после *Sigil* может идти спецпеременная. Таковыми являются: +*Letter*, единичный элемент пунктуации, цифры. Внутри фигурных скобок также может быть написан любой *statement*, а также спецпеременная +*Bareword* (см. рис. 1). Примеры обычных переменных:

- $var, @array, %hash, &func, *glob
- ${var}, @{array}, %{hash}, &{func}, *{glob}
- ${ ”scalar” . ”name” }, %{ ”hash” . $id }

Примеры специальных переменных:

- $^W, $^O, $^X, ${^W}, ${^O}, ${^X}, ...
- $0, $1, $100, ${0}, ${1}, ${100}, ...
- ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH}, ...
- $_, @_, $!, $@, $?, $”, $/, $, , ...
- ${_}, @{_}, ${!}, ${@}, ${?}, ${”}, ${/}, ${,}, ...

Числа можно записать в Perl различными способами:

```
$int          = 12345;
$pi           = 3.141592;
$pi_readable = 3.14_15_92_65_35_89;
```

```
$plank       = .6626E-33;
$hex         = 0xFEFF;
$bom         = 0xef_bb_bf;
$octal       = 0751;
$binary      = 0b10010011;
```

Особенность заключается в следующем: в числе можно в любом месте, кроме как рядом с точкой, вставлять подчеркивание. Это делается исключительно для большей легкости прочтения текста программы, интерпретатор такие подчеркивания пропускает.

Строки в Perl бывают привычного вида (в двойных кавычках и в одинарных). Строки в одинарных кавычках являются неинтерполируемыми, в двойных — интерполируемыми. Строку можно разорвать как угодно:

```
$one    = "string";
$two    = 'quoted';
$wrap   = "wrap
ped";
$join   = "prefix:$one";
$at     = __FILE__ .':'. __LINE__;
```

Внутри двойных кавычек работает интерполяция переменных: любой идентификатор, записанный внутри двойных кавычек, будет развернут в свое значение. Строку можно собирать при помощи оператора конкатенации из специальных констант.

Особенность строк в языке Perl состоит в следующем. Если в строке содержится довольно много обычных кавычек, и их использовать неудобно, то можно выбрать любой другой символ для обозначения границы строк. Оператор *q* отвечает за одинарную кавычку, *qq* — за двойную, и далее следует какой-либо разделитель:

```
$q_1    = q/single-'quoted'/;
$qq_2   = qq(double-"quoted"-$two);
```

Внутри строк с двойной кавычкой поддерживается синтаксис записи $\backslash x$ — шестнадцатеричный код юникодного символа. Так можно записать любой символ, который есть в таблице *Unicode*:

```
$smile  = ":) -> \x{263A}";
```

Еще одной особенностью Perl в данном разделе является *v-string*. Он используется для создания и обозначения версий, для сравнения их между собой.

```
$ver    = v1.2.3.599;
```

2.2. Ссылки

Помимо чисел и строк, важно также рассмотреть *ссылки*. Ссылку можно создать при помощи оператора взятия ссылки (\) на любой известный объект:

```
$scalarref = \$scalar;
$arrayref  = \@array;
$hashref   = \%hash;
$coderef   = \&function;
$globref   = \*FH;
$refref    = \$scalarref;
```

Существуют анонимные конструкторы ссылок на массив (квадратные скобки) и на хэш (фигурные скобки). Также можно создать анонимную функцию:

```
$arrayref  = [ 4,8,15,16 ];
$hashref   = { one => 1, two => 2 };
$coderef   = sub { ... };
```

Ссылки можно присваивать спискам, причем поставив ссылку на список, можно создать ссылку на все его элементы:

```
($a,$b)     = (\"one",\"two");
($a,$b)     = \("one","two");
```

Часто необходимо понять, чем является сложная структура — массивом, ссылкой на массив или даже скаляром. *Sigil* означает контекст обращения, как было показано ранее. Появляется специальный *Sigil* из двух элементов $#, обозначающий последний элемент в массиве. Далее идет идентификатор (см. рис. 2). Если после идентификатора идет квадратная или фигурная скобка, то это либо массив, либо хэш.

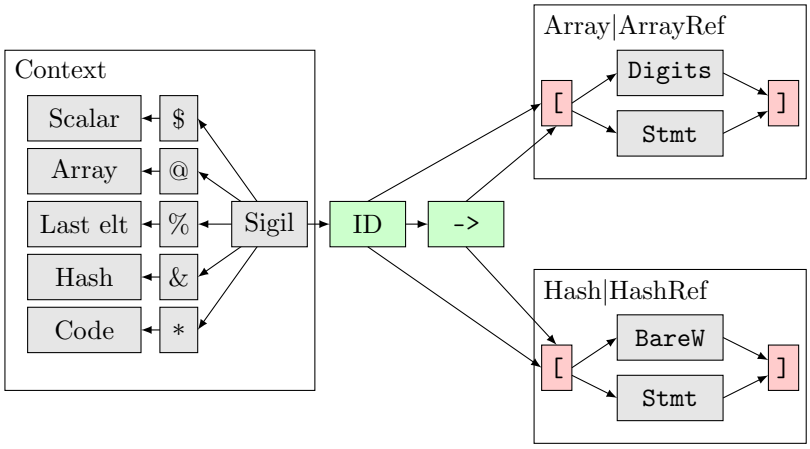

Рис. 2: Правила записи массивов и хэшей

Если же после идентификатора идет стрелка (оператор разыменования ссылки), то после него опять же может идти либо квадратная, либо фигурная скобка. Это означает, что в идентификаторе лежит ссылка — существует переменная, которая является ссылкой на массив или на хэш (см. рис. 3).

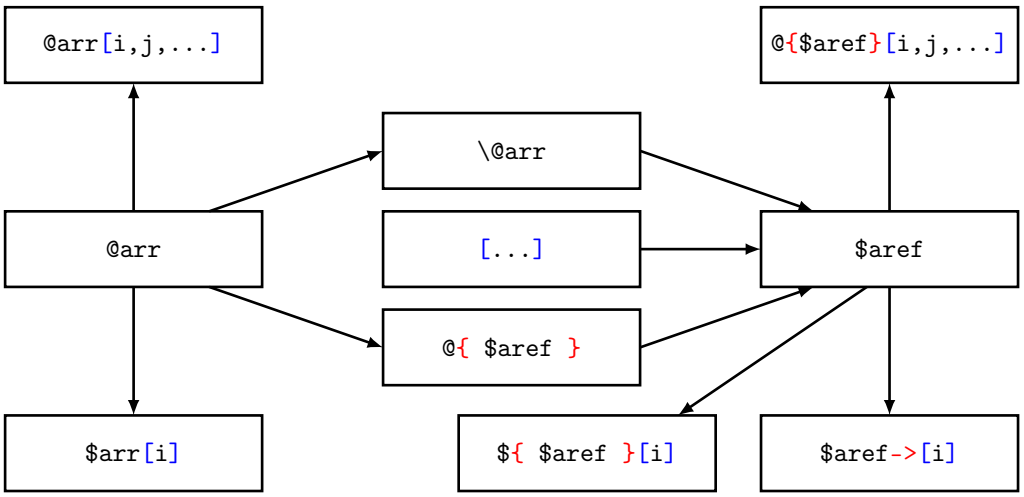

Рис. 3: Способы записи ссылок на массивы

Обычный массив записывается в виде «@ плюс идентификатор». Если нужно взять на него ссылку, то добавляется \ и получается скалярная переменная, в которой находится ссылка на исходный массив (см. рис. 3). Можно также сделать анонимный конструктор.

Если же из ссылки нужно получить массив с @, то есть передать какую-либо функцию или просто развернуть список, то производится операция разыменования (@{}). Фигурные скобки иногда можно опустить (в случае, если это очевидно). Разыменование является обратным преобразованием.

В следующем примере используется оператор *qw*, разделяющий свое содержимое по пробельным символам:

```
@simple = qw(1 2 3 bare);
@array = (4,8,15,16,23,42,@simple);
@array = (4,8,15,16,23,42,1,2,3,'bare');
$aref = \@array;
$aref = [4,8,15,16,23,42,@simple];

say $array[2];   # 15
say ${array}[2];
say ${array[2]};
say "last i = ", $#array;
say "last i = ", $#{array};

say $aref->[2];
say $$aref[2];
say ${$aref}[2];
say "last i = ", $#$aref;
say "last i = ", $#${aref};
say "last i = ", $#{${aref}};
```

Необходимо отметить, что массив всегда инициализируется списком.

У массивов есть *срезы*, используемые для того, чтобы достать из массива часть в виде списка. Здесь используется специальный синтаксис, который встречается довольно редко: идентификатор вместе с квадратными скобками записывается внутри фигурных скобок:

```
@simple = qw(1 2 3 bare);
@array = (4,8,15,16,23,42,@simple);
$aref = \@array;

say join ",", @array[0,2,4]; # 4,15,23
say join ",", @{array}[0,2,4]; # 4,15,23
say join ",", @{ array[0,2,4] }; # 4,15,23
```

Аналогично для ссылок — на то место, где был идентификатор переменной, подставляется ссылка. Так как идентификатор сам по себе является переменной, то и к нему можно добавить фигурные скобки:

```
say join ",", @$aref[0,2,4]; # 4,15,23
say join ",", @{ $aref }[0,2,4]; # 4,15,23
say join ",", @{ ${aref} }[0,2,4]; # 4,15,23
```

Везде, где у массивов квадратные скобки, у хэшей фигурные. Хэш определяется с помощью знака %, ссылка берется так же, как и у массивов, разыменование берется процентом (см. рис. 4). Хэш конструируется анонимно при помощи фигурных скобок. Обращение к одному элементу — фигурные скобки плюс ключ, обращение скалярное.

У хэша тоже есть срез. Так как срез является списковым обращением, то используется @, а не % (см. рис. 4). Обращение к ссылке происходит подобным образом (разыменование ссылки, обращение к фигурным скобкам, перечисление ключей).

Из следующего примера видно, что хэш конструируется так же, как и массив (списком). В качестве указания ключа можно использовать внутри фигурных скобок *Bareword* (можно без кавычек):

```
%simple = qw(k1 1 k2 2);
%hash = (key3 => 3, 'key4',"four", %simple);
$href = \%hash; $key = "key3";

say $hash{key3};
say $hash{"key3"};
say $hash{ $key };
say ${hash}{key3};
say ${ hash{key3} };
```

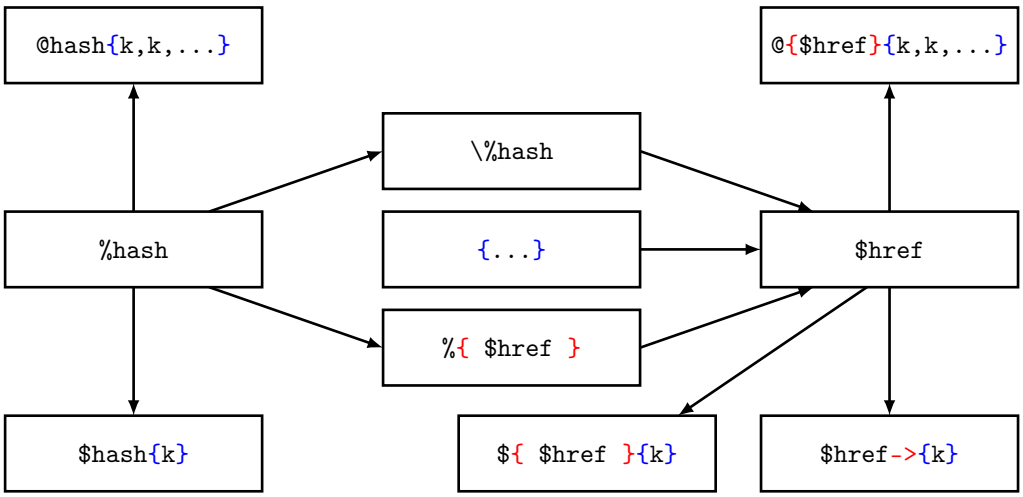

Рис. 4: Способы записи ссылок на хэши

```
say ${ hash{$key} };

say $href->{key3};
say $href->{"key3"};
say $href->{$key};
say ${href}->{key3};
say $${href}{key3};
say ${$href}{key3};
say ${${href}}{key3};
```

Сложный ключ можно взять в любые кавычки. В качестве ключа можно использовать переменную.

Аналогичный пример для срезов:

```
%simple = qw(k1 1 k2 2);
%hash = (key3 => 3, 'key4',"four", %simple);
$href = \%hash; $key = "key3";

say join ",", %simple; # k2,2,k1,1
say join ",", keys %hash; # k2,key3,k1,key4
say join ",", values %$href; # 2,3,1,four

say join ",", @hash{ "k1", $key }; # 1,3
say join ",", @{hash}{ "k1", $key }; # 1,3
say join ",", @{ hash{ "k1", $key } }; # 1,3

say join ",", @{$href}{ "k1","key3" }; # 1,3

$hash{key5} = "five";
$one = delete $href->{k1}; say $one; # 1
say $hash{k2} if exists $hash{k2}; # 2
```

В отличие от массива, хэш можно развернуть в список. Важная особенность Perl: хэши являются несортированными, то есть неважно, в каком порядке их исходно присваивали. Есть специальные функции *keys* и *values*, которые возвращают соответственно ключи хэша и значения, им соответствующие. При этом порядок, который вернет *keys*, и порядок, который вернет *values*, будет одинаковым. В ключ хэша можно присваивать значения или удалять значения из него. «Жирная» запятая (⇒) влияет на свой левый операнд: если он является *Bareword*, то вокруг него ставятся кавычки.

С помощью ссылок можно создавать сложные структуры. Если присваивается хэш или массив, то используются круглые скобки. При инициализации элементы не должны быть одного типа. Можно заметить,

что зачастую «жирная» запятая используется, даже если это не требуется левому операнду. В отличие от обычного хэша, который конструируется списком, ссылка на хэш конструируется фигурными скобками:

```
$var = 7;
%hash = (
        s => "string",
        a => [ qw(some elements) ],
        h => {
                nested => "value",
                "key\0" => [ 1,2,$var ],
        },
        f => sub { say "ok:@_"; },
);

say $hash{s}; # string
say $hash{a}->[1]; # elements
say $hash{h}->{"key\0"}->[2]; # 7
say $hash{h}{"key\0"}[2]; # 7
$hash{f}->(3); # ok:3
&{ $hash{f} }(3); # ok:3
```

Самая частая проблема у того, кто только начинает писать на Perl — путаница между массивами и ссылками на массив. Если в следующем примере создать массив и положить в него вместо круглых скобок квадратные, то получится массив, единственным элементом которого будет ссылка на массив:

```
$, = ", "; # $OUTPUT_FIELD_SEPARATOR
@array = (1,2,3);
say @array; # 1, 2, 3

@array = [1,2,3];
say @array; # ARRAY(0x7fcd02821d38)
```

Аналогичная проблема возникает с хэшем:

```
%hash  = (key => "value");
say %hash; # key, value

%hash  = {key => "value"};
say %hash; # HASH(0x7fbbd90052f0),
```

Другая проблема часто возникает при сборке хэша: вместо массива используется список. В данном примере в итоге получится список из шести элементов, которые попарно будут обращены в хэш:

```
%hash = ( key1 => (1,2), key2 => (3,4) );
say $hash{key1}; # 1
say $hash{key2}; # undef
say $hash{2};    # key2
%hash = ( key1 => 1,2 => 'key2', 3 => 4 );
```

Таким образом, если что-то идет не так, в первую очередь необходимо обратить внимание на то, как именно конструируются структуры.

В случае хэша или массива можно обращаться к несуществующим элементам. При этом Perl по пути будет создавать недостающие элементы, если обращаться к ним глубже:

```
$href = {
s => "string",
};

$href->{none}{key} = "exists";
```

```
say $href->{none};      # HASH(0x7fea...)
say $href->{none}{key}; # exists

$href->{ary}[7] = "seven";
say $href->{ary};       # ARRAY(0x7f9...)
say $href->{ary}[7];    # seven
say $#{ $href->{ary} }; # 7

$href->{s}{error} = "what?";
say $href->{s}{error};  # what?
say $string{error};     # what?
```

Если в данном примере обратиться к существующему элементу, который не является ссылкой, то логично, что программа должна «упасть», однако, этого не происходит. Более того, в процессе создается хэш с именем *string*.

Это явление носит название *символическая ссылка* (переменная, чьё значение является именем другой переменной). Так как язык является динамическим, переменные могут создаваться «на лету» (их не обязательно объявлять заранее), а также можно обратиться к переменной, чье имя известно из другой переменной. В следующем примере к переменной обращаются как к ссылке, как к массиву и как к хэшу:

```
$name = "var";
$$name = 1;     # устанавливает $var в 1
${$name} = 2;   # устанавливает $var в 2
@$name = (3,4); # устанавливает @var в (3,4)

$name->{key} = 7; # создаёт %var и
# устанавливает $var{key}=7

$name->();      # вызывает функцию var
```

Хоть символические ссылки бывают удобны при создании «однострочных» программ, их использование являются плохим подходом в программах более сложного уровня. В связи с этим полезен модуль *strict*, параметр *refs* которого запрещает использование символических ссылок:

```
use strict 'refs';

${ bareword }    #  $bareword; # ok
${ "bareword" }; # not ok

$hash{ "key1" }{ "key2" }{ "key3" }; # ok
$hash{ key1 }{ key2 }{ key3 }; # also ok

$hash{shift}; # ok for keyword, no call
$hash{ +shift }; # call is done
$hash{ shift() }; # or so
```

Важно заметить, что если внутри фигурных скобок записан *Bareword*, то это не является символической ссылкой. Когда в хэш передается *Bareword*, он его автоматически заключает в кавычки. Есть некоторые встроенные функции, которые можно явно называть, например, функция *shift*. Если нужно явно сказать, что внутри фигурных скобок стоит *expression*, то можно либо добавить унарный оператор +, либо вызвать функцию со скобками.

2.3. Интерполяция

Внутри двойных кавычек интерполируются переменные, то есть можно непосредственно в строке использовать значение другой переменной. В строках интерполируются $... и @.... Внутри кавычек также можно использовать списковый контекст ($), срезы и хэши:

```
$var = "one";
@ary = (3, "four", 5);
%hash = (k => "v", x => "y");
say "new $var";       # new one
say 'new $var';       # new $var

$" = ';'; # $LIST_SEPARATOR
say "new @ary";       # new 3;four;5
say 'new @ary';       # new @ary
say "1st: $ary[0]";   # 1st: 3
say "<@ary[1,2]>";    # <four;5>
say "<${ ary[1] }>"; # <four>
say "<$hash{x}>";     # <y>
```

Здесь x″ — это *list separator*, разделитель.

Так как в интерполяции используются $ или @, при этом ${} или @{}являются операторами разыменования ссылки, а внутри фигурных скобок можно использовать любой *expression*, то можно писать какой-либо исполняемый код внутри ссылок:

```
$var = 100;
say "1+2 = @{[ 1+2 ]}"; # 1+2 = 3
say "\$var/=10 = @{[do{ $var/=10; $var }]}";
        # $var/=10 = 10

say "1+2 = ${\( 1+2 )}";
say "1+2 = ${\do{ 1+2 }}";

say "1+2 = ${{key=> 1+2 }}{key}";
say "\$var = ${{key=> do{ $var } }}{key}";

say "Now: ${\scalar localtime}";
        # Now: Wed Sep 30 19:25:48 2015
```

Из примера видно, что здесь возможно использовать не только списочный конструктор, но и скалярный.

3. Функции

3.1. Декларация, аргументы

Функции в общем виде записываются следующим образом:

```
sub NAME;
sub NAME(PROTO);

sub NAME BLOCK
sub NAME(PROTO) BLOCK
```

Можно заранее продекларировать функцию с определенным именем или с некоторым прототипом (без тела). Анонимные функции конструируются без объявления имени:

```
$sub = sub BLOCK;
$sub = sub (PROTO) BLOCK;
```

Такие функции всегда возвращают свое значение, и его можно присвоить в какую-либо переменную.

Пример объявления функции:

```
sub mysub;
...
```

```
sub mysub {
        @_;              # <- args here
        my $a = shift;   # one arg
        my ($a,$b) = @_; # 2 args
        my %h = @_;      # kor k/v
        say "my arg: ",$_[0];

        return unless defined wantarray;
        return (1,2,3) if wantarray; # return list
        1; # implicit ret, last statement
}
```

Все аргументы функции размещаются в спецпеременной @_. Функции работы с массивами по умолчанию работают с таким массивом. В примере также показаны две наиболее часто используемые формы взятия аргументов функции. Если пишется компактная функция, которая должна быстро работать (например, она часто используется в программе), то можно обратиться к конкретному элементу из аргументов напрямую.

3.2. Контекст

Функции «чувствуют» контекст, в котором их вызывают. Для вызова функции существует три контекста. *Скалярный контекст* означает, что результат функции присваивается в скалярную переменную (либо с функцией выполняется какое-то скалярное действие):

```
# scalar context
my $var = mysub(1, 2, $var);
say 10 + mysub();
```

В *списковом контексте* результат функции либо помещается в массив, либо присваивается какому-то списку:

```
# list context
@a = mysub();
($x,$y) = mysub();
```

Void-контекст означает, что функция просто вызывается, а её результат не представляет интереса:

```
#void context
mysub();
```

Контекст проверяется при помощи специального вызова *wantarray*:

```
sub mysub;
...
sub mysub {
@_;              # <- args here
my $a = shift;   # one arg
my ($a,$b) = @_; # 2 args
my %h = @_;      # kor k/v
say "my arg: ",$_[0];

return unless defined wantarray;
return (1,2,3) if wantarray; # return list
1; # implicit ret, last statement
}
```

При void-вызове *wantarray* будет *undefined*. Если контекст списковый, то *wantarray* будет *true*, если скалярный — *false*.

В теле функции можно написать *return*. Можно сделать *return* без каких-либо результатов, можно вернуть один элемент, можно вернуть список. Любой последний *statement* в теле функции является автоматически возвращаемым значением.

Самый простой способ вызова функции — дописать к ней круглые скобки и передать аргументы:

```
mysub(...);
mysub ...;
```

Если функция была объявлена раньше, то круглые скобки можно опустить. Существует «старый» стиль записи — с использованием & и круглых скобок:

```
&mysub(  );
```

Такой синтаксис вызова применялся в Perl четвертой версии. Самостоятельно использовать данный стиль записи не рекомендуется. Вызов функции с текущими аргументами имеет вид:

```
&mysub;      #  &mysub( @_ );
```

Вызвать ссылку на функцию можно несколькими способами:

```
$sub->(...);
&$sub(...);
&$sub;       #  &$sub( @_ );
```

3.3. Прототипы

У функции может быть объявлен *прототип* (перечисление того, что функция хочет на вход). Например, можно объявить, что функция принимает ровно два аргумента, и тогда при попытке вызвать её с другим количеством аргументов программа не скомпилируется. Можно также сказать, что функция хочет первым аргументом скаляр, список, файловый дескриптор, указатель на функцию или разделитель:

```
$ # scalar
@ # list
% # list
* # filehandle
& # special codeblock
; # optional separator
```

Существуют и более расширенные прототипы:

```
_ # scalar or $_
+ # hash or array (or ref to)
\ # force type
\[%@] # ex: real hash or array

sub vararg($$;$); # 2 req, 1 opt
sub vararg($$;@); # 2 req, 0..* opt
sub noarg(); # no arguments at all
```

Если прототип пустой, то функция не принимает никаких аргументов.

Специальный прототип &, находясь в первой позиции, позволяет употреблять подобный синтаксис:

```
sub check (&@) {
        my ($code, @args) = @_;
        for (@args) {
                $code->($_);
        }
}

check {
        if( $_[0] > 10 ) {
                die "$_[0] is too big";
        }
} 1, 2, 3, 12;
```

В данном случае функция таким образом проверяет, что все аргументы на вход меньше десяти.

3.4. Встроенные функции:grep, map, sort

Есть встроенные функции, которые ведут себя подобным образом — *grep*, *map* и *sort*.

Функция *grep* принимает на вход список, для каждого элемента которого устанавливает значение в $_ и вызывает переданный блок. Блок должен вернуть *true* или *false*. Если блок возвращает *true*, элемент пропускается в выход. Таким образом, данная функция является своеобразным фильтром:

```
@nonempty = grep { length $_; } @strings;
$count    = grep { length $_; } @strings;
@nonempty = grep length($_), @strings;

@odd  = grep {     $_ % 2 } 1..100;
@even = grep { not $_ % 2 } 1..100;

%uniq = ();
@unique = grep { !$uniq{$_}++ } @with_dups;}

@a = 1..55;
@b = 45..100;
%chk; @chk{@a} = ();
@merge = grep { exists $chk{$_} } @b;
```

Здесь представлен также упрощенный синтаксис *grep*, работающий без фигурных скобок.

Функция *map* позволяет взять входной список и преобразовать его в какой-то другой список. Для каждого элемента списка *map* вызывает блок, а то, что возвращает блок, помещается в исходящий список. Данная функция тоже может принимать не обязательно блок, можно написать вместо него *expression*:

```
@squares = map { $_**2 } 1..5; # 1,4,9,16,25

say map chr($_), 32..127;

@nums = 1..100;
@sqrs = map {
        if( int(sqrt($_)) == sqrt($_) ) {
                $_
        } else { () }
} @nums;

my @reduced =
        map $_->[1],
        grep { int($_->[1]) == $_->[1] }
        map { [$_,sqrt $_] } 1..1000;
```

Функция *map* может вернуть пустой список (*map* может быть использован как частный случай *grep*).

Функция *sort* сортирует входящий список:

```
@alphabetically = sort @strings;
@nums = sort { $a <=> $b } @numbers;
@reverse = sort { $b <=> $a } @numbers;
@ci = sort { fc($a) cmp fc($b) } @strings;

sub smart {$a<=>$b || fc($a) cmp fc($b) }
@sorted = sort smart @strings;

my @byval = sort { $h{$a} cmp $h{$b} } keys %h;
```

Синтаксически *sort* может либо просто принять список (тогда отсортирует лексикографически), либо принять блок компаратора, либо принять имя функции, в которой реализован компаратор. Компаратор должен вернуть одно из трех значений: $-1, 0, 1$. В зависимости от этого сортируется исходящий список.

В языке Perl есть некоторое количество функций, которые ведут себя по-разному в зависимости от того, какие аргументы им передаются. Одной из таких функций является *eval*. Пример простейшего *try-catch*:

```
eval "syntax:invalid";
warn $@ if $@;

eval { $a/$b; };
warn $@ if $@;

eval { die "Not root" if $<; };
warn $@ if $@;

eval {      # try
        ...;
1} or do {  # catch
        warn "Error: $@";
};
```

С обычным скалярным аргументом функция *eval* берет то, что написано в этом аргументе, и исполняет это как код Perl. Если у *eval* не получится выполнить этот код, то он установит специальную переменную $@ с текстом ошибки.

Возможен блочный вызов для функции *eval*. Внутри него находится код, уже прошедший синтаксическую проверку, то есть происходит перехват исключений. Исключения можно выбросить при помощи ключевого слова *die*. За написание предупреждений отвечает *warn* (пишет, в какой строке произошло).

Важными встроенными функциями являются *chop* и *chomp*. Функция *chop* отрубает с конца один символ, а функция *chomp* отрезает с конца строки то, что содержится в разделителе входного потока:

```
$/ = "\r\n";
$a = $b = "test\r\n";
chop($a),chop($a),chop($a); # \n,\r,t
say $a;
chomp($b),chomp($b) # \r\n, '';
say $b;
```

Для работы со строками есть стандартный набор из Си (*index*, *rindex*, *substr*, *lenght*):

```
#            \downarrow-----index($_," ") # 4
$_  =   "some average string\n";
#           └──┘     ↑---rindex($_," ") # 12
#         substr($_,3,5) = "e ave"
```

Существуют также функции приведения кейса *lc*, *lcfirst*, *uc*, *ucfirst*, *fc*:

```
$big = "WORD"; $small = "word";
say lc $big;          # word "\L"
say lcfirst $big;     # wORD "\l"
say uc $small;        # WORD "\U"
say ucfirst $small;   # Word "\u"

say "equal" if
fc $big eq fc $small; # v5.16+

say "\u\LnAmE\E"; # Name
```

Функции *chr*, *ord*, *hex*, *oct* используются для преобразования символа в код и кода в символ:

```
use utf8; use open qw(:utf8 :std);
say chr(80);          # P
say ord("P");         # 80
```

```
say ord(chr(-1));     # 65533, BOM \x{fffd}
say ord("±");         # 177
say chr(177);         # ±
say chr(9786);        #
say ord "ë";          # 1105
say hex "dead_beaf"; # 3735928495
say hex "0xDEAD";     # 57005
say oct "04751";      # 2537
```

Функция *reverse* имеет двойное поведение. В скалярном контексте *reverse* переворачивает тот скаляр, который ему дали, а в списковом контексте он переворачивает список целиком, не изменяя его элементы. На эту тему имеется шутка:

```
# Why
say reverse 'dog';
# prints dog,
# but
say ucfirst reverse 'dog';
# prints God?
```

Примеры разных форматов для функции *sprintf*, знакомые многим по Си:

```
$a = sprintf"%c %s %d %u\n%o %x %e %f %g",
9786, "str", -42, -1, 2537,
57005, 1/9, 1/3, .6626E-33;
say $a;
#  str -42 18446744073709551615
# 4751 dead 1.111111e-01 0.333333 6.626e-34
```

3.5. eval

В языке Perl есть многие необходимые математические функции (*abs*, *int*, *srand*, *rand*, *log*, *exp*, *sin*, *cos*, *atan2*, *sqrt*). Они ведут себя практически так же, как и соответствующие операторы из Си. Следующий пример показывает, почему нужно смотреть на класс точности *double*:

```
while (<>) {
        say int( log ( abs $_ ) / log 10 );
}

printf "%g\n", log ( 1e6 ) / log( 10 ) - 6 ;
# -8.88178e-16
```

Для работы с хэшами используются функции *keys*, *values* и итератор *each*:

```
%h = map { $_ => -$_ } 1..3;
@a = keys %h; @b = values %h;
while (my ($k,$v) = each @a)
        { say "$k: $v ($b[$k])"; }
while (my ($k,$v) = each %h)
        { say "$k: $v"; }
```

Итератор *each* последовательно возвращает попарно элементы данного хэша или массива. Это способ обойти весь массив или хэш.

Для работы с массивами используются функции *push*, *pop*, *shift*, *unshift*, *splice*:

push(@a,$x,$y)	splice(@a,@a,0,$x,$y)
pop(@a)	splice(@a,-1)
shift(@a)	splice(@a,0,1)
unshift(@a,$x,$y)	splice(@a,0,0,$x,$y)
$a[$i] = $y	splice(@a,$i,1,$y)

По умолчанию все эти функции работают с массивом аргументов функции @_. Функция *splice* позволяет любым образом нарезать массив:

```
@a = ( 1, 2, 3, 4, 5, 6, 7 );
#|          └─────┘
#|          └─┐   |        ┌ replacement
#└──────────┐ |   |   ┌────┴───┐
splice( @a, 1, 3, ( 8, 9 ) );
say @a;
# 1, 8, 9, 5, 6, 7
```

Для работы со временем есть функции *gmtime*, *localtime*, *time* и *strftime*:

```
say time;          # 1443632440
say ~~localtime;   # Wed Sep 30 20:00:40 2015
say ~~localtime 0;# Thu Jan  1 03:00:00 1970
say ~~gmtime 0;    # Thu Jan  1 00:00:00 1970

($s,$m,$h,$D,$M,$Y,$Wd,$Yd,$dst) =
localtime( time+86400 );
printf "%04u-%02u-%02uT%02u:%02u:%02u",
$Y+1900, $M+1, $D, $h, $m, $s;
printf "Day no: %u, Weekday: %u", $Yd, $Wd;

# 2015-10-01T20:00:40
# Day no: 273, Weekday: 4

use POSIX 'strftime';
say strftime "%r",localtime(); # 08:00:40 PM
```

Perl обладает стеком, для работы с которым используются функции *caller* и *goto*:

```
sub test1 {
        my $i=0;
        while ( ($pk, $f, $l,$s) = caller($i++)) {
                say "$i. from $f:$l ($s)";
        }
}
sub test2 {
        test1()
};
sub test3 {
        test2();
}
sub test4 { goto &test2; }
test3();
test4();
```

При работе с функциями при необходимости следует изучать документацию, не обязательно помнить все особенности функций наизусть.

4. Операторы

4.1. Порядок исполнения

В отличие от большинства динамических языков, Perl смотрит не на тип операнда, а на тип оператора. В Perl есть отдельно числовые операторы и строковые. Если на вход числового оператора приходят строки,

они будут преобразованы в числа (и наоборот). Ассоциативность и приоритет арифметических операторов соответствуют тому, как это принято в математике.

Здесь перечислены все операторы, которые есть в языке Perl:

Ассоциативность	Оператор
left	TERM и LIST
left	`->`
n/a	`++`, `--`
right	`**`
right	`!`, `~`, `\` unary `+`, `-`,
left	`=~` `!~`
left	`*`, `/`, `%`, `x`
left	`+`, `-`, `.`
left	`<<`, `>>`
n/a	named unary ops
n/a	`<`, `>`, `<=`, `>=`, `lt`, `gt`, `le`, `ge`
n/a	`==`, `!=`, `<=>`, `eq`, `ne`, `cmp`, `~~`
...	...

Ассоциативность	Оператор
...	...
left	`&`
left	`|`, `^`
left	`&&`
left	`||`, `//`
n/a	`..`, `...`
right	`?:`
right	`=`, `+=`, `-=`, `*=` и т.д.
left	`,`, `=>`
n/a	LIST
right	`not`
left	`and`
left	`or`, `xor`

Операторы перечислены в порядке приоритета и с указанием ассоциативности.

4.2. Операторы TERM и LIST

Самыми высокоприоритетными оператором являются *TERM* и *LIST (L)*:

Чтобы понять, как именно Perl разбирается с таким количеством операторов, можно рассмотреть следующий пример:

```
my $v = 5;
my @a = ( 1,2,sort 3,4+$v,6x2,7 );
```

С первого взгляда непонятно, в какой последовательности будут выполнены действия. Бывает уместно воспользоваться модулем *Deparse* — он расставит скобки:

```
(my $v = 5);
(
        my @a = (
                1, 2,
                sort(
                        3,
                        (4 + $v),
                        (6 x 2),
                        7
                )
        )
);
```

Можно рассмотреть действия по шагам:

```
(
        my @a = (       #12
        1,                # 1
        2,                # 2
        sort(     #11
                        3,      #3
```

```
                        (
                                4       #4
                                  +       #6
                                $v      #5
                        ),
                        (
                                6       #7
                                  x       #9
                                2       #8
                        ),
                        7       #10
                )       #11
        )       #12
);
```

4.3. Оператор «стрелочка»

Следующий оператор по приоритету — «стрелочка» (→, **суффиксный оператор разыменования**). Зачастую он относится к идентификатору переменной. У него есть еще два вызова: вызов метода на объекте или классе и вызов метода по содержимому переменной:

```
STMT->{...} # STMT должен вернуть HASHREF

STMT->[...] # STMT должен вернуть ARRAYREF

STMT->(...) # STMT должен вернуть CODEREF

STMT->method(...)
# STMT должен вернуть объект или класс
STMT->$var(...)
# $var должен вернуть имя метода или CODEREF
```

4.4. Операторы инкремента и декремента

Далее по приоритету — **операторы инкремента** и **декремента** (аналогичны соответствующим в Си):

```
my $i = 0;
$x = $i++; # $x = 0; $i = 1;
$y = ++$i; # $y = 2; $i = 2;
```

Здесь нужно отметить, что $++\$i+\$i++$ является неопределённым поведением. Если в качестве аргумента этим операторам передается *undef*, то он всегда работает как число 0.

В языке Perl оператор ++ наделен «магическим» поведением. Данный оператор работает не только над числами, но и над строками. Если переменная, попадающая в оператор инкремента, является строкой, начинается на большие или маленькие буквы и содержит в себе буквы или цифры, то она используется как число, знаковым набором которого являются соответствующие диапазоны:

```
say ++($a = "a");   #    b
say ++($a = "aa");  #   ab
say ++($a = "AA");  #   AB
say ++($a = "Aa1"); #  Aa2
say ++($a = "Aa9"); #  Ab0
say ++($a = "Az9"); #  Ba0
say ++($a = "Zz9"); # AAa0
say ++($a = "zZ9"); # aaA0
```

При этом декремент «магическим» не является.

4.5. Унарные операторы

В Perl существует привычный по Си **оператор логического отрицания** (!). В данном языке *false* — это 0, пустая строка, *undef* и *overloaded object*. *True* — все остальное, по умолчанию 1:

```
!0     #  1
!1     # ""
!""    #  1
!undef #  1
```

Унарный оператор − (**математическое отрицание**) на числах меняет знак числа. В Perl у него также определено поведение на строках и на *Bareword*:

```
-0       #  0
-1       # -1
- -1     #  1
-"123"   # -123
-"-123"  #  123
-undef   #  0 or -0
-bare    # "-bare"
-"word"  # "-word"
-"+word" # "-word"
-"-word" # "+word"
-"-0"    #  0 or +0
```

Унарный **оператор битовой инверсии** (∼) работает как на числах, так и на строках, демонстрируя двойственное поведение:

```
# numbers
~0   # 0xffff_ffff or 0xffff_ffff_ffff_ffff
0777 & ~027 # 0750 (in oct)

# byte string
~"test"    # "\213\232\214\213"
# chr(~ord("t")).chr(~ord("e")).
# chr(~ord("s")).chr(~ord("t"));
# char string
use utf8;
"ë" #  "\x{451}"
~"ë" # "\x{fffffbae}" 32b
~"ë" # "\x{ffffffffffffbae}" 64b
```

Унарный оператор + не имеет собственного эффекта и используется как разделитель, когда нужно понизить приоритет:

```
say +( 1 + 2 ) * 3; # 9
say  ( 1 + 2 ) * 3; # 3
```

Иногда + используется как обозначение, что далее следует конструктор анонимного хэша:

```
return +{}; # empty anon hash
map { +{ $_ => -$_} } @_;
```

В языке Perl есть оператор, редко встречающийся в других языках — **оператор применения регулярного выражения**, позитивный и негативный (=∼, ! ∼):

```
# match
$var =~ /regexp/;
$var !~ /shoudn't/;
```

```
# replace
$var =~ s/word/bare/g;

# transliterate
$var =~ tr/A-Z/a-z/;

if ($var = ~ /match/) {...} # always true
#          ^
#          '-------- beware of space
if ($var = (~(/match/))) {...}
```

Необходимо заметить, что пробел между = и ∼ не должен ставиться, иначе один оператор превратится в два.

4.6. Числа и строки

Существуют также привычные **операторы арифметики** (*, /, %). Они работают на числах и приводят все к числам:

```
say 9*7; # 63
say 9/7; # 1.28571428571429
say 9%7; # 2
```

Нестандартный оператор × — это **оператор повтора**, работающий либо на скалярах, либо на списках:

```
say 9x7; # 9999999
say join ",",(9,10)x3; # 9,10,9,10,9,10
```

Бинарные операторы сложения, вычитания и конкатенации (+, −, .) работают следующим образом:

```
say 1+2;          # 3
say "1"+"2";      # 3
say "1z" + "2z"; # 3
say "a" + "b";    # 0

say 1.2;          # 1.2
say 1 . 2;        # 12
say "1"."2";      # 12
say "a"."b";      # ab
```

Необходимо быть аккуратнее при использовании конкатенации — если слева и справа от точки стоят числа, то это число с десятичной частью, в случае конкатенации нужно отделять числа от точки пробелами.

Оператор сдвига (≪, *gg*) реализован полностью с использованием Си, и его поведение аналогично:

```
say 20 << 20;  # 20971520
say 20 << 40;  # 5120 on 32-bit
        # 21990232555520 on 64-bit
use bigint;
say 20 << 80;  # 2417851639229258349412352O
```

Операторы сравнения разделены на арифметические (<, >, <=, >=) и строковые (*lt*, *gt*, *le*, *ge*):

```
say "a" > "b";    # "", 0 > 0
say "a" < "b";    # "", 0 < 0

say 100 gt 20;    # "", "100" gt "20"
say "100" > "20"; # 1
```

Аналогично **операторы равенства** ==, != и <=> работают исключительно с числами, а строковые операторы *eq*, *ne*, *cmp* работают только со строками:

```
say 10 == "10";   #  1
say "20" != "10"; #  1
say 1 <=> 2;      # -1
say 1 <=> 1;      #  0
say 2 <=> 1;      #  1
say "a" <=> "b";  #  0
say "a" == "b";   #  1

say 1 eq "1";     #  1
say "0" ne 0;     # ""
say "a" cmp "b";  # -1
say "b" cmp "a";  #  1
```

Так как изначально в язык не заложено таких понятий как *NaN*, они здесь реализованы строками. Понять, поддерживается ли данная платформа, можно следующим образом:

```
say "No NaN" if "NaN" == "NaN";
```

В Perl версии 5.10.1 внедрили *smartmatch operator* — **оператор умного сравнения** (∼∼). Этот оператор «вшит» в *given/when*, но его можно использовать и вручную:

```
my @ary = (1,2,undef,7);
say "sparse" if undef ~~ @ary;

given ($num) {
when ([1,2,3]) { # as $num ~~ [1,2,3]
say "1..3";
}
when ([4..7]) {  # as $num ~~ [4..7]
say "4..7";
}
}
```

Однако, этот оператор оказался настолько сложным в использовании, что в Perl версии 5.18 и выше его снова признали экспериментальным.

Битовые операторы *and*, *or* и *xor* (&, |,^) работают раздельно над числами и над строками:

```
$x = int(rand(2**31-1));
say $x & ~$x + 1;
say $x ^ ( $x & ($x - 1));

$x = $x ^ $y;
$y = $y ^ $x;
$x = $x ^ $y;

say "test" ^ "^^^^"; # *;-*
say "test" & "^^^^"; # TDRT
```

Если операнды смешанные, то берется тип левого операнда, и второй приводится к нему.

Существуют также **C-style логические операторы** *and*, *or*, *defined-or* (&&, ||, //). Они выполняются последовательно и передают контекст:

```
say 1 && "test"; # test
say 0 || "test"; # test
say 1 || die;    # 1   # say( 1 || die );
say 0 && die;    # 0   # say( 0 && die );

$a = $x // $y;
$a = defined $x ? $x : $y;
```

Оператор *defined-or* находится в данной категории по смыслу, но у него нет аналогов в других языках. Данный оператор проверяет переменную не на «*true/false*», а на то, что она *defined*. У этих операторов очень высокий приоритет по сравнению с обычным списком, в связи с этим могут возникать определенные трудности.

4.7. Операторы диапазона

В Perl есть **операторы диапазона**, *range operators* (.., ...). Эти операторы работают по-разному в разных контекстах. В списковом контексте они просто возвращают соответствующий список, как показано в следующем примере:

```
@a = 1..10;

for ("a".."z") {
say $_;
}

say "A"..."Z";

@b = @a[3..7];
```

Если же оператор диапазона применен в скалярном (логическом) контексте, то он проверяет переменную $. на соответствие граничным значениям. Пример с константным операндом:

```
for $. (1..5) { # $. - $INPUT_LINE_NUMBER
say "$. : ".(2..4);
}
```

```
1 :
2 : 1       # became true ($. == 2)
3 : 2       # stay true, while $. != 4
4 : 3E0     # ret true, $. == 5, became false
5 :
```

В качестве операнда в данном контексте может быть использовано выражение:

```
for $. (1..5) { print "$.";
        do {
                print "\tL,ret ".($.==2?"T":"F");
                $. == 2;
        } `..` do {
                print "\tR,ret ".($.==4?"T":"F");
                $. == 4;
        };
        say " : ".scalar(2..4);
}
```

Можно расписать действия оператора по шагам:

```
1   L,ret F :
2   L,ret T R,ret F : 1
3   R,ret F : 2
4   R,ret T : 3E0
5   L,ret F :
```

Отличия между оператором .. и оператором ... состоит в том, что на втором шаге (когда становится истинным левое выражение) второй оператор не пытается проверять правое выражение. В основном операторы диапазона используются в списковом контексте.

4.8. Тернарный оператор

Тернарный оператор (? :) в Perl аналогичен оператору из Си (за исключением того, что в него можно присваивать):

```
$a = $ok ? $b : $c;
@a = $ok ? @b : @c;

($a_or_b ? $a : $b) = $c;
```

4.9. Операторы присваивания

Операторы присваивания работают относительно всех операторов, которые есть в языке Perl:

- += -=
- *= /= %= **=
- &= |= x= «= »= ^=
- &&= ||= //=

Данные операторы того же приоритета, что и = (не разбиваются по приоритетам относительно исходных операторов).

4.10. Оператор запятая

Оператор запятая делится на обычную и «жирную» (,, ⇒). За исключением работы с *Bareword*, эти операторы работает одинаково:

```
$a = do { say "one"; 3 }, do { say "two"; 7};
# $a = 7. 3 thrown away

@list = (bareword => STMT);
# forces "" on left
@list = ("bareword", STMT);
```

В скалярном контексте данный оператор исполняет свой левый операнд, затем выбрасывает это значение, исполняет правый и возвращает его значение. В списковом контексте этот оператор возвращает тот список, который получается.

В Perl функция с пустым прототипом и с константным значением является константой. Если написать константу слева от «жирной» запятой, то она сделает из константы строку, что является важной особенностью этого оператора:

```
use constant CONST => "some";
%hash = ( CONST   => "val"); # "CONST"
%hash = (+CONST   => "val"); # "CONST"
%hash = ( CONST() => "val"); # "some"
%hash = (&CONST   => "val"); # "some"
```

4.11. Низкоприоритетные логические операторы

Низкоприоритетные логические операторы *and*, *or*, *xor* и *not* являются операторами с самым низким приоритетом из возможных:

```
open $file, "<", "0" || die "Can't";
open $file, "<", ( "0"||die "Can't" );
open $file, "<", "0" or die "Can't";
open ( $file, "<", "0" ) or die "Can't";

do_one() and do_two() or do_another();

@info = stat($file) || say "error";
#                    ^-cast scalar context
```

```
@info = stat($file) or say "error";
#                    ^-keep list context
```

Такие операторы используются, например, для проверки возвращаемого значения функций.

4.12. Оператор кавычки

Оператор кавычки — *q*, *qq*, *qw*, *qx*, *qr*, *s*, *y*, *tr*, довольно однотипные. Оператор *q* — это строка без интерполяции:

```
say 'string';
say q{string};
say q/string/;
say q;string;;
say q{str{i}ng}; # balanced
say q qtestq;
say q{str{ing};  # not ok, unbalanced }
```

По сути *q* аналогичен одинарным кавычкам. Здесь разделителем может быть любой символ (в примере — *q*), кроме пробельного символа.

Оператор *qq* — строка с интерполяцией (двойные кавычки):

```
say "perl $^V";
say qq{perl $^V};
say qq/perl $^V/;
say qq;perl $^V;;
say qq{perl $^V};
```

Оператор *qw* — генератор списка (без интерполяции):

```
$, = ', ';

say qw(a b c);
# say split / /, 'a b c';

for (qw(/usr /var)) {
        say stat $_;
}
```

Все, что записано внутри оператора *qw* и разделено по пробельным символам, будет возвращено в виде одного списка. Данный оператор часто используется при вызове внешних модулей. В качестве разделителя снова может использоваться любой символ, чаще всего используются / либо круглые скобки.

Оператор *qx* позволяет выполнить внешнюю команду (обратные кавычки):

```
say qx{uname -a};

say qx'echo $HOME';
```

Данный оператор является интерполируемым, но если в качестве разделителя поставить одинарные кавычки, то он перестанет быть таковым.

Оператор *qr* — сборка регулярного выражения:

```
$re = qr/\d+/;
if ( $a =~ m[test${re}] ) { ... }
$b =~ s{search}[replace];
y/A-Z/a-z/; # on $_
```

Здесь / ... / или *m* — это сопоставление (*match*), *s* — поиск и замена (*replace*), *y* или *tr* — транслитерация.

Помимо оператора кавычек для сборки строк существует *Here-doc* (угловые скобки с последующим идентификатором):

```
say <<EOD;
Content of document
EOD

say(<<'THIS', "but", <<THAT);
No $interpolation
THIS
For $ENV{HOME}
THAT
```

Если взять идентификатор в одинарные кавычки, здесь тоже не будет интерполяции.

На этом завершается описание синтаксиса языка Perl.

5. Домашнее задание

Основной список документации:

- perlsyn — Perl syntax
- perldata — Perl data types
- perlref — Perl references and nested data structures
- perllol — Manipulating Arrays of Arrays in Perl
- perlsub — Perl subroutines
- perlfunc — Perl builtin functions
- perlop — Perl operators and precedence
- perlglossary — Perl Glossary

Домашнее задание: написать калькулятор. Поставлены две задачи: синтаксический разбор и выполнение арифметики, преобразование в обратную польскую нотацию.

`( )`	приоритет 0
`-`	унарный минус, приоритет 1
`**` или `^`	возведение в степень, приоритет 2
`*`, `/`	умножение, приоритет 3
`+`, `-`	сложение, приоритет 4

Таблица с арифметическими операторами, которые нужно реализовать.

Примеры выражений, которые калькулятор должен уметь обрабатывать:

```
# входное выражение:
- 16 + 2 * 0.3e+2 - .5 ^ ( 2 - 3 )
# с расставленными скобками
( -16 ) + ( 2 * ( 30.0 ) ) - ( 0.5 ^ ( 2 - 3 ) )
# значение на выходе
42
# обратная польская нотация
-16 2 30 * + 0.5 2 3 - ^ -
```

6. Дополнения (Bonus tracks)

Запись однострочников в файл производится следующим образом:

sample.pl

```
#!/usr/bin/env perl -pl00730012
```

```
        perl -MO=Deparse sample.pl

BEGIN { $/ = "\n"; $\ = ";"; }
LINE: while (defined($_ = <ARGV>)) {
        chomp $_;
}
continue {
        die "-p destination: $!\n" unless print $_;
}
sample.pl syntax OK
```

При добавке внутреннего блока в постфиксные циклы работают *next* и *redo*, но не работает *last*:

```
do {{
        next if $cond1;
        redo if $cond2;
        ...
}} while ( EXPR );
```

Для того, чтобы работал *last*, необходимо добавлять внешний блок:

```
LOOP: {
        do {
                next if $cond1;
                redo if $cond2;
                ...
        } while ( EXPR );
}
```

В таком случае *redo* и *next* будут работать не так, как ожидалось (работают только в рамках данного блока). Чтобы и *last*, и *next*, и *redo* работали, можно сделать и внутренний, и внешний блок, главное — не ошибиться с метками (они принимают в качестве перехода метку блока):

```
LOOP: {
        do {{
                next if $cond1;
                redo if $cond2;
                last LOOP if $cond3;
                ...
        }} while ( EXPR );
}
```

Одна из особенностей синтаксиса в Perl 5.20 заключается в том, что была введена *постфиксная нотация для разыменования ссылки*:

```
use feature 'postderef';
no warnings 'experimental::postderef';

$sref->$*;  # same as  ${ $sref }
$aref->@*;  # same as  @{ $aref }
$aref->$#*; # same as $#{ $aref }
$href->%*;  # same as  %{ $href }
$cref->&*;  # same as  &{ $cref }
$gref->**;  # same as  *{ $gref }
```

Еще одна особенность Perl 5.20 — *хэшовый срез* (ключ/значение):

```
%hash = (
        key1 => "value1",
        key2 => "value2",
        key3 => "value3",
        key4 => "value4",
);
#%sub = (
#    key1 => $hash{key1},
#    key3 => $hash{key3},
#);
%sub = %hash{"key1","key3"};
                        ^    ^               ^
                        |    +-------------+--- на хэше
                        +---------------------- хэш-срез
```

Аналогично введен *срез ключ/значение на массиве*:

```
@array = (
"value1",
"value2",
"value3",
"value4",
);
#%sub = (
#    1 => $array[1],
#    3 => $array[3],
#);
 %sub = %array[ 1, 3 ];
#^     ^      ^
#|     +------+--- на массиве
#+----------------- хэш-срез
```

Другая особенность данной версии — *постфиксный срез* (аналогично постфиксным разыменованиям):

```
($a,$b) = $aref->@[ 1,3 ];
($a,$b) = @{ $aref }[ 1,3 ];

($a,$b) = $href->@{ "key1", "key2" };
($a,$b) = @{ $href }{ "key1","key2" };

%sub = $aref->%[ 1,3 ];
%sub = %{ $aref }[1,3];
%sub = (1 => $aref->[1], 3 => $aref->[3]);

%sub = $href->%{ "k1","k3" };
%sub = %{ $href }["k1","k3"];
%sub = (k1 => $href->{k1},
        k3 => $href->{k3});
```

Кроме того, в Perl 5.20 сделаны *сигнатуры функций*:

```
use feature 'signatures';
sub foo ($x, $y) {
return $x**2+$y;
}

sub foo {
die "Too many arguments for subroutine"
```

```
unless @_ <= 2;
die "Too few arguments for subroutine"
unless @_ >= 2;
my $x = $_[0];
my $y = $_[1];
return $x**2 + $y;
}
```

Именованные унарные операторы — это функции, имеющий строго один аргумент (большинство функций в Perl):

```
chdir $foo    || die; # (chdir $foo) || die
chdir($foo)   || die; # (chdir $foo) || die
chdir ($foo)  || die; # (chdir $foo) || die
chdir +($foo) || die; # (chdir $foo) || die

rand 10 * 20;         # rand (10 * 20)
rand(10) * 20;        # (rand 10) * 20
rand (10) * 20;       # (rand 10) * 20
rand +(10) * 20;      # rand (10 * 20)

-e($file).".ext"      # -e( ($file).".ext" )
```

Такие функции имеют очень высокий приоритет. Они трактуются как оператор над одним аргументом.