Rakudo and NQP Internals

The guts tormented implementers made

Jonathan Worthington

September 17, 2013

EDUMENT

Development and Mentorship

About This Course

Perl 6 is a large language, incorporating many features that are demanding to implement correctly.

It's easy for such a software project to be drowned by unmanaged complexity. Earlier phases of the Rakudo and NQP projects have suffered in this way, as we learned - the hard way - about the complexities that arose and could spread unchecked over the implementation.

This course will teach you how to work with Rakudo and NQP internals. Encoded in their design is a wealth of learning, built up over years, about how (and how not) to write a Perl 6 implementation. Thus, this course will also teach you why things are the way they are.

About The Teacher

- Computer Science background
- Chose to travel the world and help implement Perl 6 instead of doing a PhD
- There's more than one way to get "Permanent head Damage" :-)
- Somehow got hired at Edument AB along the way, as a teacher/consultant
- Rakudo Perl 6 core developer since 2008
- Architect of 6model, MoarVM and various aspects of NQP and Rakudo

Course overview - Day 1

- The eagle's eye view: Compilers, and the NQP/Rakudo Architecture
- The NQP Language
- The compilation pipeline
- QAST
- Exploring nqp:: ops

Course overview - Day 2

- 6model
- Bounded Serialization and Module Loading
- The regex and grammar engine
- The JVM backend
- The MoarVM backend

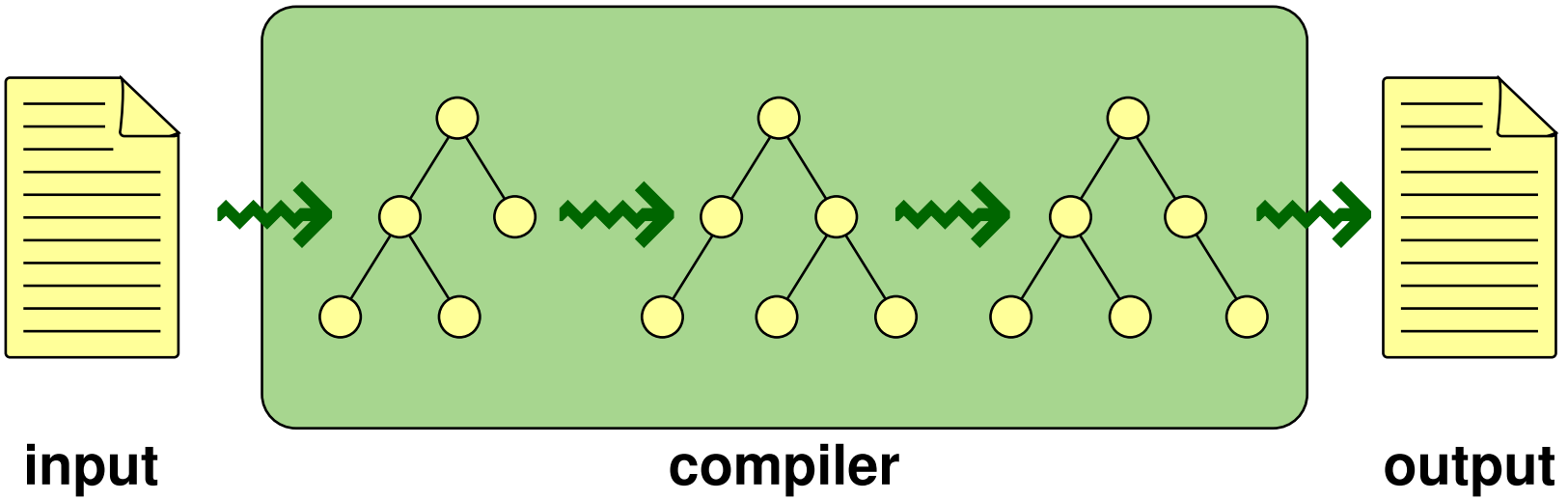
The eagle's eye view

Compilers, and the NQP/Rakudo Architecture

What compilers do

Compilers are really "just" translators

Translate a high level language (such as Perl 6) into a low level one (such as JVM bytecode)



Take flat input (text) and produce flat output (text or binary), but **the insides are rich in data structures**

Dealing with stuff as strings is, generally, a last resort

What runtimes do

To run a language like Perl 6 involves more than just translating it to low level code. Additionally, it needs **runtime support** to provide:

- Memory management
- I/O, IPC, OS interaction
- Concurrency
- Dynamic optimization

Building the things we need to build the thing

Various attempts have been made to build Perl 6 out of existing compiler construction technologies. Early designs for the compiler were at least partly based on conventional assumptions.

Such attempts were informative, but haven't worked out too well in the long run.

Perl 6 presents some interesting challenges...

Perl 6 is parsed using Perl 6

The Perl 6 standard grammar is written in Perl 6. It depends on...

- Transitive **longest token matching** (we'll see more on this later)
- Being able to switch back and forth between languages (main language, regex language, quoting language...)
- Being able to **derive new languages dynamically** (new operators, custom quoting constructs)
- Seemless integration between bottom-up expression parsing and top-down parsing for larger constructs
- Keeping various bits of state around for **awesome error reporting**

All of which essentially represent a new paradigm in parsing.

Not statically typed or dynamically typed

Perl 6 is a **gradually typed** language.

```
my int $distance = distance-between('Lund', 'Kiev');
my int $time = prompt('Travel time: ').Int;
say "Average speed: { $distance / $time }";
```

We want to make use of knowing that `$distance` and `$time` are native integers to produce better code than if we had no knowledge of the types (should just be a native division instruction in the output code).

Compile-time and runtime blur

Runtime can do some compile-time:

```
eval slurp @demos[$n];
```

Compile-time can do some runtime:

```
my $comp-time = BEGIN now;
```

Notice how the compile time computation's result must be persisted until runtime, which may have a process boundary between them!

NQP as a language

The Perl 6 grammar clearly needed to be expressed in Perl 6. This would in turn need to integrate into the rest of the compiler. Writing the whole lot in Perl 6 thus followed fairly naturally.

However, full-blown Perl 6 is large, and writing a good optimizer for it takes a lot of work.

Therefore, the **NQP (Not Quite Perl 6)** language was born: a small subset of Perl 6 designed for implementing compilers. NQP and Rakudo are mostly written in NQP.

NQP as a compiler construction toolchain

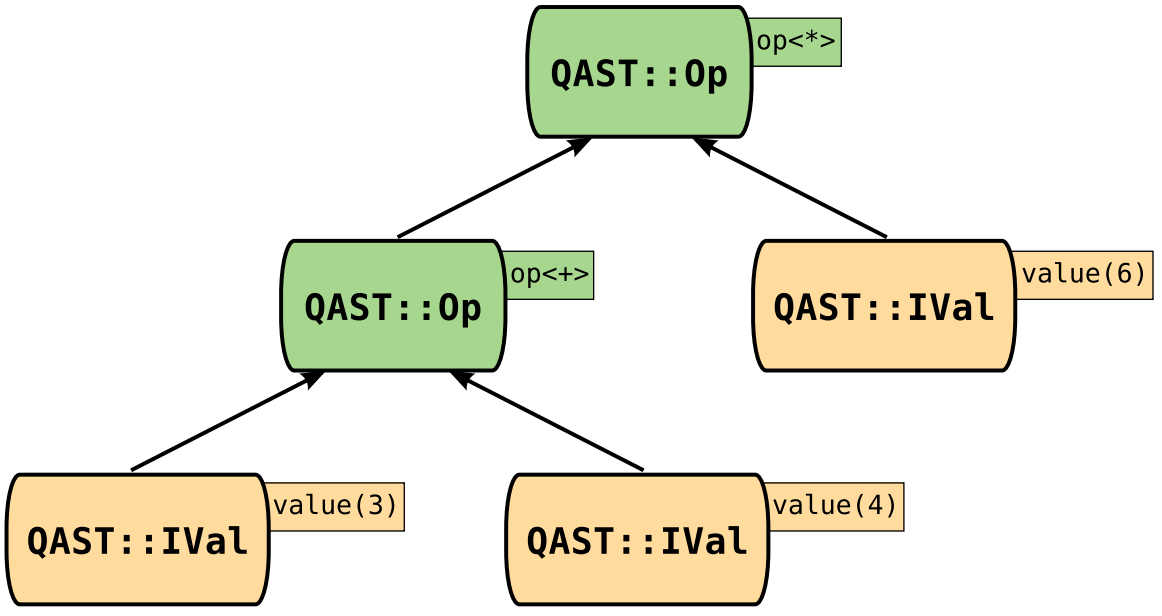
Peeking inside the NQP `src` directory reveals that there's a lot more than just NQP itself, however.

- **NQP, how, core**: these contain the NQP compiler, meta-objects (which specify how NQP's classes and roles work) and built-ins
- **HLL**: common infrastructure for building a high-level language compiler, shared between Rakudo and NQP
- **QAST**: nodes of the Q Abstract Syntax Tree, a tree notation representing the semantics of a program (that is, what will it do when executed)
- **QRegex**: objects involved in parsing and executing regexes and grammars
- **vm**: virtual machine abstraction layers, since NQP and Rakudo can run on Parrot, the JVM, MoarVM...

QAST

QAST trees are one of the most important data structures in NQP and Rakudo internals.

An **Abstract Syntax Tree** represents what a program does when executed. It is abstract in the sense of being abstracted away from the particular language that a program was written in.



the QAST for the expression `(3 + 4) * 6`

QAST

Different QAST nodes represent things like:

- Variables
- Operations (arithmetic, string, invocation, etc.)
- Literals
- Blocks

Note that there are no QAST nodes for things like classes, since those are compile-time declarations rather than runtime execution.

The nqp:: op set

Another important part of the compiler toolchain is the `nqp::op` instruction set. There are two ways in which you will encounter it, and it is critical to understand the difference!

You can use them **in NQP code**, in which case you are saying that you wish to execute the operation at that point in your program:

```
say(nqp::time_n())
```

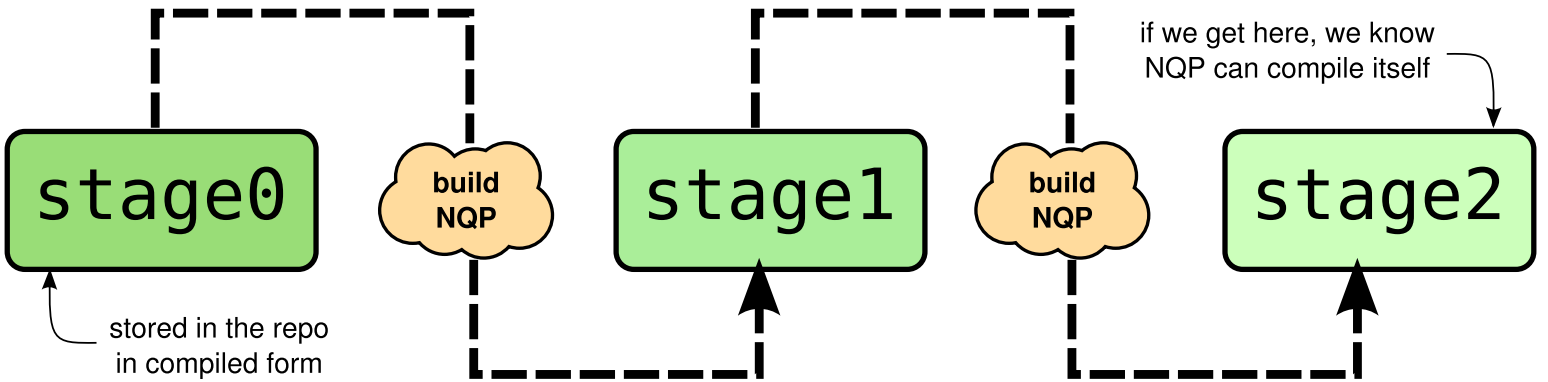
The exact same instruction set is also used in a **QAST tree** that represents a program that is being compiled:

```
QAST::Op.new(
    :op('call'), :name('&say'),
    QAST::Op.new( :op('time_n') )
)
```

Bootstrapping in a nutshell

One may wonder how NQP can ever be compiled when it's written almost entirely in NQP.

Inside each of the `vm` subdirectories is a `stage0` directory. It contains a compiled NQP (PIR files on Parrot, JAR files on JVM, etc.) We then:



Thus, the NQP you `make test` on is one that can recreate itself.

Every so often, we update the `stage0` with the latest version

How Rakudo uses NQP

Rakudo itself is not a bootstrapping compiler, which makes its development a bit easier. Most of Rakudo is written in NQP. This includes:

- The **heart of the compiler** itself, which parses Perl 6 source, builds up QAST, manages declarations and does various optimizations
- The **meta-objects**, which specify how different kinds of type (classes, roles, enums, subsets) work
- The **bootstrap**, which pieces together enough of the Perl 6 core types for us to be able to write the built-in classes, roles and routines in Perl 6

Thus, while some of Rakudo is accessible if you know Perl 6, knowing NQP - both as a langauge and as a compiler toolchain - is the gateway to working with most of the rest of Rakudo.

The NQP language

It's Not Quite Perl 6, but quite OK for building Perl 6

Design goals

NQP is designed to be...

- Ideal for writing compiler-related things in
- Almost a subset of Perl 6
- Much simpler to compile and optimize than Perl 6

Of note, it avoids:

- Assignment
- Flattening and laziness
- Operators being multi-dispatch (so, no overloading)
- Having lots of built-ins

Literals

Integer literals

```
0       42      -100
```

Floating point literals (no Rat in NQP!)

```
0.25    1e10    -9.9e-9
```

String literals

```
'non-interpolating'         "and $interpolating"
q{non-interpolating}        qq{and $interpolating}
Q{not even backslashes}
```

Sub calls

In NQP, these always need the parentheses:

```
say('Mushroom, mushroom');
```

Like in Perl 6, this adds an & to the name and does a lexical lookup of the routine.

However, there is no list-op calling syntax:

```
plan 42;    # "Confused" parse error
foo;        # Does not call foo; always a term
```

This is perhaps the most common NQP beginner mistake.

Variables

Can be `my` (lexical) or `our` (package) scoped:

```
my $pony;
our $stable;
```

The usual set of sigils are available:

```
my $ark;                # Starts as NQPMu
my @animals;            # Starts as []
my %animal_counts;      # Starts as {}
my &lasso;              # Starts as NQPMu
```

Dynamic variables are also supported:

```
my @*blocks;
```

Binding

NQP does not offer the = assignment operator. Only the := binding operator is provided. This frees NQP from the complexity of Perl 6 container semantics.

Here's a simple scalar example:

```
my $ast := QAST::Op.new( :op('time_n') );
```

Binding and arrays

Note that binding has **item assignment precedence**, so you can not write:

```
my @states := 'start', 'running', 'done';    # Wrong!
```

Instead, this must be expressed as one of:

```
my @states := ['start', 'running', 'done'];  # Fine
my @states := ('start', 'running', 'done');  # Same thing
my @states := <start running done>;          # Cutest
```

Natively typed variables

At present, NQP doesn't really support type constraints on variables. The exception is that it will pay attention to **native types**.

```
my int $idx := 0;
my num $vel := 42.5;
my str $mug := 'coffee';
```

Note: in NQP, binding is used on native types! This is illegal in Perl 6, where natives can only be assigned. It's all rather artificial, though, in so far as an assignment to a native type in Perl 6 actually compiles down to the `nqp::bind(...)` op!

Control flow

Most of the Perl 6 conditional and looping constructs also exist in NQP. As in real Perl 6, parentheses are not required around the conditional, and pointy blocks can be used also. Loop constructs support `next`/`last`/`redo`.

```
if $optimize {
    $ast := optimize($ast);
}
elsif $trace {
    $ast := insert_tracing($ast);
}
```

Available: if, unless, while, until, repeat, for

Missing: loop, given/when, FIRST/NEXT/LAST phasers

Subroutines

Declared much like in Perl 6, however the parameter list is mandatory even if taking no parameters. You may either `return` or use the last statement as an implicit return value.

```
sub mean(@numbers) {
    my $sum;
    for @numbers { $sum := $sum + $_ }
    return $sum / +@numbers;
}
```

Slurpy parameters are also available, as is | to flatten argument lists.

Note: parameters can get type constraints, but as with variables, only the native types count at present. (Exception: multiple dispatch; more later.)

Named arguments and parameters

Named parameters are supported:

```
sub make_op(:$name) {
    QAST::Op.new( :op($name) )
}

make_op(name => 'time_n');  # Fat-arrow syntax
make_op(:name<time_n>);     # Colon-pair syntax
make_op(:name('time_n'));   # The same
```

Note: NQP does *not* have `Pair` objects! Pairs - colonpairs or fat-arrow pairs - only make sense in the context of an argument list.

Blocks and pointy blocks

Pointy blocks are available with the familiar Perl 6 syntax:

```
sub op_maker_for($op) {
    return -> *@children, *%adverbs {
        QAST::Op.new( :$op, |@children, |%adverbs )
    }
}
```

As can be seen from this example, they have closure semantics.

Note: Plain blocks are also available for use as closures, but do not take an implicit $_ argument like in Perl 6!

Built-ins and nqp:: ops

NQP has relatively few built-ins. However, it provides full access to the NQP instruction set. Here are a few common instructions that are useful to know.

```
# On arrays
nqp::elems, nqp::push, nqp::pop, nqp::shift, nqp::unshift

# On hashes
nqp::elems, nqp::existskey, nqp::deletekey

# On strings
nqp::substr, nqp::index, nqp::uc, nqp::lc
```

We'll discover more during the course.

Exception handling

An exception can the thrown using the `nqp::die` instruction:

```
nqp::die('Oh gosh, something terrible happened');
```

The `try` and `CATCH` constructs are also available, though unlike in full Perl 6 you are not expected to smart-match inside of the `CATCH`; once you're in there, it's considered that the exception is caught (modulo an explicit `nqp::rethrow`).

```
try {
    something();
    CATCH { say("Oops") }
}
```

Classes, attributes and methods

Declared with the `class`, `has` and `method` keywords, as in Perl 6. A class may be lexical (`my`) or package (`our`) scoped (the default).

```
class VariableInfo {
    has @!usages;

    method remember_usage($node) {
        nqp::push(@!usages, $node)
    }

    method get_usages() {
        @!usages
    }
}
```

The `self` keyword is also available, and methods can have parameters just like subs.

More on attributes

NQP has no automatic accessor generation, so you can't do:

```
has @.usages; # Not supported
```

Natively typed attributes are supported, and will be efficiently stored directly in the object body. Any other types are ignored.

```
has int $!flags;
```

Unlike in Perl 6, the default constructor can be used to set the private attributes, since that's all we have.

```
my $vi := VariableInfo.new(usages => @use_so_far);
```

Roles (1)

NQP supports roles. Like classes, roles can have attributes and methods.

```
role QAST::CompileTimeValue {
    has $!compile_time_value;

    method has_compile_time_value() {
        1
    }

    method compile_time_value() {
        $!compile_time_value
    }

    method set_compile_time_value($value) {
        $!compile_time_value := $value
    }
}
```

Roles (2)

A role can be composed into a class using the does trait:

```
class QAST::WVal is QAST::Node does QAST::CompileTimeValue {
    # ...
}
```

Alternatively, the MOP can be used to mix a role into an individual object:

```
method set_compile_time_value($value) {
    self.HOW.mixin(self, QAST::CompileTimeValue);
    self.set_compile_time_value($value);
}
```

Multiple dispatch

Basic multiple dispatch is supported. It is a subset of the Perl 6 semantics, using a simpler (but compatible) version of the candidate sorting algorithm.

Unlike in full Perl 6, you **must write a `proto`** sub or method; there is not auto-generation.

```
proto method as_jast($node) {*}

multi method as_jast(QAST::CompUnit $cu) {
    # compile a QAST::CompUnit
}

multi method as_jast(QAST::Block $block) {
    # compile a QAST::Block
}
```

Exercise 1

A chance to get acquainted with the basic NQP syntax, if you have not done so already.

Also a chance to learn how common mistakes look, so you can recognize them if you encounter them in real work. :-)

Grammars

While in many areas NQP is quite limited compared to full Perl 6, grammars are supported almost to the same level. This is because NQP grammars have to be good enough to cope with parsing Perl 6 itself.

Grammars are a kind of class, and are introduced using the `grammar` keyword.

```
grammar INIFile {
}
```

In fact, grammars are so like classes that in NQP they are implemented by the same meta-object. The difference is what they inherit from by default and what you put inside of them.

INI Files

As an initial, simple example, we'll consider parsing INI files.

Keys with values, potentially arranged into sections.

```
name = Animal Facts
author = jnthn

[cat]
desc = The smartest and cutest
cuteness = 100000

[dugong]
desc = The cow of the sea
cuteness = -10
```

The overall approach

A grammar contains a set of rules, declared with the keywords `token`, `rule` or `regex`. Really, they are just like methods, but written in rule syntax.

```
token integer { \d+ }       # one or more digits
token sign    { <[+-]> }    # + or - (character class)
```

More complex rules are made up by calling existing ones:

```
token signed_integer { <sign>? <integer> }
```

These calls to other rules can be quantified, placed in alternations, and so forth.

Aside: grammars and regexes

At this point, you may be wondering how grammars and regexes relate. After all, a grammar seems to be made up of regex-like things.

There is also a `regex` declarator, which can be used in a grammar.

```
regex email { <[\w.-]>+ '@' <[\w.-]>+ '.' \w+ }
```

The key difference is that **a `regex` will backtrack**, whereas a `rule` or `token` will not. Supporting backtracking involves keeping lots of state, and for a complex grammar parsing a lot of input, this would quickly use up large amounts of memory! Big languages tend to avoid backtracking in their parsers.

Aside: regexes in NQP

NQP does provide support for regexes in the normal sense too, for smaller scale things.

```
if $addr ~~ /<[\w.-]>+ '@' <[\w.-]>+ '.' \w+/ {
    say("I'll mail you maybe");
}
else {
    say("That's no email address!");
}
```

This evaluates to the match object.

Parsing entries

An entry has a key (some word characters) and a value (everything up to the end of the line):

```
token key   { \w+ }
token value { \N+ }
```

Together, they form an entry:

```
token entry { <key> \h* '=' \h* <value> }
```

The \h matches any horizontal whitespace (space, tab, etc.). The = must be quoted, as anything non-alphanumeric is treated as regex syntax in Perl 6.

Start at the TOP

The entry point to a grammar is the special rule, TOP. For now, we look for the entire file to be lines containing an entry or simply nothing.

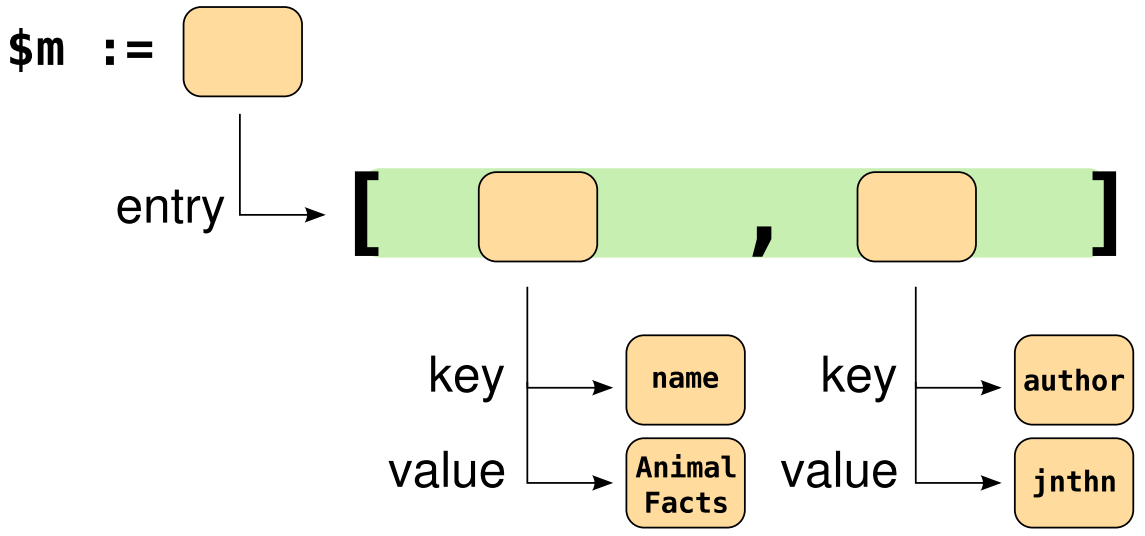
```
token TOP {
    ^
    [
    | <entry> \n
    | \n
    ]+
    $
}
```

Note that in Perl 6, square brackets are a non-capturing group (the Perl 5 `(?:...)`), not a character class.

Trying our grammar

We can try our grammar out by calling the `parse` method on it. This returns a **match object**.

```
my $m := INIFile.parse(Q{
name = Animal Facts
author = jnthn
});
```



Iterating through the results

Each call to a rule yields a match object, and the `<entry>` call syntax will capture it into the match object.

Since we matched many entries we get an array under the `entry` key in the match object.

Thus, we can do loop over it to get each of the entries:

```
for $m<entry> -> $entry {
    say("Key: {$entry<key>}, Value: {$entry<value>}");
}
```

Tracing our grammar

NQP comes with some built-in support for tracing where grammars go. It's not a full-blown debugger, but it can be helpful to see how far a grammar gets before it fails. It is turned on with:

```
INIFile.HOW.trace-on(INIFile);
```

And produces output like:

```
Calling parse
  Calling TOP
    Calling entry
      Calling key
      Calling value
    Calling entry
      Calling key
      Calling value
```

token vs. rule

When we use `rule` in place of `token`, any whitespace after an atom is turned into a **non-capturing** call to `ws`. That is:

```
rule entry { <key> '=' <value> }
```

Is the same as:

```
token entry { <key> <.ws> '=' <.ws> <value> <.ws> } # . = non-capturing
```

We inherit a default `ws`, but we can supply our own too:

```
token ws { \h* }
```

Parsing sections (1)

A section has a heading and many entries. However, the top-level can also have entries. Thus, it makes sense to factor this out.

```
token entries {
    [
    | <entry> \n
    | \n
    ]+
}
```

The TOP rule can then become:

```
token TOP {
    ^
    <entries>
    <section>+
    $
}
```

Parsing sections (2)

Last but not least here is the section token:

```
token section {
    '[' ~ ']' <key> \n
    <entries>
}
```

The ~ syntax is cute. The first line is like:

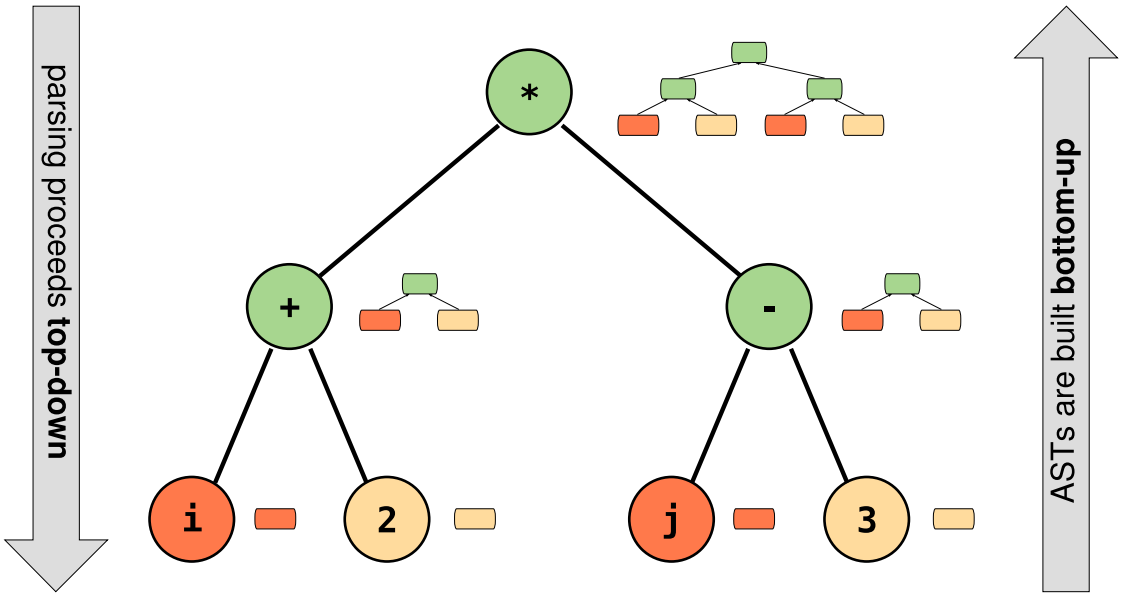
```
'[' <key> ']' \n
```

However, failure to find the closing] produces a descriptive error message instead of just failing to match.

Actions

Parsing a grammar can happen using an **actions class**; its methods have names matching some or all rules in the grammar.

The methods are called **after a successful match of the corresponding rule**.



In the Rakudo and NQP compilers, **actions construct QAST trees**. For this example, we'll do something a little simpler.

Actions example: aim

Given an INI file like:

```
name = Animal Facts
author = jnthn

[cat]
desc = The smartest and cutest
cuteness = 100000
```

We'd like to use the actions class to build up a **hash of hashes**. The top level hash will contain the keys `cat` and `_` (the underscore collecting any keys not in a section). The values are hashes of the key/value pairs in that section.

Actions example: entries

Action methods take the match object of the just-matched rule as a parameter. It is convenient to put it into $/ so we can use the $<entry> sugar (which maps to $/<entry>).

```
class INIFileActions {
    method entries($/) {
        my %entries;
        for $<entry> -> $e {
            %entries{$e<key>} := ~$e<value>;
        }
        make %entries;
    }
}
```

Finally, `make` attaches the produced hash to $/. This is so the `TOP` action method will be able to retrieve it while building the top-level hash.

Actions example: TOP

The `TOP` action method builds the top-level hash out of the hashes made by the `entries` action method. While `make` attaches something to `$/`, the `.ast` method retrieves what was attached to some other match object.

```
method TOP($/) {
    my %result;
    %result<_> := $<entries>.ast;
    for $<section> -> $sec {
        %result{$sec<key>} := $sec<entries>.ast;
    }
    make %result;
}
```

Thus, the top-level hash gets the hashes produced by the `entries` action method installed into it, by section name.

Actions example: parsing with actions

The actions are passed as a named parameter to `parse`:

```
my $m := INIFile.parse($to_parse, :actions(INIFileActions));
```

The result hash can be obtained from the resulting match object using the `.ast`, as we already saw.

```
my %sections := $m.ast;
for %ini -> $sec {
    say("Section {$sec.key}");
    for $sec.value -> $entry {
        say("    {$entry.key}: {$entry.value}");
    }
}
```

Actions example: output

The dumping code on the previous slide produces output as follows:

```
Section _
    name: Animal Facts
    author: jnthn
Section cat
    desc: The smartest and cutest
    cuteness: 100000
Section dugong
    desc: The cow of the sea
    cuteness: -10
```

Exercise 2

A chance to have a little practice with grammars and actions.

The goal is to parse the text format of the Perl 6 IRC log; for example, see http://irclog.perlgeek.de/perl6/2013-07-19/text

Another example: SlowDB

Parsing INI files is a nice introductory example, but feels a long way from a compiler. As a step in that direction, we'll build a small, stupid, in-memory database with a query interpreter.

It should work something like this:

```
INSERT name = 'jnthn', age = 28
[
    result: Inserted 1 row
]
SELECT name WHERE age = 28
[
    name: jnthn
]
SELECT name WHERE age = 50
Nothing found
```

The query parser (1)

We either parse an `INSERT` or `SELECT` query.

```
token TOP {
    ^ [ <insert> | <select> ] $
}

token insert {
    'INSERT' :s <pairlist>
}

token select {
    'SELECT' :s <keylist>
    [ 'WHERE' <pairlist> ]?
}
```

Note that `:s` turns on auto-`<.ws>` insertion.

The query parser (2)

The `pairlist` and `keylist` rules are defined as follows.

```
rule pairlist { <pair>+ % [ ',' ] }
rule pair     { <key> '=' <value>  }
rule keylist  { <key>+ % [ ',' ] }
token key     { \w+ }
```

The interesting new syntax here is `%`. It attaches to the last quantifier, and indicates that something (in this case, a comma) should come between each of the quantified elements.

The square brackets around the comma literal are to ensure `<.ws>` calls are generated as part of the separator.

The query parser (3)

Finally, here is how values can be parsed.

```
token value { <integer> | <string> }
token integer { \d+ }
token string  { \' <( <-[']>+ )> \' }
```

Notice the use of the $<($ **and** $)>$ syntax. These indicate the limits of what should be captured by the `string` token overall, meaning that the quote characters don't end up being captured.

Alternations and LTM (1)

Recall the top rule:

```
token TOP {
    ^ [ <insert> | <select> ] $
}
```

If we trace the parsing of a SELECT query, we see something like this:

```
Calling parse
  Calling TOP
    Calling select
      Calling ws
      Calling keylist
```

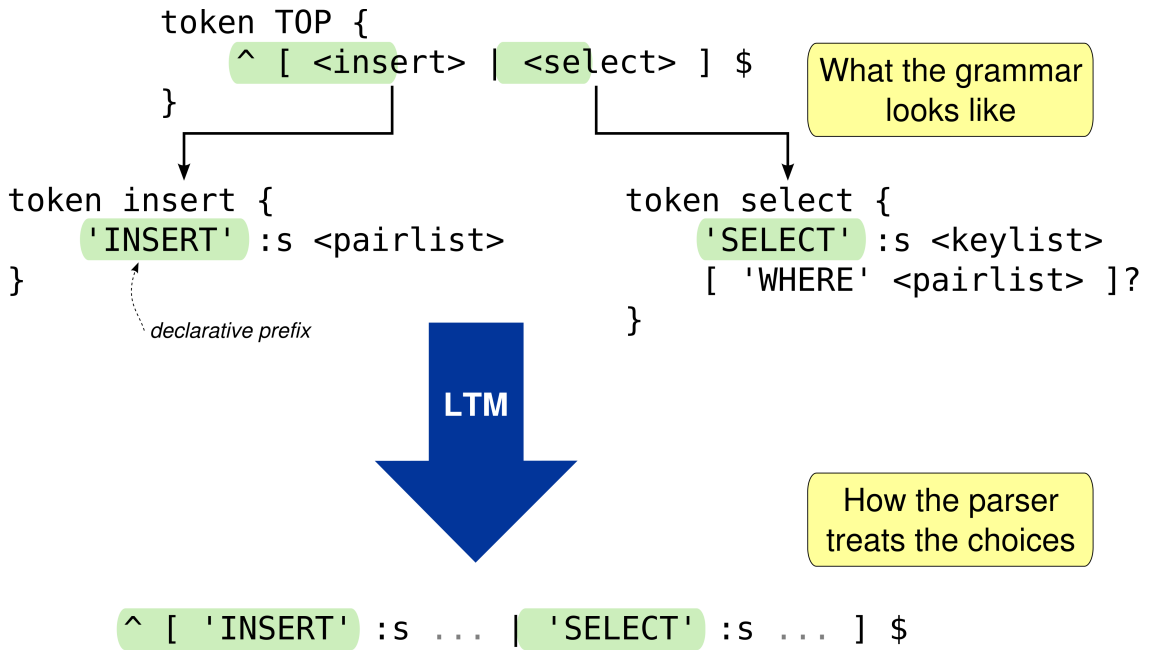
So how did it know not to bother trying <insert>?

Alternations and LTM (2)

The answer is **Transitive Longest Token Matching**. The grammar engine builds an NFA (state machine) that, upon encountering an alternation, sorts the branches by the number of characters they would match. It then tries them longest first, not bothering with those it realizes are impossible.

Alternations and LTM (3)

It doesn't just look at a rule in isolation. Instead, it **considers subrule calls transitively**. This means entire call chains that lead to something impossible can be eliminated.



It is bounded by non-declarative constructs (such as a lookahead, a code block, or a call to the default `ws` rule) or recursive subrule calls.

A slight pain point

One annoyance we have is that our `TOP` action method ends up looking like this:

```
method TOP($/) {
    make $<select> ?? $<select>.ast !! $<insert>.ast;
}
```

It's easy to see how this will become painful to maintain once we add `UPDATE` and `DELETE` queries. It's even more painful if the grammar is subclassed.

Our `value` action method is similar:

```
method value($/) {
    make $<integer> ?? $<integer>.ast !! $<string>.ast;
}
```

Protoregexes

The answer to our woes is **protoregexes**. They provide **a more extensible way to express an alternation**.

```
proto token value {*}
token value:sym<integer> { \d+ }
token value:sym<string>  { \' <( <-[']>+ )> \' }
```

Essentially, we introduce a new syntactic category, `value`, and then define difference cases of it. A call like <value> will use LTM to sort and try the candidates - just like an alternation did.

Protoregexes and action methods (1)

Back in the actions class, we need to update our action methods to match the names of the rules:

```
method value:sym<integer>($/) { make ~$/ }
method value:sym<string>($/)  { make ~$/ }
```

However, **we do not need an action method for `value` itself**. Anything that looks at $<value> will be provided with the match object from the successful candidate - and thus $<value>.ast will obtain the correct thing.

Protoregexes and action methods (2)

For example, after we refactor queries:

```
token TOP { ^ <query> $ }

proto token query {*}
token query:sym<insert> {
    'INSERT' :s <pairlist>
}
token query:sym<select> {
    'SELECT' :s <keylist>
    [ 'WHERE' <pairlist> ]?
}
```

The `TOP` action method can then simply be:

```
method TOP($/) {
    make $<query>.ast;
}
```

keylist and pairlist

These are two boring action methods, included for completeness.

```
method pairlist($/) {
    my %pairs;
    for $<pair> -> $p {
        %pairs{$p<key>} := $p<value>.ast;
    }
    make %pairs;
}

method keylist($/) {
    my @keys;
    for $<key> -> $k {
        nqp::push(@keys, ~$k)
    }
    make @keys;
}
```

Interpreting a query

So how do we ever run the query? Well, here's the action method for `INSERT` queries:

```
method query:sym<insert>($/) {
    my %to_insert := $<pairlist>.ast;
    make -> @db {
        nqp::push(@db, %to_insert);
        [nqp::hash('result', 'Inserted 1 row' )]
    };
}
```

Here, instead of a data structure, we `make` a closure that takes the current database state (an array of hashes, where each hash is a row) and push the hash produced by the `pairlist` action method onto it.

The SlowDB class itself

```
class SlowDB {
    has @!data;

    method execute($query) {
        if QueryParser.parse($query, :actions(QueryActions)) -> $parsed {
            my $evaluator := $parsed.ast;
            if $evaluator(@!data) -> @results {
                for @results -> %data {
                    say("[");
                    say("    {$_.key}: {$_.value}") for %data;
                    say("]");
                }
            } else {
                say("Nothing found");
            }
        } else {
            say('Syntax error in query');
        }
    }
}
```

Exercise 3

A chance to practice with protoregexes a bit, and study for yourself what we have been looking through.

Take the SlowDB example that we have been considering. Add support for the `UPDATE` and `DELETE` queries.

Limitations and other differences from full Perl 6

Here's an assortment of other things worth knowing.

- There is a `use` statement, but it expects anything that it uses to have been pre-compiled already.
- There is no array flattening; `[@a, @b]` is always an array of 2 elements
- The hash composer {} only works for empty hashes; anything other than that will be treated as a block
- `BEGIN` blocks exist but are highly constrained in what they can see in the outer scope (only types, not variables)

Backend differences

NQP on JVM and MoarVM are relatively consistent. NQP on Parrot is the odd one out: not everything is a 6model object. That is, while `.WHAT` or `.HOW` will work on anything in NQP on JVM and MoarVM, it may fail on Parrot. This happens on integer, number and string literals, arrays and hashes, exceptions and some kinds of code object.

Exception handlers also work out a bit differently. Those on JVM and MoarVM run on the stack top at the point of the exception throw, as is the Perl 6 semantics. Those in NQP on Parrot will unwind then run, with resumption being provided by a continuation. Note that Rakudo is consistent on this on all backends.

Overall. . .

NQP, despite being a relatively small subset of Perl 6, still packs in quite a few powerful language features.

Generally, demand for them has been driven by what was needed by those working on Rakudo. As a result, NQPs feature set is shaped by compiler-writing needs.

The grammars and action method material we have covered is perhaps the most important, as this is the starting point for understanding how NQP and Perl 6 are compiled.

The compilation pipeline

Stage by stage, we compile the program...

From start to finish

Now we know a bit about NQP as a language, it's time to dive under the covers and see what happens when we feed NQP a program to run.

To start with, we'll consider this simple example...

```
nqp -e "say('Hello, world')"
```

...all the way from NQP's sub `MAIN` through to the output appearing.

We'll choose the JVM backend to examine this.

The “stagestats” option

We can get an insight into what is going on inside of NQP by running it with the `--stagestats` option, which shows the times for each of the stages that the compiler goes through.

```
Stage start      :   0.000      # Startup
Stage classname  :   0.010      # Compute classname
Stage parse      :   0.067      # Parse source, build AST
Stage ast        :   0.000      # Obtain AST
Stage jast       :   0.106      # Turn into JVM AST
Stage classfile  :   0.032      # Turn into JVM bytecode
Stage jar        :   0.000      # Maybe make a JAR
Stage jvm        :   0.002      # Actually run the code
```

Dumping the parse tree

We can get a dump of some of the stages. For example, `--target=parse` will produce a dump of the parse tree.

```
- statementlist: say('Hello world')
  - statement: 1 matches
    - EXPR: say('Hello world')
      - deflongname: say
        - identifier: say
      - args: ('Hello world')
        - arglist: 'Hello world'
          - EXPR: 'Hello world'
            - value: 'Hello world'
              - quote: 'Hello world'
                - quote_EXPR: 'Hello world'
                  - quote_delimited: 'Hello world'
                    - quote_atom: 1 matches
                    - stopper: '
                    - starter: '
```

Dumping the AST

Also sometimes useful is `--target=ast`, which dumps the QAST (output below has been simplified).

```
- QAST::CompUnit
  - QAST::Block
    - QAST::Var(lexical @ARGS :decl(param))
    - QAST::Stmts
      - QAST::Var(lexical GLOBALish :decl(static))
      - QAST::Var(lexical $?PACKAGE :decl(static))
      - QAST::Var(lexical EXPORT :decl(static))
    - QAST::Stmts say('Hello world')
      - QAST::Stmts
        - QAST::Op(call &say) 'Hello world'
          - QAST::SVal(Hello world)
```

Dumping the JVM AST

You can even get some representation of the low-level AST that is turned into Java bytecode with `--target=jast`, but it's an utter brain-screw (small bit of it below to illustrate). :-)

```
.push_sc Hello world
58 __TMP_S_0
.push_sc &say
.push_idx 1
43
25 __TMP_S_0
.try
186 subcall_noa org/perl6/nqp/runtime/IndyBootstrap subcall_noa 0
:reenter_1
.catch Lorg/perl6/nqp/runtime/SaveStackException;
.push_idx 1
167 SAVER
.endtry
```

Going inside

Our journey starts in NQP's `MAIN` sub, located in `src/NQP/Compiler.nqp`. Here is a slightly simplified version (stripped out setting up command line options and other minor details).

```
class NQP::Compiler is HLL::Compiler {
}

# Create and configure compiler object.
my $nqpcomp := NQP::Compiler.new();
$nqpcomp.language('nqp');
$nqpcomp.parsegrammar(NQP::Grammar);
$nqpcomp.parseactions(NQP::Actions);

sub MAIN(*@ARGS) {
    $nqpcomp.command_line(@ARGS, :encoding('utf8'));
}
```

The HLL::Compiler class

The `command_line` method is inherited from HLL::Compiler, located in `src/HLL/Compiler.nqp`. This class contains the logic that orchestrates the compilation process.

Its functionality includes:

- Argument processing (delegates to HLL::CommandLine)
- Reading source files in from disk
- Invoking each of the stages, stopping at `--target` if specified
- Providing a REPL
- Providing a pluggable way to handle uncaught exceptions

The path through HLL::Compiler

`command_line` parses the arguments, then invokes `command_eval`

`command_eval` works out, based on the arguments, if we should load source files from disk, obtain source from `-e` or enter the REPL. The paths invoke a range of methods, but all converge back in `eval`.

`eval` calls `compile` to compile the code, then invokes it

`compile` loops through the stages, passing the result of the previous one as the input to the next one

A simplified version of compile

Big takeaway: stages are methods on the compiler object or a backend object.

```
method compile($source, :$from, *%adverbs) {
    my $target := nqp::lc(%adverbs<target>);
    my $result := $source;
    for self.stages() {
        if nqp::can(self, $_) {
            $result := self."$_"($result, |%adverbs);
        }
        elsif nqp::can($!backend, $_) {
            $result := $!backend."$_"($result, |%adverbs);
        }
        else {
            nqp::die("Unknown compilation stage '$_'");
        }
        last if $_ eq $target;
    }
    return $result;
}
```

Stage management

It's possible for compilers to insert extra stages into the pipeline. For example, Rakudo inserts its optimizer.

```
$comp.addstage('optimize', :after<ast>);
```

Then, in `Perl6::Compiler`, it provides an `optimize` method:

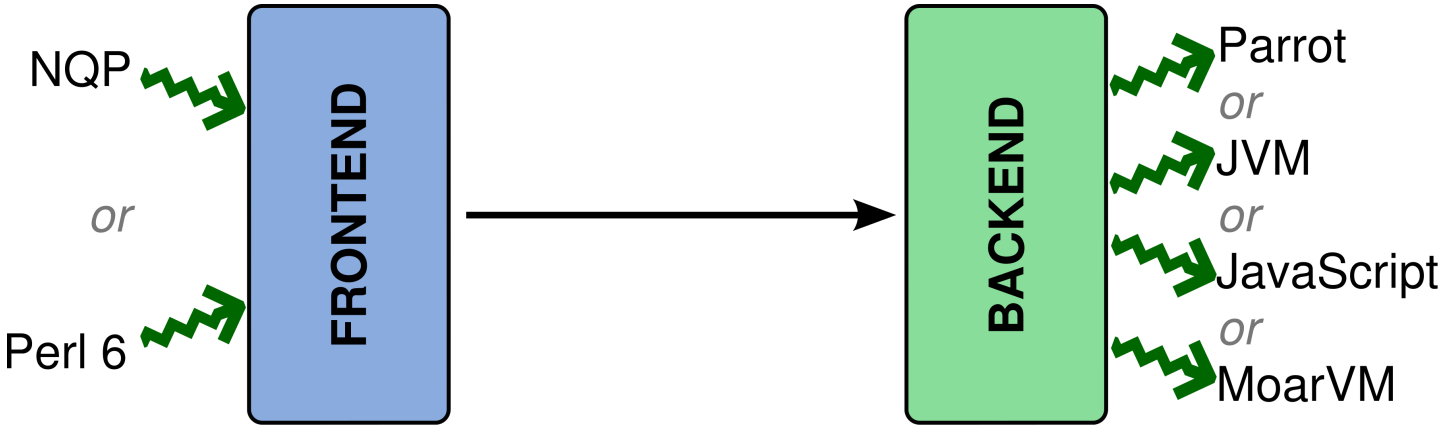
```
method optimize($ast, *%adverbs) {
    %adverbs<optimize> eq 'off' ??
        $ast !!
        Perl6::Optimizer.new.optimize($ast, |%adverbs)
}
```

Frontends and backends

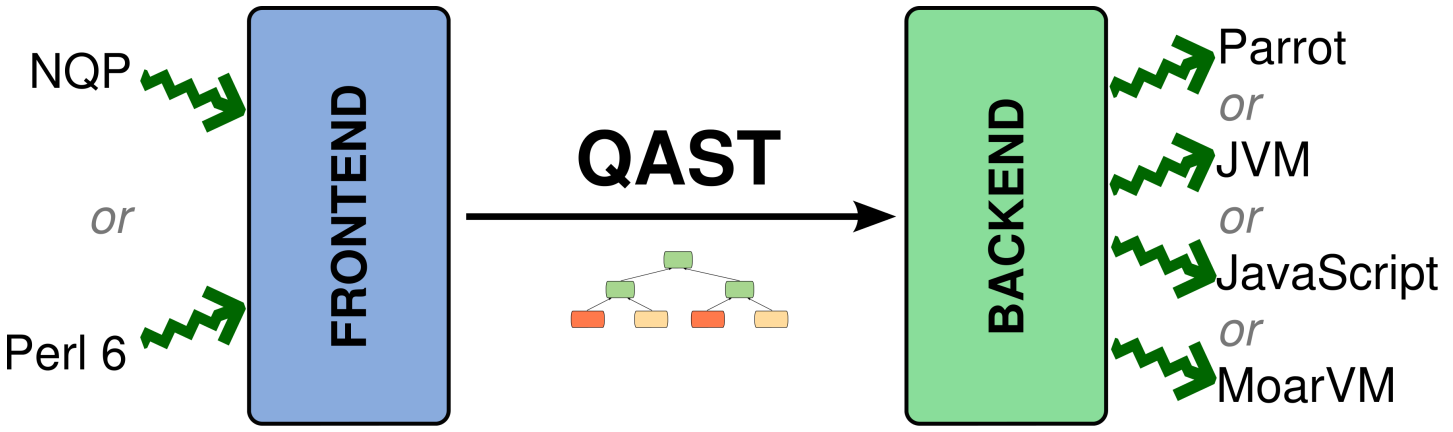
Earlier, we saw that `compile` looks for stage methods on the current compiler object, then on a backend object.

The **compiler object** is **about the language** that we are compiling (NQP, Perl 6, etc.) We collectively call these stages the **frontend**.

The **backend object** is **about the target VM** that we want to produce code for (Parrot, JVM, MoarVM, etc.) It is not tied to any particular language. We collectively call these stages the **backend**.



Frontends, backends, and the QAST between them



The last stage in the front end always gives a **QAST tree**, and the first stage in a backend always expects one.

A **cross-compiler setup** simply has a backend different from the current VM we are running on.

Parsing in NQP

The parse stage invokes `parse` on the language's grammar (for our case, NQP::Grammar), passing the source code and `NQP::Actions`. It may also turn on tracing.

```
method parse($source, *%adverbs) {
    my $grammar := self.parsegrammar;
    my $actions;
    $actions    := self.parseactions unless %adverbs<target> eq 'parse';
    $grammar.HOW.trace-on($grammar) if %adverbs<rxtrace>;
    my $match   := $grammar.parse($source, p => 0, actions => $actions);
    $grammar.HOW.trace-off($grammar) if %adverbs<rxtrace>;
    self.panic('Unable to parse source') unless $match;
    return $match;
}
```

NQP::Grammar.TOP (1)

As in the grammars we already saw, execution starts in `TOP`. In NQP, we find it's actually a `method`, not a `token` or `rule`!

```
method TOP() {
    # Various things we'll consider in a moment.
    ...

    # Then delegate to comp_unit
    self.comp_unit;
}
```

This is actually OK, so long as it ultimately returns a `Cursor` object. And since `comp_unit` will return one, it all works out just fine.

It's a method as it doesn't do any parsing, just setup work.

NQP::Grammar.TOP (2)

The first thing that `TOP` does is set up a **language braid**.

```
my %*LANG;
%*LANG<Regex>           := NQP::Regex;
%*LANG<Regex-actions>   := NQP::RegexActions;
%*LANG<MAIN>            := NQP::Grammar;
%*LANG<MAIN-actions>    := NQP::Actions;
```

While we didn't make the distinction too carefully earlier, when we start to parse a `token`, `rule` or `regex`, we're actually **switching language**. A block nested inside of a regex will in turn switch back to the main language.

Thus, `%*LANG` keeps track of the current set of languages we're using in the parse, entangled like strands of beautifully braided hair. Rakudo has a third language in its braid: `Q`, the quoting language.

NQP::Grammar.TOP (3)

Next, the current set of **meta-objects** are set up. Each package declarator (`class`, `role`, `grammar`, `module`, `knowhow`) is mapped to an object that implements this kind of package.

```
my %*HOW;
%*HOW<knowhow>       := nqp::knowhow();
%*HOW<knowhow-attr>  := nqp::knowhowattr();
```

We only have one of those built-in - `knowhow`. It supports having methods and attributes, but not role composition or inheritance.

All the more interesting meta-objects are written in terms of KnowHOW, and are in a module that is loaded at startup. We'll return to this topic in much more detail in day 2.

NQP::Grammar.TOP (4)

Next, an `NQP::World` object is created. This represents the **declarative aspects** of a program (such as class declarations).

```
my $file := nqp::getlexdyn('$?FILES');
my $source_id := nqp::sha1(self.target()) ~
    (%*COMPILING<%?OPTIONS><stable-sc> ?? '' !! '-' ~ ~nqp::time_n());
my $*W := nqp::isnull($file) ??
    NQP::World.new(:handle($source_id)) !!
    NQP::World.new(:handle($source_id), :description($file));
```

Each compilation unit needs to have a **globally unique handle**. Since NQP bootstraps, we must usually base this off something more than the source, as otherwise the running compiler and the compiler being compiled would have overlapping handles!

(The `--stable-sc` option suppresses this for those needing to cross-compile NQP itself when porting to a new VM.)

NQP::Grammar.comp_unit

Next, we reach comp_unit. Here it is, stripped to the essentials.

```
token comp_unit {
    :my $*UNIT := $*W.push_lexpad($/);

    # Create GLOBALish - the current GLOBAL view.
    :my $*GLOBALish := $*W.pkg_create_mo(%*HOW<knowhow>,
                                         :name('GLOBALish'));
    {
        $*GLOBALish.HOW.compose($*GLOBALish);
        $*W.install_lexical_symbol($*UNIT, 'GLOBALish', $*GLOBALish);
    }

    # This is also the starting package.
    :my $*PACKAGE := $*GLOBALish;
    { $*W.install_lexical_symbol($*UNIT, '$?PACKAGE', $*PACKAGE); }

    <.outerctx>
    <statementlist>
    [ $ || <.panic: 'Confused'> ]
}
```

Dissecting comp_unit: scopes

There are various methods on `$*W` that related to scopes.

`$*W.push_lexpad($/)` is used to enter a new lexical scope, nested inside the current one. It returns a new `QAST::Block` object representing it.

`$*W.pop_lexpad()` is used to exit the current lexical scope, returning it.

`$*W.cur_lexpad()` is used to obtain the current scope.

As the names suggest, it's really just a stack.

Dissecting comp_unit: pkg_create_mo

Various methods on `NQP::World` are about packages. The `pkg_create_mo` method is used to create a type-object and meta-object representing a new package.

```
:my $*GLOBALish := $*W.pkg_create_mo(%*HOW<knowhow>, :name('GLOBALish'));
```

Due to **separate compilation**, everything in NQP starts out with a clean, empty view of `GLOBAL`, which we know as `GLOBALish`. These are unified at module load time.

The `pkg_create_mo` method is also used when dealing with keywords like `class`; in this case, it uses `%*HOW<class>`.

Dissecting comp_unit: install_lexical_symbol

Consider the following NQP snippet.

```
for @acts {
    my class Act { ... }
    my $a := Act.new(:name($_));
}
```

This lexical scope will clearly have the symbols `Act` and `$a`. However, they differ in an important way. `Act` is **fixed at compile time**, whereas `$a` is fresh each time around the loop. Symbols fixed at compile time in a lexical scope are installed with:

```
$*W.install_lexical_symbol($*UNIT, 'GLOBALish', $*GLOBALish);
```

Dissecting comp_unit: outer_ctx

The `outer_ctx` token looks like this:

```
token outerctx { <?> }
```

Huh? That's an "always succeed" assertion! The success, however, triggers the `outer_ctx` action method in `NQP::Actions`. Its most important line is:

```
my $SETTING := $*W.load_setting(
    %*COMPILING<%?OPTIONS><setting> // 'NQPCORE');
```

Which loads the NQP setting (`NQPCORE` by default), which in turn brings in the meta-objects (for `class`, `role`, etc.) and also types like `NQPMu` and `NQPArray`.

statementlist

The final thing the `comp_unit` token does is call `statementlist`, which does what the name suggests: parses a list of statements.

```
rule statementlist {
    | $
    | [<statement><.eat_terminator> ]*
}
```

The `eat_terminator` rule will match a semicolon, but also handles the use of a closing curly bracket to terminate a statement. Note the space after it is so a $<$`.ws`$>$ will be inserted.

statement

The `statement` rule expects to find either a `statement_control` (things like `if`, `while` and `CATCH` - this is a protoregex!) or an expression, which may be followed by a statement modifying condition and/or loop.

```
# **0..1 is like Perl 5 {0,1}; forces an array, which ? does not.
token statement {
    <!before <[\])}]> | $ >
    [
    | <statement_control>
    | <EXPR> <.ws>
        [
        || <?MARKED('endstmt')>
        || <statement_mod_cond> <statement_mod_loop>**0..1
        || <statement_mod_loop>
        ]**0..1
    ]
}
```

Aside: expression parsing

When we need to parse something like...

```
$x * -$grad + $c
```

...we need to pay attention to precedence. Trying to encode precedence as a bunch of rules calling each other would be terribly inefficient (one call for each level in the table!) and horrible to maintain.

Thus, `EXPR` actually calls into an **operator precedence parser**. Its implementation lives in `HLL::Grammar`, though we'll not look into that during this course; it's mildly terrifying and not something you're ever likely to need to change.

We will, however, see how to configure it later.

Terms

The operator precedence parser in `EXPR` is interested not only in operators, but also in the **terms** that the operators apply to. When it wants a term, it calls `termish`, which in turn calls `term`, another proto-regex.

For our `say('Hello, world')` example, the interesting term is the one that parses a function call:

```
token term:sym<identifier> {
    <deflongname> <?[(]> <args>  # <?[(]> is a lookahead
}
```

Now we're getting there! We just need to parse a name and an argument list.

deflongname

Parses an identifier (nothing clever here), followed by an optional colonpair (since things like `infix:<+>` are valid function names).

```
token deflongname {
    <identifier> <colonpair>**0..1
}
```

After we parse this, we (finally!) end up calling our first action method:

```
method deflongname($/) {
    make $<colonpair>
         ?? ~$<identifier> ~ ':' ~ $<colonpair>[0].ast.named
                ~ '<' ~ colonpair_str($<colonpair>[0].ast) ~ '>'
         !! ~$/;
}
```

Parsing arguments

Parses parentheses, then delegates off to the operator precedence parser again to parse either a single argument or a comma separated list of arguments.

```
token args {
    '(' <arglist> ')'
}

token arglist {
    <.ws>
    [
    | <EXPR('f=')>
    | <?>
    ]
}
```

`f=` indicates loosest allowed precedence level

Parsing values

Once again, the operator precedence parser calls `term`, and this time we end up reaching `term:sym<value>`.

```
token term:sym<value> { <value> }
token value {
    | <quote>
    | <number>
}
```

We have a quoted string, and thus end up in the `quote` protoregex, which in turn puts us in the candidate that parses a single quoted string.

```
token quote:sym<apos> { <?[']> <quote_EXPR: ':q'>  }
```

Actions all the way up!

We've actually bottomed out in the parsing of this statement now. However, we did not yet build any QAST nodes, which will indicate what the program should actually *do* when we run it.

The `quote_EXPR` action inherited from `HLL::Actions` does the hard work with regard to our quoted string:

```
method quote:sym<apos>($/) { make $<quote_EXPR>.ast; }
```

It produces a `QAST::SVal` node, which represents a string literal:

```
QAST::SVal.new( :value('Hello, world!') )
```

value actions

The `value` action method simply checks if we parsed a quote or a number, and then calls `make` with the AST of what we parsed.

```
method value($/) {
    make $<quote> ?? $<quote>.ast !! $<number>.ast;
}
```

And the value case of a `term` simply passes the value QAST on upwards:

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

arglist actions

The `arglist` action method makes a `QAST::Op` node that represents a `call`. The name will be attached later. It has to handle 3 cases: zero arguments (so `$<EXPR>` was not matched), a single argument, or a comma-separated list of arguments.

```
method args($/) { make $<arglist>.ast; }
method arglist($/) {
    my $ast := QAST::Op.new( :op('call'), :node($/) );
    if $<EXPR> {
        my $expr := $<EXPR>.ast;
        if nqp::istype($expr, QAST::Op) && $expr.name eq '&infix:<,>' {
            for $expr.list { $ast.push($_); }
        }
        else { $ast.push($expr); }
    }
    make $ast;
}
```

Function call actions

Now we have canonicalized the name and built a QAST node that represents a call. Therefore, the action method for a term that is a function call takes the call QAST node, sets its name (prepending an &) and passes it on up.

```
method term:sym<identifier>($/) {
    my $ast := $<args>.ast;
    $ast.name('&' ~ $<deflongname>.ast);
    make $ast;
}
```

Action methods higher up tend to combine together ASTs, generated by action methods lower in the parse, into bigger ASTs.

statement actions

Here's a simplified version. There's nothing really new to see in here. The real thing is only more complex because it's handling the statement modifying conditionals and loops.

```
method statement($/, $key?) {
    my $ast;
    if $<EXPR> { $ast := $<EXPR>.ast; }
    elsif $<statement_control> { $ast := $<statement_control>.ast; }
    else { $ast := 0; }
    make $ast;
}
```

The 0 simply means "we didn't find anything to parse here" - probably due to reaching the end of the source.

statementlist actions

Slightly simplified, but not much. A `QAST::Stmts` node indicates a set of things to do sequentially. We push the QAST node for each statements (in our case, one) onto it.

```
method statementlist($/) {
    my $ast := QAST::Stmts.new( :node($/) );
    if $<statement> {
        for $<statement> {
            $ast.push($_.ast);
        }
    }
    else {
        $ast.push(default_for('$'));
    }
    make $ast;
}
```

The `else` ensures we never produce an empty QAST::Stmts that would evaluate to `null`, but rather evaluate to `NQPMu`.

comp_unit actions

Finally, we reach the top! The `comp_unit` action method - again slightly simplified - pushes the `QAST::Stmts` on to the `QAST::Block` node, making these the statements to be executed by that block. Everything is then wrapped in a `QAST::CompUnit`, which also specifies which language the code is from.

```
method comp_unit($/) {
    # Push mainline statements into UNIT.
    my $mainline := $<statementlist>.ast;
    my $unit     := $*W.pop_lexpad();
    $unit.push($mainline);

    # Wrap everything in a QAST::CompUnit.
    make QAST::CompUnit.new(
        :hll('nqp'),
        # Much elided here; details later.
        $unit
    );
}
```

The end of the frontend

At this point, stage `parse` is completed! We have successfully executed the grammar, which produced us a `Match` object. And attached to this match object is a QAST tree that represents the semantics of the program.

Therefore, stage `ast` is rather straightforward.

```
method ast($source, *%adverbs) {
    my $ast := $source.ast();
    self.panic("Unable to obtain AST"
        unless $ast ~~ QAST::Node;
    $ast;
}
```

From here, we now enter the backend.

Aside: why interleave parsing and AST building?

One may wonder why parsing is not completed in full, and then an AST built. The answer is that in many cases, we need to evaluate pieces of the AST as we go about the parsing. For example, in:

```
BEGIN { say("OMG I'm alive!") }
1 2
```

That BEGIN block should actually run and produce its output, even though there is a syntax error right after it.

BEGIN-time things can have side-effects that actually influence the parse that follows on from them.

Code generation: a quick overview

The job of a backend is to take a QAST tree and produce code for the target runtime. This, once again, is organized by a set of stages. Their names will vary depending on if you are targetting Parrot, the JVM, MoarVM, etc.

We shall postpone looking at the details of any of those stages until later, and even then shall not dive too deeply into them. Much of the code that is contained within them is unlikely to change much in the future, and to make sense of much of it needs an intimate knowledge of the backend in question.

For now, we'll treat these stages as a magical black box. :-)

Building a tiny language from scratch

So, that's diving in to NQP. It can be a little overwhelming, and so it's good to practice on something a bit smaller.

Therefore, we'll build ourselves a couple of small compilers. I'll do one here, and you'll do one in the exercise.

The funny thing is that mine will be a Ruby subset.

The funnier thing is that yours will be a PHP subset.

We'll start by achieving "Hello, world", then in the next section - as we learn more about QAST - start to add language features.

Stubbing a compiler

Just subclass three things from the NQPHLL library.

```
use NQPHLL;

grammar Rubyish::Grammar is HLL::Grammar {
}

class Rubyish::Actions is HLL::Actions {
}

class Rubyish::Compiler is HLL::Compiler {
}

sub MAIN(*@ARGS) {
    my $comp := Rubyish::Compiler.new();
    $comp.language('rubyish');
    $comp.parsegrammar(Rubyish::Grammar);
    $comp.parseactions(Rubyish::Actions);
    $comp.command_line(@ARGS, :encoding('utf8'));
}
```

We already have a REPL

If we run the code from the previous slide, we find we already have ourselves a simple REPL (Read Eval Print Loop).

Predictably, trying to run things doesn't work:

```
> puts "Hello world"
Method 'TOP' not found for invocant of class 'Rubyish::Grammar'
```

Of course, that also tells us exactly what we should do next...

A basic grammar

Rubyish is line-oriented, so each statement is separated by a newline, and only horizontal whitespace is allowed between tokens.

```
grammar Rubyish::Grammar is HLL::Grammar {
    token TOP           { <statementlist> }

    rule statementlist { [ <statement> \n+ ]* }

    proto token statement {*}
    token statement:sym<puts> {
        <sym> <.ws> <?["]> <quote_EXPR: ':q'>
    }

    # Whitespace required between alphanumeric tokens
    token ws { <!ww> \h* || \h+ }
}
```

What have we now?

With this, we can now parse our simple program, but it fails when trying to obtain the AST:

```
> puts "Hello world"
Unable to obtain AST from NQPMatch
```

Which, again, tells us what we need to do next: actions!

Basic actions

```
class Rubyish::Actions is HLL::Actions {
    method TOP($/) {
        make QAST::Block.new( $<statementlist>.ast );
    }

    method statementlist($/) {
        my $stmts := QAST::Stmts.new( :node($/) );
        for $<statement> {
            $stmts.push($_.ast)
        }
        make $stmts;
    }

    method statement:sym<puts>($/) {
        make QAST::Op.new(
            :op('say'),
            $<quote_EXPR>.ast
        );
    }
}
```

It works!

Recall that the backends are language independent; they simply expect a QAST tree as input. And our actions produce one. As such, we now have a working compiler for our very, very simple language.

```
> puts "Hello world"
Hello World
```

We can also dump the AST:

```
- QAST::Block
  - QAST::Stmts puts \"Hello, world\"\n
    - QAST::Op(say)
      - QAST::SVal(Hello, world)
```

In summary. . .

We've walked our way through the flow of control from invoking NQP at the command line, seeing it parse our program, build up a QAST tree for it and pass it off to the backend for compilation.

We then used this same technology to build a tiny compiler up from scratch.

Since it's built on the same technology as NQP and Rakudo, it gets the same benefits. For example, out of the box our compiler already works both on Parrot and the JVM.

Exercise 4

In this exercise, you'll build the PHPish equivalent of my Rubyish.

The main difference is that the keyword you want is `echo`, and the lines are separated by semicolons rather than newline characters.

QAST

Between frontend and backend: the Q Abstract Syntax Tree

Digging deeper into QAST

We've already built some very simple QAST trees so far. However, they have barely scratched the surface of what is available in QAST.

In this section of the course, we will look at a much wider range of node types and the options they support.

To provide concrete examples, Rubyish will be extended to support a wider range of language features.

QAST::Node: children

All of the QAST node types inherit from the base class `QAST::Node`.

All QAST nodes support having **child nodes**. The initial set of child nodes may be passed as positional arguments to `new`. On any node, it's possible to:

```
my $first := $ast[0];       # get first child
$ast[0] := $child;          # set first child
$ast.push($child);          # push a child
$child := $ast.pop();       # pop a child
$ast.unshift($child);       # unshift a child
$child := $ast.shift();     # shift a child
@children := $ast.list();   # get underlying children list
```

QAST::Node: annotations

All QAST nodes can be given arbitrary **annotations** by using hash indexing on the node.

```
$var<used> := 1;
```

This can be very useful, but it's easy to overuse and create a tangled mess. Yes, I learned this the hard way.

All annotations can be obtained using the `hash` method:

```
my %anno := $var.hash();
```

QAST::Node: return type

There are two other important things you can do with a QAST node. All nodes can be annotated with the type that they will evaluate to.

```
$ast.returns($some_type);
```

Note that you **specify a type object** to represent the type, *not* a string name of the type! In some cases, the type set here is used in code generation (for example, natively typed variables get their native storage allocated by virtue of this).

This can also be set when creating a node in the first place:

```
QAST::Var.new( ..., :returns(int) )
```

QAST::Node: node

One other important thing we may wish to do is associate a QAST node with a source location. This information is persisted by the backend code generation, such that it can be used to produce meaningful backtraces when runtime errors occur.

The `node` method expects to be given a match object:

```
$ast.node($/);
```

Once again, it can be specified (typically on QAST::Stmts nodes) as a node constructor argument.

```
my $ast := QAST::Stmts.new( :node($/) );
```

The top of the tree

At the top level, a QAST tree must have either a `QAST::CompUnit` or a `QAST::Block`.

A `QAST::CompUnit` represents a compilation unit. It should have a single child which is a `QAST::Block`. However, it can also specify many other bits of configuration; we'll see more later.

A `QAST::Block` represents a lexical scope. Whenever one `QAST::Block` is nested inside another, it represents a nested lexical scope that can see the variables in the outer one. Combined with cloning, this also facilitates closure semantics.

Literals: QAST::IVal, QAST::NVal and QAST::SVal

These three node types represent integer, floating point and string literals. If we update our grammar to parse different kinds of value:

```
proto token value {*}
token value:sym<string>  { <?["]> <quote_EXPR: ':q'> }
token value:sym<integer> { '-'? \d+ }
token value:sym<float>   { '-'? \d+ '.' \d+ }
```

Then we can write the actions as:

```
method value:sym<string>($/) {
    make $<quote_EXPR>.ast;
}
method value:sym<integer>($/) {
    make QAST::IVal.new( :value(+$/.Str) )
}
method value:sym<float>($/) {
    make QAST::NVal.new( :value(+$/.Str) )
}
```

Trying our literals

After a small tweak to `puts` parsing...

```
token statement:sym<puts> {
    <sym> <.ws> <value>
}
```

...and the matching tweak in the actions, we can now do:

```
> puts 42
42
> puts 0.999
0.999
> puts "It's not a bacon tree, it's a hambush!"
It's not a bacon tree, it's a hambush!
```

Operations: QAST::Op

The `QAST::Op` node is the gateway to an incredible number of operations. They are the same ones available through the `nqp::op(...)` syntax.

Typically, a QAST::Op node looks something like this:

```
QAST::Op.new(
    :op('add_n'),
    $left_child_ast,
    $right_child_ast
)
```

The operation is specified with the `:op(...)` named argument, and operands are the node's children.

Parsing some mathematical operators (1)

Let's add addition, subtraction, multiplication and division. For these, we need to set up the operator precedence parser, configuring two precedence levels.

```
INIT {
    # Steal precedence level names from Perl 6 grammar
    Rubyish::Grammar.O(':prec<u=>, :assoc<left>', '%multiplicative');
    Rubyish::Grammar.O(':prec<t=>, :assoc<left>', '%additive');
}
```

Note that the `O` method we call here is inherited from `HLL::Grammar`. The first argument specifies precedence level and associativity. The second then saves this particular configuration by name, so we can refer to it when we declare operators.

Parsing some mathematical operators (2)

With the precedence levels in place, we can add some operators into the grammar. This is done by adding them to the `infix` protoregex, which we inherit from `HLL::Grammar`.

```
token infix:sym<*> { <sym> <O('%multiplicative, :op<mul_n>')> }
token infix:sym</> { <sym> <O('%multiplicative, :op<div_n>')> }
token infix:sym<+> { <sym> <O('%additive, :op<add_n>')> }
token infix:sym<-> { <sym> <O('%additive, :op<sub_n>')> }
```

The `:op<...>` syntax instructs the `EXPR` action method we inherit from `HLL::Actions` to construct a `QAST::Op` node of that op for us!

Terms

We are nearly ready to use the operator precedence parser, but not quite. We must also instruct it on how to **obtain a term**. We inherit a `term` protoregex from `HLL::Grammar`, and so need only add candidates for it.

For us, that means a candidate for a term that is a value:

```
token term:sym<value> { <value> }
```

And the matching action method:

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

Wiring it all up

The final thing we need to do is update the grammar rule for `puts`:

```
token statement:sym<puts> {
    <sym> <.ws> <EXPR>
}
```

Along with the action method:

```
method statement:sym<puts>($/) {
    make QAST::Op.new(
        :op('say'),
        $<EXPR>.ast
    );
}
```

Trying out our operators

Basic arithmetic now works, and precedence is done correctly.

```
> puts 10 * 9 + 1
91
```

We may also inspect the AST to see the QAST::Op nodes:

```
- QAST::Block
  - QAST::Stmts puts 10 * 9 + 1\n
    - QAST::Op(say)
      - QAST::Op(add_n &infix:<+>) +
        - QAST::Op(mul_n &infix:<*>) *
          - QAST::IVal(10)
          - QAST::IVal(9)
        - QAST::IVal(1)
```

Sequencing: QAST::Stmts and QAST::Stmt

There are two node types that represent running each of their children in order

`QAST::Stmts` does, quite literally, nothing more than that

`QAST::Stmt` has the added effect of stating that any temporaries that are created during code generation will not be needed beyond the end of this node's execution

Generally, using them in places where the language user would think of having a set of statements vs. a single statement makes sense.

Block structure

A common idiom, though in no way enforced, is for a `QAST::Block` to have two `QAST::Stmts` nodes within it

The first one is used to hold **declarations**, for example of variables or of nested routines

The second one is used to hold **the statements parsed by statementlist** for that block

This idiom is used in both NQP and Rakudo; for example:

```
$block[0].push(QAST::Var.new(:name<$/>, :scope<lexical>, :decl<var>));
```

Variables

It's time to add variables to Rubyish! In Rubyish, variables aren't declared explicitly. Instead, they are declared in the current scope on their first assignment.

First, let's add a precedence level for assignment:

```
Rubyish::Grammar.O(':prec<j=>, :assoc<right>',  '%assignment');
```

And parse the assignment operator, using the `bind` NQP operation which will bind the expression on the right to a variable on the left:

```
token infix:sym<=> { <sym> <O('%assignment, :op<bind>')> }
```

Expressions as statements

One thing you may recall from the NQP grammar is that an expression was also a valid statement. We need to do that in Rubyish too.

This means adding to the grammar:

```
token statement:sym<EXPR> { <EXPR> }
```

And the actions:

```
method statement:sym<EXPR>($/) { make <EXPR>.ast; }
```

Identifier parsing

For now, we'll treat all identifiers as if they were variables. We parse them like this:

```
token term:sym<ident> {
    :my $*MAYBE_DECL := 0;
    <ident>
    [ <?before \h* '=' [\w | \h+] { $*MAYBE_DECL := 1 }> || <?> ]
}
```

Notice how this looks ahead to see if we can find an assignment operator with whitespace around it or an identifier right after it (must not treat == as if it were an assignment!)

A dynamic variable is used to convey if an assignment happens, which may mean we have a declaration.

Identifier actions

Here is a first, cheating attempt at the actions for an identifier.

```
method term:sym<ident>($/) {
    if $*MAYBE_DECL {
        make QAST::Var.new( :name(~$<ident>), :scope('lexical'),
                            :decl('var') );
    }
    else {
        make QAST::Var.new( :name(~$<ident>), :scope('lexical') );
    }
}
```

This does allow us to run:

```
a = 7
b = 6
puts a * b
```

The problem

Things come unstuck fairly quickly, unfortunatley. Every assignment is now taken to be a declaration. Thus:

```
a = 1
puts a
a = 2
puts a
```

Fails with:

```
Error while compiling block: Error while compiling op bind:
Lexical 'a' already declared
```

The symbol table

Every `QAST::Block` comes with a symbol table that can be used to store extra information about the symbols declared within it.

Really, it's just a hash of hashes, the first hash keyed on the symbol and the inner hashes storing whatever information we wish.

We can add to or update a symbol's entries by doing:

```
$block.symbol($ident, :declared(1));
```

We can get hold of the current information held on a symbol by doing:

```
my %sym := $block.symbol($ident);
```

Next challenge: keeping track of the block

We need to have access to the current block we are declaring things in before we can use `symbols`. This is most easily handled by placing it in a dynamic variable, creating it in the `TOP` grammar rule:

```
token TOP {
    :my $*CUR_BLOCK := QAST::Block.new(QAST::Stmts.new());
    <statementlist>
    [ $ || <.panic('Syntax error')> ]
}
```

With the `TOP` action method becoming:

```
method TOP($/) {
    $*CUR_BLOCK.push($<statementlist>.ast);
    make $*CUR_BLOCK;
}
```

Using symbol

Now, we can use `symbol` to track what was already declared and not re-declare it.

```
method term:sym<ident>($/) {
    my $name := ~$<ident>;
    my %sym  := $*CUR_BLOCK.symbol($name);
    if $*MAYBE_DECL && !%sym<declared> {
        $*CUR_BLOCK.symbol($name, :declared(1));
        make QAST::Var.new( :name($name), :scope('lexical'),
                            :decl('var') );
    }
    else {
        make QAST::Var.new( :name($name), :scope('lexical') );
    }
}
```

Other scopes

The `QAST::Var` node isn't just for lexical scoping. The available scopes are:

```
lexical         visible to nested blocks
local           like lexical, but not visible to nested blocks
contextual      dynamically scoped lookup of a lexical
attribute       object attribute (children: invocant, package)
positional      array indexing (children: array, index)
associative     hash indexing (children: hash, key)
```

Note that only the first 3 make sense as a declaration. Also note that Rakudo does not use the last 2 (its array and hash handling is factored differently), though NQP does.

Routines

To demonstrate lexical scoping a little more, let's add routines. The syntax for declaring and calling them is as follows:

```
def greet
    puts "hello"
end
greet()
```

We'll keep things simple by not handling the other forms of calling.

Parsing a routine declaration

Nothing especially new in here. We take care to start a new lexical scope, so any declarations made will not polluate the surrounding scope. The split is so the first token's action method can see the `$*CUR_BLOCK` to install into.

```
token statement:sym<def> {
    'def' \h+ <defbody>
}
rule defbody {
    :my $*CUR_BLOCK := QAST::Block.new(QAST::Stmts.new());
    <ident> \n
    <statementlist>
    'end'
}
```

NQP and Rakudo do pretty much the same, the only difference being that they abstract the pushing/popping of the blocks and keep a stack of them.

Parsing calls

A call is an identifier followed by some parentheses. We also take care to avoid keywords.

```
token term:sym<call> {
    <!keyword>
    <ident> '(' ')'
}
```

The <!keyword> is also applied to term:sym<ident>.

Actions for routine declaration

defbody finishes up the QAST::Block, and it is installed as a lexical by statement:sym<def>.

```
method statement:sym<def>($/) {
    my $install := $<defbody>.ast;
    $*CUR_BLOCK[0].push(QAST::Op.new(
        :op('bind'),
        QAST::Var.new( :name($install.name), :scope('lexical'),
                       :decl('var') ),
        $install
    ));
    make QAST::Op.new( :op('null') );
}
method defbody($/) {
    $*CUR_BLOCK.name(~$<ident>);
    $*CUR_BLOCK.push($<statementlist>.ast);
    make $*CUR_BLOCK;
}
```

Invocation

Calling is an operation, and therefore done with `QAST::Op`. By default, the name of the thing to call - which will be resolved lexically - is specified in the `name` named argument.

```
method term:sym<call>($/) {
    make QAST::Op.new( :op('call'), :name(~$<ident>) );
}
```

Any case where `name` is not specified will take the first child of the node as the thing to invoke. Thus we could have written:

```
method term:sym<call>($/) {
    make QAST::Op.new(
        :op('call'),
        QAST::Var.new( :name(~$<ident>), :scope('lexical')
    );
}
```

Parameters and arguments

Argument and parameter handling involve no node types we haven't seen before. Arguments are just children to the `QAST::Op` call node, and parameters are simply `QAST::Var` nodes with the `decl` set to `param`.

First, let's add parsing for parameters.

```
rule defbody {
    :my $*CUR_BLOCK := QAST::Block.new(QAST::Stmts.new());
    <ident> <signature>? \n
    <statementlist>
    'end'
}
rule signature {
    '(' <param>* % [ ',' ] ')'
}
token param { <ident> }
```

Parameter actions

The `param` action method looks like this:

```
method param($/) {
    $*CUR_BLOCK[0].push(QAST::Var.new(
        :name(~$<ident>), :scope('lexical'), :decl('param')
    ));
    $*CUR_BLOCK.symbol(~$<ident>, :declared(1));
}
```

Interestingly, it never does `make`. This may seem odd at first, as the action methods elsewhere have done so. But it has no reason to; what we really wish to do is install the declared parameter into the current block. It's easier to just get at it contextually.

Passing arguments

Here's a quick and easy way to parse the arguments:

```
token term:sym<call> {
    <!keyword>
    <ident> '(' :s <EXPR>* % [ ',' ] ')'
}
```

Then we update the actions:

```
method term:sym<call>($/) {
    my $call := QAST::Op.new( :op('call'), :name(~$<ident>) );
    for $<EXPR> {
        $call.push($_.ast);
    }
    make $call;
}
```

So far...

So far, we have used the following QAST node types:

```
QAST::Block     A lexical scope
QAST::Stmts     A sequence of things to execute
QAST::Stmt      As above, but also a temporaries boundary
QAST::Op        An operation of some kind
QAST::Var       A variable or parameter usage/declaration
QAST::IVal      Integer literal
QAST::NVal      Floating point literal
QAST::SVal      String literal
```

We'll consider a few more node types today; we'll put off some (`QAST::WVal` and `QAST::Regex`) until tomorrow.

Block references with QAST::BVal

A QAST::Block should only ever appear once inside a QAST tree. Where it is placed defines its lexical scope.

So what if you want to **refer to a `QAST::Block`** elsewhere in the tree? That's what a `QAST::BVal`, short for Block Value, is for. For example, it is used when emitting code to make the CORE setting be a program's outer lexical scope.

```
my $set_outer := QAST::Op.new(
    :op('forceouterctx'),
    QAST::BVal.new( :value($*UNIT) ),
    QAST::Op.new(
        :op('callmethod'), :name('load_setting'),
        # stuff left out here
    ));
```

Boxed vs. unboxed, void vs. non-void context

As the backend code generation takes place, it may need to box and/or unbox things, or it may determine that something will be in a void (sink) context.

While it can reliably produce working code, it may not be efficient. Consider the integer constant handling here:

```
my int $x = 42;      # Needs an unboxed native int
my $x = 42;          # Needs a boxed Int object
```

When we write the action method for integer literals, we have a dilemma. Should we emit a `QAST::IVal`, which will have to be boxed in the second case? Or should we put an Int constant 42 into the constants pool and reference it with a `QAST::WVal` (more on this node type tomorrow)?

QAST::Want to the rescue

Rather than choosing, we can present both options, and let the code generator pick whichever will be most efficient. This is done through the `QAST::Want` node.

```
QAST::Want.new(
    QAST::WVal.new( :value($boxed_constant) ),
    'Ii', QAST::IVal.new( :value($the_value) )
)
```

The first child is the default thing. It is followed by a set of selectors for different contexts we may be in.

```
Ii      native integer
Nn      native floating point number
Ss      native string
v       void
```

The backend escape hatch: QAST::VM (1)

Sometimes, there's a need to **do things conditionally by backend**, or to do some **VM-specific** operation. The QAST::VM node handles this need.

For example, here is some code from NQP that loads the NQP module loader. It needs to know what filename to look for by backend.

```
QAST::Op.new(
    :op('loadbytecode'),
    QAST::VM.new(
        :parrot(QAST::SVal.new( :value('ModuleLoader.pbc') )),
        :jvm(QAST::SVal.new( :value('ModuleLoader.class') ))
    ))
```

If there's no applicable option for the current backend, an exception will be thrown by the code generator.

The backend escape hatch: QAST::VM (2)

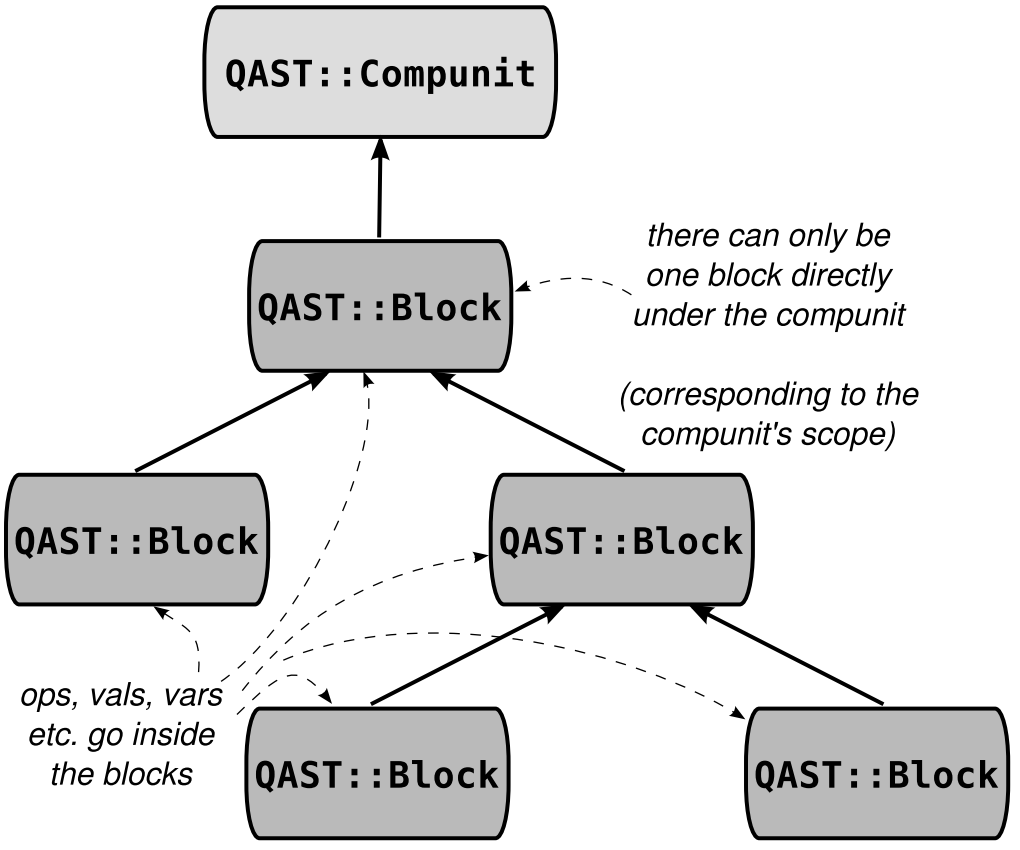
The `QAST::VM` node type is also behind the `pir::op_SIG(...)` syntax that is available in NQP and Rakudo. Here is how `pir::op` is parsed and implemented in NQP.

```
token term:sym<pir::op> {
    'pir::' $<op>=[\w+] <args>**0..1
}

method term:sym<pir::op>($/) {
    my @args := $<args> ?? $<args>[0].ast.list !! [];
    my $pirop := ~$<op>;
    $pirop := join(' ', nqp::split('__', $pirop));
    make QAST::VM.new( :pirop($pirop), :node($/), |@args );
}
```

At the top: QAST::CompUnit

QAST trees produced by Rakudo and NQP have a `QAST::CompUnit` at the top.



What we can do with QAST::CompUnit

Here's a look at some of what `QAST::CompUnit` can do (we'll see it again tomorrow).

```
my $compunit := QAST::CompUnit.new(
    # Set the language this contains.
    :hll('nqp'),

    # What to do if the compilation unit is loaded as a module.
    :load(QAST::Op.new(
        :op('call'),
        QAST::BVal.new( :value($unit) )
    )),

    # What to do if the compilation unit is invoked as the main,
    # top-level program.
    :main(...),

    # 1 child, which is the top-level QAST::Block
    $unit
);
```

Exercise 5

In this exercise, you'll add a few features to PHPish, in order to explore the QAST nodes we've been studying.

Looking at the NQP grammar and actions to understand how they work - or even stealing from them wholesale and cargo-culting - is encouraged! :-)

Exploring nqp:: ops

Learn all the operations!

Just a glimpse

There are literally hundreds of available `nqp::ops`. They range from arithmetic to string manipulation, from flow control (like looping) to type creation.

We've already seen some of the operations. Tomorrow, we'll see a bunch more as we look at 6model and serialization contexts, which have a bunch of `nqp::ops` associated with them.

In this section, we'll take an overview of "the rest". The overview is not exhaustive, as that would be exhausting.

Remember they can be used in the `nqp::op` form *or* in a `QAST::Op` node, so this knowledge is reusable for both!

Arithmetic

These come in native integer form:

```
add_i   sub_i   mul_i   div_i   mod_i
neg_i   abs_i
```

As well as native float form:

```
add_n   sub_n   mul_n   div_n   mod_n
neg_n   abs_n
```

To help with implementing rationals, we also have:

```
lcm_i   gcd_i
```

Numerics

The basic stuff:

```
pow_n       ceil_n      floor_n
ln_n        sqrt_n      log_n
exp_n       isnanorinf  inf
neginf      nan
```

Trigometric:

```
sin_n   asin_n  cos_n   acos_n  tan_n
atan_n  atan2_n sinh_n  cosh_n  tanh_n
sec_n   asec_n  sech_n
```

Relational

For comparing native integers, native floats and native strings (the code generator will unbox as needed). For example, the native integer forms are:

```
cmp_i       compare; returns -1, 0, or 1
iseq_i      non-zero if equal
isne_i      non-zero if non-equal
islt_i      non-zero if less than
isle_i      non-zero if less than or equal to
isgt_i      non-zero if greater than
isge_i      non-zero if greater than or equal to
```

The `_n` and `_s` forms all exist too.

Array operations

There are various operations for manipulating arrays:

`atpos`	`atpos_i`	`atpos_n`	`atpos_s`
`bindpos`	`bindpos_i`	`bindpos_n`	`bindpos_s`
`push`	`push_i`	`push_n`	`push_s`
`pop`	`pop_i`	`pop_n`	`pop_s`
`shift`	`shift_i`	`shift_n`	`shift_s`
`unshift`	`unshift_i`	`unshift_n`	`unshift_s`
`splice`	`existspos`	`elems`	`setelems`

Note that the natively typed versions are **not coercive**, but only work on a natively typed array.

Hash operations

Don't look too different from the array operations.

```
atkey        atkey_i     atkey_n     atkey_s
bindkey      bindkey_i   bindkey_n   bindkey_s
existskey    deletekey   elems
```

These all assume that the keys are strings; any non-string key will be coerced to a string first.

Aside: Perl 6 use of the array/hash ops

In Perl 6, something like:

```
@a[0] = 42;
```

Actually uses `atpos` to get the scalar container bound into the underlying array storage, and then assigns to that container. `bindpos` is only used for doing:

```
@a[0] := 42;
```

Also, you never do this directly on a Perl 6 `Array` or `Hash` object. These objects *contain* a lower-level array or hash as an attribute, and methods use this ops on that.

Creating lists and hashes

The `nqp::list` op creates a (low level) array with the elements passed to it. As a result, it is a variable argument op.

```
nqp::list($foo, $bar, $baz)
```

Natively typed lists can be created with `list_i`, `list_n` and `list_s`.

There is a similar `nqp::hash`, which expects a key, a value, ...

```
nqp::hash('name', $name, 'age', $age)
```

Finally, `islist` and `ishash` tell you if something is a low-level array or hash.

String

String operations are mostly named as Perl 6 does.

```
chars       uc          lc          x
concat      chr         join        split
flip        replace     substr      ord
index       rindex      codepointfromname
```

There are also operations for checking character class membership. These are mostly emitted when compiling regexes or in the regex-related classes, but may be used elsewhere. They are:

```
nqp::iscclass(class, str, index)
nqp::findcclass(class, str, index, limit)
nqp::findnotcclass(class, str, index, limit)
```

Where `class` is one of the `nqp::const::CCLASS_*`.

Conditionals

The `if` and `unless` ops expect two or three children: a condition, a "then", and an optional "else". Note that `elsif` in NQP and Perl 6 is compiled by nesting `if` `QAST::Op` nodes.

```
# AST for '$/.ast ?? $/.ast !! $/.Str'
QAST::Op.new(
    :op('if'),
    QAST::Op.new(
        :op('callmethod'), :name('ast'),
        QAST::Var.new( :name('$/'), :scope('lexical') )
    ),
    QAST::Op.new(
        :op('callmethod'), :name('ast'),
        QAST::Var.new( :name('$/'), :scope('lexical') )
    ),
    QAST::Op.new(
        :op('callmethod'), :name('Str'),
        QAST::Var.new( :name('$/'), :scope('lexical') )
    )
)
```

Conditionals and arity-1 blocks

Both NQP and Perl 6 supporting things like:

```
if %core_ops{$name} -> $mapper {
    return $mapper($qastcomp, $op);
}
```

This evaluates `%core_ops{$name}`, then passes it in to `$mapper` if it's a truthy value.

At QAST level, this is represented by the second child of the `if` op being a `QAST::Block` whose `arity` is set to a non-zero value.

Loops

There are four related loop constructs:

	Loop while true	Loop while false
Condition, then body	while	until
Body, then condition	repeat_while	repeat_until

They take two or three children:

- The condition
- The body
- Optionally, something to do after the body

If a `redo` control exception is thrown, the second child is re-evaluated. The third is only evaluated after any `redo`s have taken place. It's used by the Perl 6 (C-style) `loop` construct.

Loop example

The Perl 6 `let` and `temp` keywords keep a list of containers and their original values (a container, a value, etc.) This is the loop that goes through this list at block exit to do restoration.

```
$phaser_block.push(QAST::Op.new(
    :op('while'),
    QAST::Var.new( :name($value_stash), :scope('lexical') ),
    QAST::Op.new(
        :op('p6store'),
        QAST::Op.new(
            :op('shift'),
            QAST::Var.new( :name($value_stash), :scope('lexical') )
        ),
        QAST::Op.new(
            :op('shift'),
            QAST::Var.new( :name($value_stash), :scope('lexical') )
        ))));
```

Other control structures

There are three others that are worth knowing about:

- `for` takes two children, something iterable (typically a low level array or list) and a block. It invokes the block for each thing in the iterable. Used in NQP only; Rakudo does iterators completely differently.
- `ifnull` takes two children. It evaluates the first. If it is not null, it just produces this value. If it *is* null, it evaluates the second child.
- `defor` is the same as `ifnull`, but considers definedness rather than nullness

Throwing exceptions

There are various operations for creating and throwing an exception:

```
newexception    Creates a new, empty, exception object
setextype       Sets the exception category (nqp::const::CONTROL_*)
setmessage      Sets the exception message (string)
setpayload      Sets the exception payload (object)
throw           Throws an exception object
die             Makes/throws an exception with a string message
```

There is an easier way to throw some of the common control exceptions:

```
QAST::Op.new( :op('control'), :name('next') )
```

Other valid names here are are `redo` and `last`.

Handling exceptions

The `handle` op is used to express exception handling. The first child is the code to protect with the handler(s). The handlers are then specified as a string specifying the kind of exception to handle, followed by the QAST to run to handle it.

NQP and Rakudo keep a per-block `%*HANDLERS`, and build the `handle` op out of it when the block is fully parsed.

```
my $ast := $<statementlist>.ast;
if %*HANDLERS {
    $ast := QAST::Op.new( :op('handle'), $ast );
    for %*HANDLERS {
        $past.push($_.key);
        $past.push($_.value);
    }
}
```

Working with exception objects

Within a handler, the following operations can be used. Except the first, they all take an exception object.

```
exception       Gets the current exception object
getextype       Gets the exception category (nqp::const::CONTROL_*)
getmessage      Gets the exception message (string)
getpayload      Gets the exception payload (object)
rethrow         Re-throws the exception
resume          Resumes the exception, if possible
```

Finally, there are two more operations that relate to backtraces; `backtrace` returns an array of hashes, each hash describing a backtrace entry, while `backtracestrings` simply returns an array of strings describing the entries.

Context introspection

Various operations are available to introspecting the symbols in a lexical scope, or walk the dynamic (caller) or static (lexical) chain of scopes. They are typically used to implement features such as the `CALLER` and `OUTER` pseudo-packages in Perl 6.

```
ctx             get an object representing the current context
ctxouter        take a context and return its outer context, or null
ctxcaller       take a context and return its caller contxt, or null
ctxlexpad       take a context and return its lexpad
curlexpad       get the current lexpad
lexprimspec     given a lexpad and a name, get the name's primitive type
```

The lexpad itself can be used with the appropriate hash operations (`atkey`, `bindkey`) to manipulate the symbols contained within it.

Big integers

Perl 6 needs big integer support for its `Int` type. Therefore, it is provided for in the NQP operations. The big integer operations are only valid on an object with the `P6bigint` representation (more on representations tomorrow) or something that boxes it.

Those operations that have a big integer result differ from their native relatives by taking an extra operand, which is the type object for the result. The following `multi`s from the Perl 6 setting illustrate this.

```
multi infix:<+>(Int:D \a, Int:D \b) returns Int:D {
    nqp::add_I(nqp::decont(a), nqp::decont(b), Int);
}
multi infix:<+>(int $a, int $b) returns int {
    nqp::add_i($a, $b)
}
```

The `_I` suffix is used for big integer ops.

Exercise 6

If time allows, you can explore some of the nqp::ops by adding support to PHPish for (take them in order, or pick those you'd find most fun):

- Basic numeric relational operators (<, >, ==, etc.)
- `if`/`else if`/`else`
- `while` loops

See the exercise sheet for a few hints.

That's all for today

Today, we've covered a lot of ground, starting out with the NQP language and then building up to how it can be used to implement a simple compiler.

That's a good start, but we're still missing several very important pieces that both NQP and Rakudo depend heavily on. This includes objects and the concept of serialization contexts. We'll take these on tomorrow.

Any more questions?