A Perl 6 trek for Perl 5 pilgrims

Carl Mäsak

May 29, 2009

Table of contents

Perl 6 is still Perl, but different

```
say 'Hello, Perl 5 people!';
```

New operators

```
# bitwise operators
5   +| 3;          # 7          # ternary op
6   +^ 3           # 6          $a == $b ?? 2 * $a !! $b - $a
5   +& 3;          # 1
"b" ~| "d"         # 'f'

# string concatenation
'a' ~ 'b'          # 'ab'

# file tests
if '/etc/passwd' ~~ :e { say "exists" }

# repetition
'a' x 3            # 'aaa'
'a' xx 3           # 'a', 'a', 'a'
```

Equality operators

```
"ab"     eq      "ab"     # True
"1.0"    eq      "1"      # False
"a"      ==      "b"      # True
"1"      ==      1.0      # True
1        ===     1        # True
[1, 2]   ===     [1, 2]   # False
$x = [1, 2];
$x       ===     $x       # True
$x       eqv     $x       # True
[1, 2]   eqv     [1, 2]   # True
1.0      eqv     1        # False

'abc'    ~~      m/a/     # True
1        ~~      0..4     # True
-3       ~~      0..4     # False
```

Deref arrow is a dot nowadays

```
# $obj->method()  # Perl 5

  $obj.method()   # Perl 6

  for @parcels {
      .address;
      .weigh;
      .ship;
      while .shipping {
          .fold;
          .spindle;
          .mutilate;
      }
      .deliver;
  }
```

Product and zipping operators

```
<a b> X 1..3   # ('a', 1), ('a', 2), ('a', 3),
               # ('b', 1), ('b', 2), ('b', 3)

<a b c> Z 1..3 # 'a', 1, 'b', 2, 'c', 3
```

Two dots and three dots

```
my $range = 1..1000;
say $range.max;       # 1000
$range = 1..*;
say $range.max;       # Whatever()<0xcba660>

@fib = 1,1 ... &[+]   # The entire Fibonacci sequence

sub foo() { ... }
```

Foreach loops rolled into operators

```
say [+] 1...5;        # sum
say [*] 1...5;        # product
say [<] @list;        # is @list in strict ascending order?
say [!=] @list;       # are no two consecutive items equal?
say [eq] @list;       # are all items string-equal?
say [max] @list;      # largest element

say @a >>+<< @b;        # element-wise sum
say [+] @a >>*<< @b;    # dot product from linear algebra
```

No parentheses

```
if $percent > 100  {
    say "weird mathematics";
}

for 1..3 {
    # using $_ as loop variable
    say 2 * $_;
}

while $stuff.is_wrong {
    $stuff.try_to_make_right;
}
```

Powerful looping constructs

```
for 1..10 -> $x {
    say $x;
}

for 0..5 -> $even, $odd {
    say "Even: $even \t Odd: $odd";
}

my %h = a => 1, b => 2, c => 3;
for %h.kv -> $key, $value {
    say "$key: $value";
}

for (1..1000).pick(50).kv -> $index, $value {
    say "Number $index is $value";
}
```

Gradual typing

```
my Int $x = 3;
$x = "foo";           # error
say $x.WHAT;          # 'Int'

# check for a type:
if $x ~~ Int {
    say '$x contains an Int'
}

'a string'       # Str
2                # Int
3.14             # Num
(1, 2, 3)        # List
```

Sigil invariance

```
my $five = 5;
print "an interpolating string, just as in perl $five\n";

my @array = 1, 2, 3, 'foo';
my $sum = @array[0] + @array[1];
if $sum > @array[2] {
    say "not executed";
}
my $number_of_elems = @array.elems;     # or +@array
my $last_item = @array[*-1];

my %hash = foo => 1, bar => 2, baz => 3;
say %hash{'bar'};               # 2
say %hash<bar>;                 # this is auto-quoting
```

More no parens, .perl, and .invert

```
my %comments =
    perl6 => <Amazing Revolutionary>,
    perl5 => <Essential Amazing>,
    perl4 => <Classic>,
    perl1 => <Classic>;

my %epithets;
%epithets.push(%comments.invert);

say %comments.perl;
# {"perl6" => ["Amazing", "Revolutionary"],
#"perl5" => ["Essential", "Amazing"], "perl4" =>
# "Classic", "perl1" => "Classic"}
say %epithets.perl;
# {"Amazing" => ["perl6", "perl5"], "Revolutionary"
# => "perl6", "Essential" => "perl5", "Classic" =>
# ["perl4", "perl1"]}
```

Junctions

```
if $dice_sum == 7 | 11 {
    say 'Natural!'
}
elsif $dice_sum == 2 | 3 | 12 {
    say 'Craps!'
}
```

Chained comparisons

```
if 20 < $bob.age < 30 {
    say 'Bob is a twenty-something.';
}

if 0 <= $angle < 2 * pi { ... }

if 0 < all(@coefficients) <= 1 {
    say 'Coefficients are already normalized.';
}
```

Signatures

```
# sub without a signature - perl 5 like
sub print_arguments {
    say "Arguments:";
    for @_ {
        say "\t$_";
    }
}

# Signature with fixed arity and type:
sub distance(Num $x1, Num $y1, Num $x2, Num $y2) {
    return sqrt ($x2-$x1)**2 + ($y2-$y1)**2;
}
say distance(3, 5, 0, 1);
```

Signatures

```
# Default arguments
sub logarithm($num, $base = 2.7183) {
    return log($num) / log($base)
}
say logarithm(4);        # uses default second argument
say logarithm(4, 2);     # explicit second argument

# named arguments

sub doit(:$when, :$what) {
    say "doing $what at $when";
}
doit(what => 'stuff', when => 'once');
doit(:when<noon>, :what('more stuff'));
```

Multisubs

```
multi sub fib (Int $n where 0|1) { return $n }
multi sub fib (Int $n) { return fib($n-1) + fib($n-2) }

# or simply...

multi sub fib (0) { return 0 }
multi sub fib (1) { return 1 }
multi sub fib (Int $n) { return fib($n-1) + fib($n-2) }
```

Operator overloading

```
sub postfix:<!>(Int $x) {
    [*] 1..$x
}

say 5!;          # 120
```

Unicode!

Classes

```
class Foo {
}

class Bar is Foo {
    # class Bar inherits from class Foo
    ...
}
```

Methods

```
class SomeClass {
    # these two methods do nothing but return
    # the invocant (self)
    method foo {
        return self;
    }
    method bar($s: ) {
        return $s;
    }
}

my SomeClass $x .= new;
$x.foo.bar                          # same as $x
```

Submethods

```
class BaseClass {
    method foo { ... }
    submethod bar { ... }
    sub baz { ... }
}

class DerivingClass is BaseClass {
}

my DerivingClass $d .= new;
$d.foo();      # works
$d.bar();      # nope, submethod
$d.baz();      # nope, sub
```

Attributes

```
class SomeClass {
    has $!a;
    has $.b;
    has $.c is rw;

    method do_stuff {
        return $!a + $!b + $!c;
    }
}
my $x = SomeClass.new;
say $x.a;        # ERROR!
say $x.b;        # ok
$x.b = 2;        # ERROR!
$x.c = 3;        # ok
```

Roles

```
role Paintable {
    has $.colour is rw;
    method paint { ... }
}
class Shape {
    method area { ... }
}

class Rectangle is Shape does Paintable {
    has $.width;
    has $.height;
    method area {
        $!width * $!height;
    }
}
```

Enums

```
enum Day <Mon Tue Wed Thu Fri Sat Sun>;
if custom_get_date().Day == Day::Sat | Day::Sun {
    say "Weekend";
}
```

Subset types

```
subset Even of Int where { $_ % 2 == 0 }
# Even can now be used like every other type name

my Even $x = 2;
my Even $y = 3; # type missmatch error
```

Subset types

```
sub foo (Int where { ... } $x) { ... }
# or with the variable at the front:
sub foo ($x of Int where { ... } ) { ... }
```

New regex syntax

```
# These things work the same:
# * Capturing (...)
# * Repetition quantifiers: *, +, and ?
# * Alternatives: |
# * Backslash escape: \
# * Minimal matching suffix: ??, *?, +?

# You have to quote all non-alphanumerics.

# A dot . matches _any_ character.
# ^ and $ match start/end of string.
# Use ^^ and $$ for start/end of line. (/s is gone)
# \n now matches a logical newline.
# \N matches anything except a newline. (/m is gone)
```

Bracket rationalization

```
# (...) delimits capturing groups
# [...] delimits non-capturing groups
# {...} delimits code blocks

# <...> is used for extensible metasyntax:
#   * subrule:
#       / <sign>? <mantissa> <exponent>? /
#   * code assertion:
#       / (\d**1..3) <?{ $0 < 256 }> /
#   * character classes:
#       / <[ a..z _ ]>* /
#       / <-[a..z_]> <-alpha> /
#   * zero-width assertions:
#       <?alpha>     # match null before a letter, don't capt
#       / <?before pattern> /     # lookahead
#       / <?after pattern> /      # lookbehind
```

Repetition

```
. ** 42                    # match exactly 42 times
<item> ** 3..*             # match 3 or more times

<alt> ** '|'               # separator is character
<addend> ** <addop>        # separator is subrule
<item> ** [ \!?'==' ]      # separator is operator
<file>**\h+                # separator is whitespace
```

Backtracking

```
# :         make preceeding atom not backtrack
# ::        fail surrounding group instead of backtracking
# :::       fail current regex     instead of backtracking
# commit    fail entire match      instead of backtracking
# cut       continue, but delete everything up to here
```

Regex, token, rule

```
regex { pattern }    # always {...} as delimiters
rx    / pattern /    # (almost) any chars as delimiters

token ident { [ <alpha> | _ ] \w* }     # same as...
regex ident { [ <alpha>: | _: ]: \w*: } # ...this

rule quux { next cmd '='   <condition> }    # same as...
token quux { <.ws> next <.ws> cmd <.ws> '=' # ...this
             <.ws> <condition> }
```

Grammars

```
grammar XML {
    token TOP   { ^ <xml> $ };
    token xml   { <text> [ <tag> <text> ]* };
    token text {  <-[<>&]>* };
    rule tag   {
        '<'(\w+) <attributes>*
        [
            | '/>'                      # a single tag
            | '>'<xml>'</' $0 '>'       # an opening
                                        # and a closing tag
        ]
    };
    token attributes { \w+ '="' <-["<>]>* '"' };
};
```

Grammar inheritance

```
grammar Letter {
    rule text  { <greet> <body> <close> }
    rule greet { [Hi|Hey|Yo] $<to>=(\S+?) , $$}
    rule body  { <line>+? }
    rule close { Later dude, $<from>=(.+) }
    # etc.
}

grammar FormalLetter is Letter {
    rule greet { Dear $<to>=(\S+?) , $$}
    rule close { Yours sincerely, $<from>=(.+) }
}
```

Actions

```
grammar Integer {
    token TOP {
        | 0b<[01]>+ {*}  #= binary
        | \d+       {*}  #= decimal
    }
}
class Twice {
    multi method TOP($match, $tag) {
        my $text = ~$match;
        $text = :2($text) if $tag eq 'binary'
        make $text;
    }
    multi method TOP($match) {
        make 2 * $match.ast;
    }
}
Integer.parse('21', :action(Twice.new)).ast      # 42
```

And more...

```
given $source_code {
    $parsetree = m:keepall/<Perl::prog>/;
}

grammar LolPerl is STD { ... }

$~MAIN = LolPerl;

# LOL!
```

Bibliography

The material in this talk has been nicked^Wadapted from a number of sources.

 Moritz Lentz' "Perl 5 to Perl 6" tutorial.

 An email from Damian Conway to perl6-language.

 The synopses.

KTHXBAI

```
# Questions?

# ("When will Perl 6 be released?")
```