An Introduction to Perl6

Perl 6 is more than simply a rewrite of Perl 5.

By separating the parsing from the compilation and the runtime, Perl 6 opens the doors for multiple languages to cooperate.

You'll be able to write your program in Perl 6, Perl 5, TCL, Python, or any other language for which there is a parser. Interchangable runtime engines let you interpret your bytecode or convert it to something else (e.g., Java, C, or even back to Perl).

What is in Perl6 ?

- Multimethods, coroutines, continuations, currying, signatures, captures, exceptions
- Powerful yet convenient object-oriented programming, generics, roles
- Functional programming primitives, {lazy and eager} evaluation, junctions, autothreading, hyperoperators (vector operators)
- Definable grammars for {pattern matching and generalized string processing}, with many other powerful extensions
- Advanced introspection and meta-programming
- {Extensible and overridable} {Perl 6 primitives and Perl 6 grammar}, language-level macros
- Module aliasing and versioning, optional static/gradual typing
- Full Unicode processing support
- Garbage collection
- Greatly-improved foreign function interface
- Optional Typing System

Getting Perl6

- Current Perl 6 distribution is Rakudo Star http://rakudo.org
- Since Rakudo is under rapid development, potential developers should download Rakudo directly from github and build from there:

```
git clone git://github.com/rakudo/rakudo.git

cd rakudo

perl Configure.pl --gen-parrot

make

make install
```

Strings

Strings are surrounded by double quotes (in which case they are interpolating), or with single quotes.

Backslash escapes work just like in Perl 5.

However the interpolation rules have changed a bit. The following things interpolate

```
my $scalar = 6;
my @array = 1, 2, 3;
say "Perl $scalar";           # 'Perl 6'
say "An @array[]";            # 'An 1 2 3', a so-called "Zen slice"
say "@array[1]";              # '2'
say "Code: { $scalar * 2 }" # 'Code: 12'
```

Arrays and hashes only interpolate if followed by an index (or a method call that ends in parenthesis, like `"some $obj.method()"`), empty indexes interpolate the whole data structure.

A block in curly braces is executed as code, and the result is interpolated.

Arrays

Arrays variables still begin with the `@` sigil. And they always do, even when accessing stored items, ie. when an index is present.

```
my @a = 5, 1, 2;                # no parens needed anymore
say @a[0];                      # yes, it starts with @
say @a[0, 2];                   # slices also work
```

Lists are constructed with the Comma operator. `1,` is a list, `(1)` isn't.

Since everything is an object, you can call methods on arrays:

```
my @b = @a.sort;
@b.elems;                       # number of items
if @b > 2 { say "yes" }         # still works
@b.end                          # number of last index. Replaces
$#array
my @c = @b.map({$_ * 2 });  # map is also a method, yes
```

There is a short form for the old `qw/../` quoting construct:

```
my @methods = <shift unshift push pop end delete sort map>;
```

Hashes

While Perl 5 hashes are even sized lists when viewed in list context, Perl 6 hashes are lists of pairs in that context.
Pairs are also used for other things, like named arguments for subroutines.
Just like with arrays the sigil stays invariant when you index it. And hashes also have methods that you can call on them.

```
my %drinks =
    France  => 'Wine',
    Bavaria => 'Beer',
    USA     => 'Coke';

say "The people in France love ",  %drinks{'France'};
my @countries = %drinks.keys.sort;
```

Note that when you access hash elements with %hash{...}, the key is not automatically quoted like in Perl 5. So %hash{foo} doesn't access index "foo", but calls the function foo(). The auto quoting isn't gone, it just has a different syntax:

```
say %drinks<Bavaria>;
```

Types

Perl 6 has types.
Everything is an object in some way, and has a type.
Variables can have type constraints, but they don't need to have one.

There are some basic types that you should know about:

```
    'a string'      # Str
    2               # Int
    3.14            # Rat (rational number)
    (1, 2, 3)       # Seq
```

All "normal" built-in types begin with an upper case letter.
All "normal" types inherit from Any, and absolutely everything inherits from Mu.

You can restrict the type of values that a variable can hold by adding the type name to the declaration.

```
    my Numeric $x = 3.4;
    my Int @a = 1, 2, 3;
```

It is an error to try to put a value into a variable that is of a "wrong" type (ie neither the specified type nor a subtype).
A type declaration on an Array applies to its contents, so my Str @s is an array that can only contain strings.

Some types stand for a whole family of more specific types, for example integers (type Int), rationals (typeRat) and floating-point numbers (type Num) conform to the Numeric type.

Introspection

You can learn about the direct type of a thing by calling its .WHAT method.

```
say "foo".WHAT;     # Str()
```

However if you want to check if something is of a specific type, there is a different way, which also takes inheritance into account and is therefore recommended:

```
if $x ~~ Int {
    say 'Variable $x contains an integer';
}
```

Basic Control Structures

Most Perl 5 control structures are quite similar in Perl 6.
The biggest visual difference is that you don't need a pair of parenthesis after if, while, for etc.
In fact you are discouraged from using parenthesis around the conditions. The reason is that any identifier followed immediately (ie. without whitespace) by an opening parenthesis is parsed as a subroutine call, soif($x < 3)</c tries to call a function named if. While a space after the if fixes that, it is safer to just omit the parens.
Branches
if is mostly unchanged, you can still add elsif and else branches.
unless is still there, but no elsebranch is allowed after unless.

```
if $sheep == 0 {
    say "How boring";
} elsif $sheep == 1 {
    say "One lonely sheep";
} else {
    say "A herd, how lovely!";
}
```

You can also use if and unless as a statement modifier, i.e. after a statement:

```
say "you won" if $answer == 42;
```

Loops

You can manipulate loops with next and last just like in Perl 5.
The for-Loop is now only used to iterate over lists. By default the topic variable $_ is used, unless an explicit loop variable is given.

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

The -> $x { ... } thing is called a "pointy block" and is something like an anonymous sub, or a lambda in lisp.
You can also use more than one loop variable:

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

This is also how you iterate over hashes:

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

The C-style for-loop is now called loop (and the only looping construct that requires parenthesis):

```
    loop (my $x = 1; $x < 100; $x = $x**2) {
        say $x;
    }
```

Subroutines & Signatures

Subroutines are declared with the sub key word, and can have a list of formal parameters, just like in C, Java and most other languages. Optionally these parameters can have type constraints.

Parameters are read-only by default. That can be changed with so-called "traits":

```
    sub try-to-reset($bar) {
        $bar = 2;       # forbidden
    }

    my $x = 2;
    sub reset($bar is rw) {
        $bar = 0;       # allowed
    }
    reset($x); say $x;    # 0

    sub quox($bar is copy){
        $bar = 3;
    }
    quox($x); say $x    # still 0
```

Parameters can be made optional by adding a question mark ? after them, or by supplying a default value.

```
    sub foo($x, $y?) {
        if $y.defined {
            say "Second parameter was supplied and defined";
        }
    }

    sub bar($x, $y = 2 * $x) {
        ...
    }
```

Named Parameters

To define a named parameter, you simply put a colon : before the parameter in the signature list:

```
sub area(:$width, :$height) {
    return $width * $height;
}
area(width => 2,  height => 3);
area(height => 3, width => 2 ); # the same
area(:height(3), :width(2));    # the same
```

In these the variable name is also used as the name of the parameter. You can use a different name, though:

```
sub area(:width($w), :height($h)){
    return $w * $h;
}
area(width => 2,  height => 3);
```

Slurpy Parameters

Just because you give your sub a signature doesn't mean you have to know the number of arguments in advance. You can define so-called slurpy parameters (after all the regular ones) which use up any remaining arguments:

```
    sub tail ($first, *@rest){
        say "First: $first";
        say "Rest: @rest[]";
    }
    tail(1, 2, 3, 4);           # "First: 1\nRest: 2 3 4\n"
```

Named slurpy parameters are declared by using an asterisk in front of a hash parameter:

```
    sub order-meal($name, *%extras) {
        say "I'd like some $name, but with a few modifications:";
        say %extras.keys.join(', ');
    }

    order-meal('beef steak', :vegetarian, :well-done);
```

Interpolation

By default arrays aren't interpolated in argument lists, so unlike in Perl 5 you can write something like this:

```
sub a($scalar1, @list, $scalar2){
    say $scalar2;
}

my @list = "foo", "bar";
a(1, @list, 2);                    # 2
```

That also means that by default you can't use a list as an argument list:

```
my @indexes = 1, 4;
say "abc".substr(@indexes)       # doesn't do what you want
```

You can achieve the desired behavior with a prefix |

```
say "abcdefgh".substr(|@indexes) # bcde
```

Multi Subs

You can actually define multiple subs with the same name but with different parameter lists:

```
    multi sub my_substr($str) { ... }                    # 1
    multi sub my_substr($str, $start) { ... }            # 2
    multi sub my_substr($str, $start, $end) { ... }      # 3
    multi sub my_substr($str, $start, $end, $subst) { ... }    # 4
```

Now whenever you call such a sub, the one with the matching parameter list will be chosen.
The multis don't have to differ in the arity (ie number of arguments), they can also differ in the type of the parameters:

```
    multi sub frob(Str $s) { say "Frobbing String $s"  }
    multi sub frob(Int $i) { say "Frobbing Integer $i" }

    frob("x")         # Frobbing String x
    frob(2)           # Frobbing Integer 2
```

Objects and Classes

Perl 6 has an object model that is much more fleshed out than the Perl 5 one.
It has keywords for creating classes, roles, attributes and methods, and has encapsulated private attributes and methods. In fact it's much closer to the Moose Perl 5 module (which was inspired by the Perl 6 object system).

There are two ways to declare classes

```
class ClassName;
# class definition goes here
```

The first one begins with class ClassName; and stretches to the end of the file. In the second one the class name is followed by a block, and all that is inside the blocks is considered to be the class definition.

```
class YourClass {
    # class definition goes here
}
# more classes or other code here
```

Methods

Methods are declared with the method key word.
Inside the method you can use the self keyword to refer to the object on which the method is called (the invocant).

You can also give the invocant a different name by adding a first parameter to the signature list and append a colon : to it.
Public methods can be called with the syntax $object.method if it takes no arguments, and $object.method(@args) or $object.method: @args if it takes arguments.

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

Methods (continued)

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

(The my SomeClass $x .= new is actually a shorthand for my SomeClass $x .= SomeClass.new. It works because the type declaration fills the variable with a "typo object" ofSomeClass, which is an object representing the class.)
Methods can also take additional arguments just like subs.
Private methods can be declared with method !methodname, and called withself!method_name.

```
class Foo {
    method !private($frob) {
        return "Frobbed $frob";
    }

    method public {
        say self!private("foo");
    }
}
```

Private methods can't be called from outside the class.

Attributes

Attributes are declared with the has key word, and have a "twigil", that is a special character after the sigil. For private attributes that's a bang !, for public attributes it's the dot .. Public attributes are just private attributes with a public accessor. So if you want to modify the attribute, you need to use the ! sigil to access the actual attribute, and not the accessor (unless the accessor is marked is rw).

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

        method set_stuff {
            $!a = 1;    # ok, writing to attribute from within the clas
            $!b = 2;    # same
            $.b = 3;    # ERROR, can't write to ro-accessor
        }

        method do_stuff {
            # you can use the private name instead of the public one
            # $!b and $.b are really the same thing
            return $!a + $!b + $!c;
        }
}
my $x = SomeClass.new;
say $x.a;       # ERROR!
say $x.b;       # ok
$x.b = 2;       # ERROR!
$x.c = 3;       # ok
```

Inheritance

Inheritance is done through an is trait.

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

All the usual inheritance rules apply - methods are first looked up on the direct type, and if that fails, on the parent class (recursively). Likewise the type of a child class is conforming to that of a parent class:

```
        class Bar { }
        class Foo is Bar { }
        my Bar $x = Foo.new();   # ok, since Foo ~~ Bar
```

In this example the type of $x is Bar, and it is allowed to assign an object of type Foo to it, because "everyFoo is a Bar".
Classes can inherit from multiple other classes:

```
    class ArrayHash is Hash is Array {
        ...
    }
```

Though multiple inheritance also comes with multiple problems, and people usually advise against it. Roles are often a safer choice.

Roles and Composition

In general the world isn't hierarchical, and thus sometimes it's hard to press everything into an inheritance hierarchy.
Which is one of the reasons why Perl 6 has Roles.
Roles are quite similar to classes, except you can't create objects directly from them, and that composition of multiple roles with the same method names generate conflicts, instead of silently resolving to one of them, like multiple inheritance does.
While classes are intended primarily for type conformance, roles are the primary means for code reuse in Perl 6.

```
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;
    }
}
```

Contexts

When you write something like this

```
    $x = @a
```

in Perl 5, $x contains less information than @a - it contains only the number of items in @a.
To preserve all information, you have to explicitly take a reference: $x = \@a.

In Perl 6 it's the other way round: by default you don't lose anything, the scalar just stores the array. This was made possible by introducing a generic item context (called scalar in Perl 5) and more specialized numeric, integer and string contexts. Void and List context remain unchanged.

You can force contexts with special syntax.

syntax	context
~stuff	String
?stuff	Bool (logical)
+stuff	Numeric
-stuff	Numeric (also negates)
$(stuff)	Generic item context
@(stuff)	List context
%(stuff)	Hash context
@@(stuff)	Slice context

Slice Context

There's also a new context called slice context. It is a list context in which sublists don't interpolate.

```
    @( <a b> Z <c d> )      # <a c b d>
   @@( <a b> Z <c d> )      # (['a', 'c'], ['b', 'd'])
```

This was introduced after the observation that many built in list functions had two versions - one that returned a flat list, and one that returned a list of lists.

You can force slice context with @@(stuff).

Regexes (also called "rules")

Regexes are one of the areas that has been improved and revamped most in Perl 6. We don't call them *regular expressions* anymore because they are even less regular than they are in Perl 5.

There are three large changes and enhancements to the regexes

Syntax clean up
Many small changes make rules easier to write. For example the dot `.` matches any character now, the old semantics (anything but newlines) can be achieved with `\N`.

Modifiers now go at the start of a regex, and non-capturing groups are `[...]`, which are a lot easier to read and write than the old `(?:...)`.

Nested captures and match object
In Perl 5, a regex like this `(a(b))(c)` would put `ab` into `$1`, `b` into `$2` and `c` into `$3` upon successful match. This has changed. Now `$0` (enumeration starts at zero) contains `ab`, and `$0[0]`or `$/[0][0]` contains `b`. `$1` holds `c`. So each nesting level of parenthesis is reflected in a new nesting level in the result match object.

All the match variables are aliases into `$/`, which is the so-called *Match object*, and it actually contains a full match tree.

Named regexes and grammars
You can declare regexes with names just like you can with subs and methods. You can refer to these inside other rules with `<name>`. And you can put multiple regexes into grammars, which are just like classes and support inheritance and composition

Syntax Clean up

Letter characters (ie underscore, digits and all Unicode letters) match literally, and have a special meaning (they are metasyntactic) when escaped with a backslash. For all other characters it's the other way round - they are metasyntactic unless escaped.

literal	metasyntactic
a b 1 2	\a \b \1 \2
* \: \. \?	* : . ?

Not all metasyntactic tokens have a meaning (yet). It is illegal to use those without a defined meaning.

There is another way to escape strings in regexes: with quotes.

```
m/'a literal text: $#@!!'/
```

The changed semantics of . has already been mentioned, and that [...] are now non-capturing groups. Char classes are <[...]>, and negated char classes <-[...]>. ^ and $ always match begin and end of the string, to match begin and end of lines use ^^ and $$.

This means that the /s and /m modifiers are gone. Modifiers are now given at the start of a regex, and are spelled as pairs.

```
if "abc" ~~ m:i/B/ {
    say "Match";
}
```

Modifiers have a short and a long form. The old /x modifier is now the default, i.e. white spaces are ignored

Syntax Clean up (cont'd)

```
short   long              meaning
-----------------------------
:i      :ignorecase       ignore case (formerly /i)
:m      :ignoremark       ignore marks (accents, diaeresis etc.)
:g      :global           match as often as possible (/g)
:s      :sigspace         Every white space in the regex matches
                          (optional) white space
:P5     :Perl5            Fall back to Perl 5 compatible regex syntax
:4x     :x(4)             Match four times (works for other numbers as well)
:3rd    :nth(3)           Third match
:ov     :overlap          Like :g, but also consider overlapping matches
:ex     :exhaustive       Match in all possible ways
        :ratchet          Don't backtrack
```

The `:sigspace` needs a bit more explanation. It replace all whitespace in the pattern with `<.ws>` (that is it calls the rule `ws` without keeping its result). You can override that rule. By default it matches one or more white spaces if it's enclosed in word characters, and zero or more otherwise.

(There are more new modifiers, but probably not as important as the listed ones).

The Match Object

Every match generates a so-called match object, which is stored in the special variable `$/`. It is a versatile thing. In boolean context it returns `Bool::True` if the match succeeded. In string context it returns the matched string, when used as a list it contains the positional captures, and when used as a hash it contains the named captures. The `.from` and `.to` methods contains the first and last string positions of the match.

```
if 'abcdefg' ~~ m/(.(.)) (e | bla ) $<foo> = (.) / {
    say $/[0][0];               # d
    say $/[0];                  # cd
    say $/[1];                  # e
    say $/<foo>                 # f
}
```

`$0`, `$1` etc are just aliases for `$/[0]`, `$/[1]` etc. Likewise `$/<x>` and `$/{'x'}` are aliased to `$<x>`.

Note that anything that you access via `$/[...]` and `$/{...}` is a match object (or a list of Match objects) again. This allows you to build real parse trees with rules.

Named Regexes & Grammars

Regexes can either be used with the old style `m/.../`, or be declared like subs and methods.

```
regex a { ... }
token b { ... }
rule  c { ... }
```

The difference is that `token` implies the `:ratchet` modifier (which means no backtracking, like a `(?> ... )` group around each part of the regex in perl 5), and `rule` implies both `:ratchet` and `:sigspace`.

To call such a rule (we'll call them all rules, independently with which keyword they were declared) you put the name in angle brackets: `<a>`. This implicitly anchors the sub rule to its current position in the string, and stores the result in the match object in `$/<a>`, ie it's a named capture. You can also call a rule without capturing its result by prefixing its name with a dot: `<.a>`.

A grammar is a group of rules, just like a class (see the SYNOPSIS for an example). Grammars can inherit, override rules and so on.

```
grammar URL::HTTP is URL {
    token schema { 'http' }
}
```

Comparing and Matching

```
"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
'abc'    ~~      Str      True
'abc'    ~~      Int      False
Str      ~~      Any      True
Str      ~~      Num      False
1        ~~      0..4     True
-3       ~~      0..4     False
```

Smart Matching

Perl 6 has a "compare everything" operator, called "smart match" operator, and spelled ~~.

For immutable types it is a simple equality comparison. A smart match against a type object checks for type conformance. A smart match against a regex matches the regex. Matching a scalar against a `Range` object checks if that scalar is included in the range.

There are other, more advanced forms of matching: for example you can check if an argument list (`Capture`) fits to the parameter list (`Signature`) of a subroutine, or apply file test operators (like `-e` in Perl 5).

What you should remember is that any "does $x fit to $y?"-Question will be formulated as a smart match in Perl 6.

Containers and Values

Perl 6 distinguishes between containers, and values that can be stored in containers.

A normal scalar variable is a container, and can have some properties like type constraints, access constraints (for example it can be read only), and finally it can be aliased to other containers.

Putting a value into a container is called *assignment*, and aliasing two containers is called `binding`.

```
my @a = 1, 2, 3;
my Int $x = 4;
@a[0] := $x;      # now @a[0] and $x are the same variable
@a[0] = 'Foo';    # Error 'Type check failed'
```

Types like `Int` and `Str` are immutable, ie the objects of these types can't be changed; but you can still change the variables (the containers, that is) which hold these values:

```
my $a = 1;
$a = 2;       # no surprise here
```

Binding can also be done at compile time with the `::=` operator.

You can check if two things are bound together the =:= comparison operator.

Changes to Perl 5 Operators

All the numeric operators (+, -, /, *, **, %) remain unchanged.

Since |, ^ and & now construct junctions, the bit wise operators have a changed syntax. They now contain a context prefix, so for example +| is bit wise OR with numeric context, and ~^ is one's complement on a string. Bit shift operators changed in the same way.

String concatenation is now ~, the dot . is used for method calls.

File tests are now formulated in `Pair` notation, Perl 5 `-e` is now `:e`. If something other than `$_` should be used as the file name, it can be supplied via `$filename.IO ~~ :e`.

The repetition operator `x` is now split into two operators: `x` replicates strings, `xx` lists.

The ternary operator, formerly `$condition ? $true : $false`, is now spelled `$condition ?? $true !! $false`.

Comparison operators can now be chained, so you can write `$a < $b < $c` and it does what you mean.

Laziness

In this case *lazy* means, that the evaluation is delayed as much as possible. When you write something like`@a := map BLOCK, @b`, the block isn't executed at all. Only when you start to access items from `@a`the `map` actually executes the block and fills `@a` as much as needed.

Note the use of binding instead of assignment: Assigning to an array might force eager evaluation (unless the compiler knows the list is going to be infinite; the exact details of figuring this out are still subject to change), binding never does.

Laziness allows you to deal with infinite lists: as long as you don't do anything to all of its arguments, they take up only as much space as the items need that have already been evaluated.

There are pitfalls, though: determining the length of a list or sorting it kills laziness - if the list is infinite, it will likely loop infinitely, or fail if the infiniteness can be detected.

In general all conversions to a scalar (like `List.join`) are *eager*, i.e. non-lazy.

Laziness prevents unnecessary computations, and can therefore boost performance while keeping code simple.

When you read a file line by line in Perl 5, you don't use `for (<HANDLE>)` because it reads all the file into memory, and only then starts iterating. With laziness that's not an issue:

```
my $file = open '/etc/passwd';
for $file.lines -> $line {
    say $line;
}
```

Since `$file.lines` is a lazy list, the lines are only physically read from disk as needed (besides buffering, of course).

gather/take

A very useful construct for creating lazy lists is `gather { take }`. It is used like this:

```
my @list := gather {
    while True {
        # some computations;
        take $result;
    }
}
```

`gather BLOCK` returns a lazy list. When items from `@list` are needed, the `BLOCK` is run until `take` is executed. `take` is just like return, and all `taken` items are used to construct `@list`. When more items from `@list` are needed, the execution of the block is resumed after `take`.

`gather/take` is dynamically scoped, so it is possible to call `take` outside of the lexical scope of the`gather` block:

```
my @list = gather {
    for 1..10 {
        do_some_computation($_);
    }
}

sub do_some_computation($x) {
    take $x * ($x + 1);
}
```

Note that `gather` can act on a single statement instead of a block too:

```
my @list = gather for 1..10 {
    do_some_computation($_);
}
```

gather/take

A very useful construct for creating lazy lists is `gather { take }`. It is used like this:

```
my @list := gather {
    while True {
        # some computations;
        take $result;
    }
}
```

`gather BLOCK` returns a lazy list. When items from `@list` are needed, the `BLOCK` is run until `take` is executed. `take` is just like return, and all `taken` items are used to construct `@list`. When more items from `@list` are needed, the execution of the block is resumed after `take`.

`gather/take` is dynamically scoped, so it is possible to call `take` outside of the lexical scope of the`gather` block:

```
my @list = gather {
    for 1..10 {
        do_some_computation($_);
    }
}

sub do_some_computation($x) {
    take $x * ($x + 1);
}
```

Note that `gather` can act on a single statement instead of a block too:

```
my @list = gather for 1..10 {
    do_some_computation($_);
}
```

Custom Operators

Operators are functions with unusual names, and a few additional properties like precedence and associativity. Perl 6 usually follows the pattern `term infix term`, where `term` can be optionally preceded by prefix operators and followed by postfix or postcircumfix operators.

```
1 + 1                   infix
+1                      prefix
$x++                    postfix
<a b c>                 circumfix
@a[1]                   postcircumfix
```

Operator names are not limited to "special" characters, they can contain anything except whitespace.

The long name of an operator is its type, followed by a colon and a string literal or list of the symbol or symbols, for example `infix:<+>` is the the operator in `1+2`. Another example is `postcircumfix:<[ ]>`, which is the operator in `@a[0]`.

With this knowledge you can already define new operators:

```
multi sub prefix:<€> (Str $x) {
    2 *  $x;
}
say €4;                         # 8
```

Precedence

In an expression like `$a + $b * $c` the `infix:<*>` operator has tighter precedence than `infix:<+>`, which is why the expression is evaluated as `$a + ($b * $c)`.

The precedence of a new operator can be specified in comparison to to existing operators:

```
multi sub infix:<foo> is equiv(&infix:<+>) { ...  }
mutli sub infix:<bar> is tighter(&infix:<+>) { ... }
mutli sub infix:<baz> is looser(&infix:<+>) { ... }
```

Accociativity

Most infix operators take only two arguments. In an expression like `1 / 2 / 4` the *associativity* of the operator decides the order of evaluation. The `infix:</>` operator is left associative, so this expression is parsed as `(1 / 2) / 4`. for a right associative operator like `infix:<**>` (exponentiation) `2 ** 2 ** 4` is parsed as `2 ** (2 ** 4)`.

Perl 6 has more associativities: `none` forbids chaining of operators of the same precedence (for example `2 <=> 3 <=> 4` is forbidden), and `infix:<,>` has `list` associativity. `1, 2, 3` is translated to`infix:<,>(1; 2; 3)`. Finally there's the `chain` associativity: `$a < $b < $c` translates to `($a < $b) && ($b < $c)`.

```
multi sub infix:<foo> is tighter(&infix:<+>)
                      is assoc('left')
                      ($a, $b) {
    ...
}
```

The MAIN subroutine

Calling subs and running a typical Unix program from the command line is visually very similar: you can have positional, optional and named arguments.

You can benefit from it, because Perl 6 can process the command line for you, and turn it into a sub call. Your script is normally executed (at which time it can munge the command line arguments stored in`@*ARGS`), and then the sub `MAIN` is called, if it exists.

If the sub can't be called because the command line arguments don't match the formal parameters of the`MAIN` sub, an automatically generated usage message is printed.

Command line options map to subroutine arguments like this:

```
-name                   :name
-name=value             :name<value>

# remember, <...> is like qw(...)
--hackers=Larry,Damian  :hackers<Larry Damian>

--good_language         :good_language
--good_lang=Perl        :good_lang<Perl>
--bad_lang PHP          :bad_lang<PHP>

+stuff                  :!stuff
+stuff=healty           :stuff<healthy> but False
```

The `$x = $obj but False` means that `$x` is a copy of `$obj`, but gives `Bool::False` in boolean context.

So for simple (and some not quite simple) cases you don't need an external command line processor, but you can just use sub `MAIN` for that.

Twigils

Calling subs and running a typical Unix program from the command line is visually very similar: you can have positional, optional and named arguments.

You can benefit from it, because Perl 6 can process the command line for you, and turn it into a sub call. Your script is normally executed (at which time it can munge the command line arguments stored in`@*ARGS`), and then the sub `MAIN` is called, if it exists.

If the sub can't be called because the command line arguments don't match the formal parameters of the`MAIN` sub, an automatically generated usage message is printed.

Command line options map to subroutine arguments like this:

```
-name                   :name
-name=value             :name<value>

# remember, <...> is like qw(...)
--hackers=Larry,Damian  :hackers<Larry Damian>

--good_language         :good_language
--good_lang=Perl        :good_lang<Perl>
--bad_lang PHP          :bad_lang<PHP>

+stuff                  :!stuff
+stuff=healty           :stuff<healthy> but False
```

The `$x = $obj but False` means that `$x` is a copy of `$obj`, but gives `Bool::False` in boolean context.

So for simple (and some not quite simple) cases you don't need an external command line processor, but you can just use sub `MAIN` for that.

Enums

Enums are versatile beasts. They are low-level classes that consist of an enumeration of constants, typically integers or strings (but can be arbitrary).

These constants can act as subtypes, methods or normal values. They can be attached to an object with the `but` operator, which "mixes" the enum into the value:

```
my $x = $today but Day::Tue;
```

You can also use the type name of the Enum as a function, and supply the value as an argument:

```
$x = $today but Day($weekday);
```

Afterwards that object has a method with the name of the enum type, here `Day`:

```
say $x.Day;                 # 1
```

The value of first constant is 0, the next 1 and so on, unless you explicitly provide another value with pair notation:

```
Enum Hackers (:Larry<Perl>, :Guido<Python>, :Paul<Lisp>);
```

You can check if a specific value was mixed in by using the versatile smart match operator, or with `.does`:

```
if $today ~~ Day::Fri {
    say "Thank Christ it's Friday"
}
if $today.does(Fri) { ... }
```

Note that you can specify the name of the value only (like `Fri`) if that's unambiguous, if it's ambiguous you have to provide the full name `Day::Fri`.

Unicode

Perl 5's Unicode model suffers from a big weakness: it uses the same type for binary and for text data. For example if your program reads 512 bytes from a network socket, it is certainly a byte string. However when (still in Perl 5) you call `uc` on that string, it will be treated as text. The recommended way is to decode that string first, but when a subroutine receives a string as an argument, it can never surely know if it had been encoded or not, ie if it is to be treated as a blob or as a text.

Perl 6 on the other hand offers the type `buf`, which is just a collection of bytes, and `Str`, which is a collection of logical characters.

Logical character is still a vague term. To be more precise a `Str` is an object that can be viewed at different levels: `Byte`, `Codepoint` (anything that the Unicode Consortium assigned a number to is a codepoint),`Grapheme` (things that visually appear as a character) and `CharLingua` (language defined characters).

For example the string with the hex bytes `61 cc 80` consists of three bytes (obviously), but can also be viewed as being consisting of two codepoints with the names `LATIN SMALL LETTER A` (U+0041) and`COMBINING GRAVE ACCENT` (U+0300), or as one grapheme that, if neither my blog software nor your browser kill it, looks like this: à.

So you can't simply ask for the length of a string, you have to ask for a specific length:

```
$str.bytes;
$str.codes;
$str.graphs;
```

There's also method named `chars`, which returns the length in the current Unicode level (which can be set by a pragma like `use bytes`, and which defaults to graphemes).

In Perl 5 you sometimes had the problem of accidentally concatenating byte strings and text strings. If you should ever suffer from that problem in Perl 6, you can easily identify where it happens by overloading the concatenation operator:

```
sub GLOBAL::infix:<~> is deep (Str $a, buf $b)|(buf $b, Str $a) {
    die "Can't concatenate text string «"
        ~ $a.encode("UTF-8")
          "» with byte string «$b»\n";
}
```

Scoping

```
for 1 .. 10 -> $a {
    # $a visible here
}
# $a not visible here

while my $b = get_stuff() {
    # $b visible here
}
# $b still visible here
```

```
my $c = 5;
{
    my $c = $c;
    # $c is undef here
}
# $c is 5 here

my $y;
my $x = $y + 2 while $y = calc();
# $x still visible
```

Lexical Scoping

Scoping in Perl 6 is quite similar to that of Perl 5. A Block introduces a new lexical scope. A variable name is searched in the innermost lexical scope first, if it's not found it is then searched for in the next outer scope and so on. Just like in Perl 5 a `my` variable is a proper lexical variable, and an `our` declaration introduces a lexical alias for a package variable.

But there are subtle differences: variables are exactly visible in the rest of the block where they are declared, variables declared in block headers (for example in the condition of a `while` loop) are not limited to the block afterwards.

If you want to limit the scope, you can use formal parameters to the block:

```
if calc() -> $result {
    # you can use $result here
}
# $result not visible here
```

Variables are visible immediately after they are declared, not at the end of the statement as in Perl 5.

```
my $x = .... ;
        ^^^^^
        $x visible here in Perl 6
        but not in Perl 5
```

Dynamic Scoping

The `local` adjective is now called `temp`, and if it's not followed by an initialization the previous value of that variable is used (not `undef`).

There's also a new kind of dynamically scoped variable called a *hypothetical* variable. If the block is left with an exception, then the previous value of the variable is restored. If not, it is kept.

Context Variables

Some variables that are global in Perl 5 (`$!`, `$_`) are *context* variables in Perl 6, that is they are passed between dynamic scopes.

This solves an old Problem in Perl 5. In Perl 5 a `DESTROY` sub can be called at a block exit, and accidentally change the value of a global variable, for example one of the error variables:

```
# Broken Perl 5 code here:
sub DESTROY { eval { 1 }; }

eval {
    my $x = bless {};
    die "Death\n";
};
print $@ if $@;           # No output here
```

In Perl 6 this problem is avoided by not implicitly using global variables.

(In Perl 5.14 there is a workaround that protects `$@` from being modified, thus averting the most harm from this particular example.)

Pseudo-packages

If a variable is hidden by another lexical variable of the same name, it can be accessed with the `OUTER` pseudo package

```
my $x = 3;
{
    my $x = 10;
    say $x;                 # 10
    say $OUTER::x;          # 3
    say OUTER::<$x>         # 3
}
```

Likewise a function can access variables from its caller with the `CALLER` and `CONTEXT` pseudo packages. The difference is that `CALLER` only accesses the scope of the immediate caller, `CONTEXT` works like UNIX environment variables (and should only be used internally by the compiler for handling `$_`,`$!` and the like). To access variables from the outer dynamic scope they must be declared with `is context`.

Subset Types

Java programmers tend to think of a type as either a class or an interface (which is something like a crippled class), but that view is too limited for Perl 6. A type is more generally a constraint of what a values a container can constraint. The "classical" constraint is *it is an object of a class* `X` *or of a class that inherits from* `X`. Perl 6 also has constraints like *the class or the object does role* `Y`, or *this piece of code returns true for our object*. The latter is the most general one, and is called a *subset* type:

```
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
```

(Try it out, Rakudo implements subset types already, but not yet mulit dispatch based on subset types).

You can also use anonymous subtypes in signatures:

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

Quoting and Parsing

Perl 5 had single quotes, double quotes and `qw(...)` (single quotes, splitted on whitespaces) as well as the `q(..)` and `qq(...)` forms which are basically synonyms for single and double quotes.

Perl 6 in turn defines a quote operator named `Q` that can take various modifiers. The `:b` (*backslash*) modifier allows interpolation of backslash escape sequences like `\n`, the `:s` modifier allows interpolation of scalar variables, `:c` allows the interpolation of closures (`"1 + 2 = { 1 + 2 }"`) and so on, `:w` splits on words as `qw/.../` does.

You can arbitrarily combine those modifiers. For example you might wish a form of `qw/../` that interpolates only scalars, but nothing else? No problem:

```
my $stuff = "honey";
my @list = Q :w :s/milk toast $stuff with\tfunny\nescapes/;
say @list[*-1];                          # prints with\nfunny\nescapes
```

Quoting

Here's a list of what modifiers are available, mostly stolen from S02 directly. All of these also have long names, which I omitted here.

```
Features:
    :q          Interpolate \\, \q and \'
    :b          Other backslash escape sequences like \n, \t
Operations:
    :x          Execute as shell command, return result
    :w          Split on whitespaces
    :ww         Split on whitespaces, with quote protection
Variable interpolation
    :s          Interpolate scalars   ($stuff)
    :a          Interpolate arrays    (@stuff[])
    :h          Interpolate hashes    (%stuff{})
    :f          Interpolate functions (&stuff())
Other
    :c          Interpolate closures  ({code})
    :qq         Interpolate with :s, :a, :h, :f, :c, :b
    :regex      parse as regex
```

Quoting (continued)

There are some short forms which make life easier for you:

```
q          Q:q
qq         Q:qq
m          Q:regex
```

You can also omit the first colon : if the quoting symbol is a short form, and write it as a singe word:

```
symbol       short for
qw           q:w
Qw           Q:w
qx           q:x
Qc           Q:c
# and so on.
```

However there is one form that does not work, and some Perl 5 programmers will miss it: you can't write `qw(...)` with the round parenthesis in Perl 6. It is interpreted as a call to sub `qw`.

Parsing

This is where parsing comes into play: Every construct of the form `identifier(...)` is parsed as sub call. Yes, every.

```
if($x<3)
```

is parsed as a call to sub `if`. You can disambiguate with whitespaces:

```
if ($x < 3) { say '<3' }
```

Or just omit the parens altogether:

```
if $x < 3 { say '<3' }
```

This implies that Perl 6 has no keywords. Actually there are keywords like `use` or `if`, but they are only special if used in a special syntax.

The Reduction Meta Operator

The reduction meta operator `[...]` can enclose any associative infix operator, and turn it into a list operator. This happens as if the operator was just put between the items of the list, so `[op] $i1, $i2, @rest` returns the same result as if it was written as `$i1 op $i2 op @rest[0] op @rest[1] ....`

This is a very powerful construct that promotes the plus `+` operator into a `sum` function, `~` into a `join` (with empty separator) and so on. It is somewhat similar to the `List.reduce` function, and if you had some exposure to functional programming, you'll probably know about `foldl` and `foldr` (in Lisp or Haskell). Unlike those `[...]` respects the associativity of the enclosed operator, so `[/] 1, 2, 3` is interpreted as `(1 / 2) / 3` (left associative), `[**] 1, 2, 3` is handled correctly as `1 ** (2**3)` (right associative).

Like all other operators whitespaces are forbidden, so you while you can write `[+]`, you can't say `[ + ]`.

Since comparison operators can be chained, you can also write things like

```
if    [==] @nums { say "all nums in @nums are the same" }
elsif [<]  @nums { say "@nums is in strict ascending order" }
elsif [<=] @nums { say "@nums is in ascending order"}
```

You can even reduce the assignment operator:

```
my @a = 1..3;
[=] @a, 4;              # same as @a[0] = @a[1] = @a[2] = 4;
```

Note that `[...]` always returns a scalar, so `[,] @list` is actually the same as `[@list]`.

The Cross Meta Operator

```
for <a b> X 1..3 -> $a, $b {
    print "$a: $b   ";
}
# output: a: 1   a: 2   a: 3   b: 1   b: 2   b: 3

.say for <a b c> X 1, 2;
# output: a1\n a2\n b1\n b2\n c1\n c2\n
# (with real newlines instead of \n)
```

The cross operator `X` returns the Cartesian product of two or more lists, which means that it returns all possible tuples where the first item is an item of the first list, the second item is an item of second list etc.

If an operator follows the `X`, then this operator is applied to all tuple items, and the result is returned instead. So `1 , 2 X+ 3 , 6` will return the values `1+3 , 1+6 , 2+3 , 2+6` (evaluated as `4 , 7 , 5 , 8` of course).

Most Wanted Perl6 Modules

- ExtUtils::Command: Port from Perl 5 (or should that be Shell::Command?)
- Email::MIME
- HTTP::Response (and likely HTTP::Request too). LWP::Simple is evolving and evolving, so maybe there's a need for a full-blown LWP? (Of course! Preferably with JavaScript handling at some point..)
- SSL
- Digest::* (preferably with a unified API) - MD5 and SHA256 already exist. What other Digests are wanted?
- All the boring stuff like PDF, Excel etc
- Data formats of all kinds from SOAP, CSV, YouNameIt (ideally Grammar - validating - generating - parsing - the whole range)
- Net::* - all kinds of networking stuff
- A common widget set (Qt, Gtk)
- Something like either GD, Imager or Image Magick
- GZIP!
- Curses, Readline - all things terminal-ish
- Crypto - GPG etc
- Smart::Match, mebbe with a bit different name?