Rakudo and NQP Internals

The guts tormented implementers made

Jonathan Worthington

© Edument AB

September 16, 2013

EDUMENT

Development and Mentorship

Course overview - Day 2

Welcome back. Today, we will cover the following topics:

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

6model

Ingredients for cooking up object systems

What is 6model?

6model provides a set of **primitives for building type and object systems**.

Rakudo's classes, roles, enumerations and subset types are all assembled out of these primitives. The same is true of NQP, although NQP's object system is simpler, just providing classes and roles.

These primitives have been implemented on Parrot and the JVM. MoarVM provides them also, but it goes a step further, making 6model *the* object system for the VM.

The primitives are more primitive than you may first imagine. For example, 6model has **no built-in concept of inheritance or role composition**. These are built at a higher level.

Object = behavior + state

Whatever language you look at, you'll find that objects always have:

- A mechanism for making object instances, which can have **state**
- A mechanism for taking an object and a name, locating a **behavior** with that name, and (provided it exists) invoking it

The state may be shaped by classes or freeform. The behavior may be directly attached to the object, attached at a per-class level, or located through a multiple dispatch mechanism.

But there will always be **state** and **behavior**.

Types

Most languages also have some notion of **type**. Typically, types fall into relationships with each other. For example, given:

```
class Event {
    has $.name;
    has $.start-date;
    has $.days;
}
class Hackathon is Event {
    has $.topic;
    has @.hackers;
}
```

We can say that `Hackathon` is a subtype of `Event`.

Package kinds

Not only do we have different types, we have **different kinds of type**. In Perl 6, these correspond to different **package and type declarators**.

```
package     module      knowhow     class
grammar     role        enum        subset
```

They have rather different properties, and behave in rather different ways. For example, type-checking against a `subset` type involves invoking its `where` clause.

The differences aside, they each result in some kind of **type object** that represents the type they declare.

Meta-objects

So if 6model doesn't natively know how things like inheritance and role composition work, let alone subset types, where are these things implemented?

The answer lies in **meta-objects**. Each object in existence has an associated meta-object, which describes how that object works. Many objects may have the same meta-object. In Perl 6, for example, **all objects of the same class will share a meta-object**.

What's critical to understand is that **a meta-object is just an object**. There is *nothing* magical about it. It just happens to have methods with names like `new_type`, `add_method`, `add_parent`, and so forth. As such, a meta-object is **not tied to a particular target VM**.

Representations

Meta-objects are interested in an object's type and semantics. However, they are explicitly *not* concerned with how an object is laid out in memory.

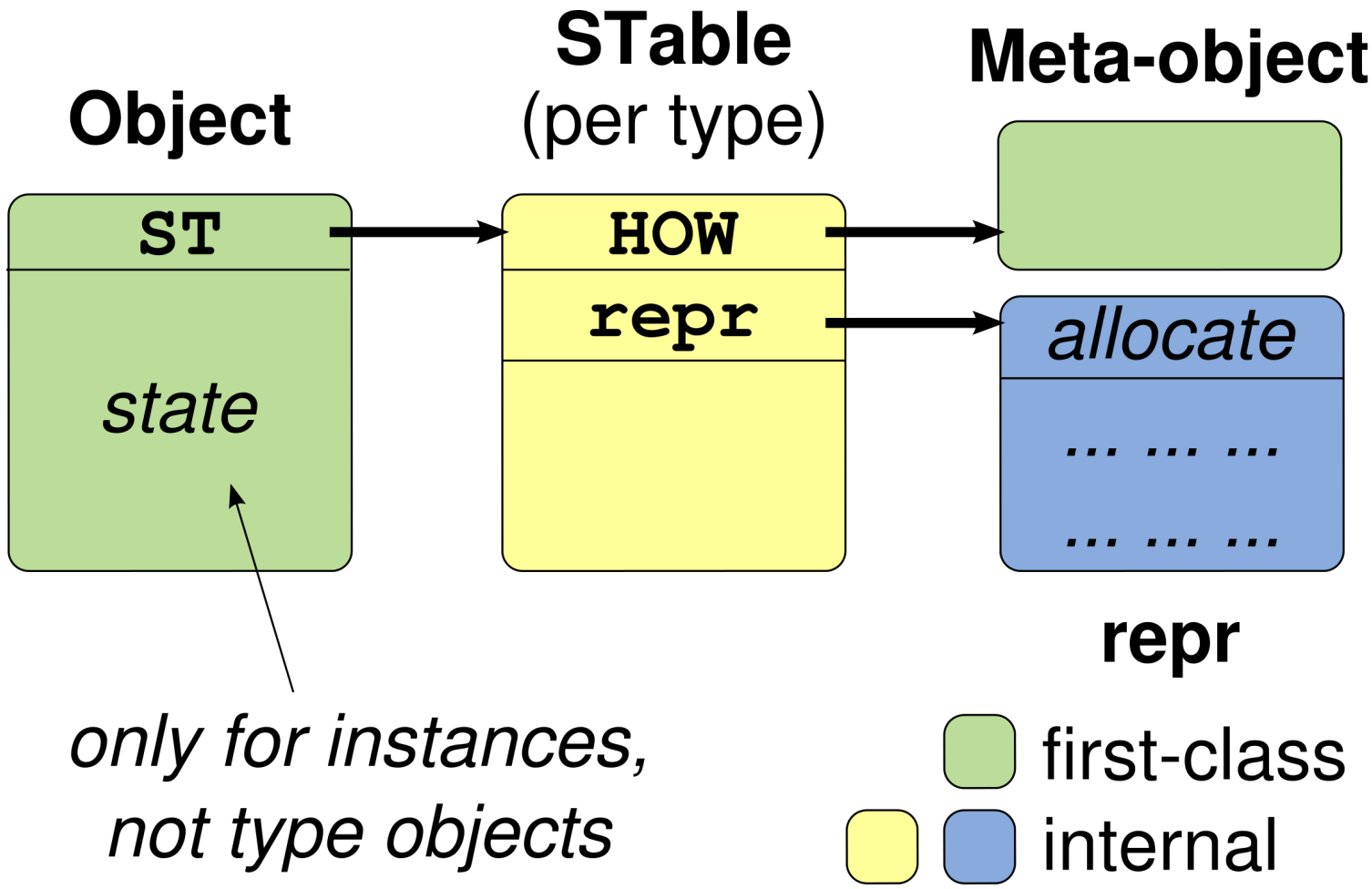
The allocation, layout and access of memory related to an object is controlled by a **representation**.

Representations are not objects. They are **low level** and **implemented in a different way per backend**. The API they provide, however, is the same.

Thus, as well as having a meta-object, each object has a representation. While meta-objects may exist per type, representations are much fewer in number.

STables combine meta-objects and REPRs

While an object has a meta-object and has a representation, there is actually a level of indirection between then: the **STable**. We'll look at these more closely later on.



Our first meta-object

Here's our very first object system. It supports types that have methods. The types will always have the P6opaque representation.

```
class SimpleHOW {
    has %!methods;

    method new_type() {
        nqp::newtype(self.new(), 'P6opaque')
    }

    method add_method($obj, $name, $code) {
        %!methods{$name} := $code;
    }

    method find_method($obj, $name) {
        %!methods{$name}
    }
}
```

Using our meta-object

First, let's create a new type and add a single method to it.

```
my $Greeter := SimpleHOW.new_type();
$Greeter.HOW.add_method($Greeter, 'greet',
    -> $self, $name { say("Hello, $name") });
```

The `$Greeter` variable now contains a type object for the new type. If we call the `greet` method on it:

```
$Greeter.greet('Katerina');
```

Then our meta-object's `find_method` method will be called with the argument `greet`, and whatever it returns will be invoked, passing `$Greeter` and the string `Katerina` as arguments.

HOW's that?

The word `HOW` is often used in connection with meta-objects. By convention, the meta-object for the keyword `class` will have a name like `ClassHOW`. It's not strictly followed, even in Rakudo and NQP. But all of the meta-objects for types do end in `HOW`.

If you take an object and use `.HOW` on it, you get the meta-object back. We can chase this up the chain as far as we wish.

```
my $mo := $Greeter.HOW;
say($mo.HOW.name($mo));              # SimpleHOW
my $momo := $mo.HOW;
say($momo.HOW.name($momo));          # NQPClassHOW
my $momomo := $momo.HOW;
say($momomo.HOW.name($momomo));      # KnowHOW
```

KnowHOW, the root of it all

If you chase the `HOW`-chain far enough back on any object, eventually you'll reach something that claims to be a `KnowHOW`. Keep going, and you just go in circles; the end of the chain is self-describing.

`KnowHOW` is the only meta-object provided by the 6model core. It supports:

- Having a name
- Having attributes
- Having methods
- In a type check, it only type checks against itself

That's it. No role composition. No inheritance.

The knowhow declarator

The `knowhow` package declarator exists in both Rakudo and NQP. You've very little reason to use it yourself. However, inside of `src/how/` in NQP, you will find a bunch of meta-objects using it:

```
NQPModuleHOW                An NQP module
NQPClassHOW                 An NQP class or grammar
NQPNativeHOW                An NQP native type (int/num/str)
NQPParametricRoleHOW        An NQP role declaration
NQPConcreteRoleHOW          An NQP role made concrete for a given class
NQPCurriedRoleHOW           An NQP role with some arguments pre-set
```

Naturally, we can't use `class` until the meta-object that implements classes is available! Thus, `knowhow` is all we have.

Giving Rubyish classes

We'll take a look at the meta-objects for NQP and Rakudo in a little bit. But first, to see something a little more manageable, let's return to the Rubyish compiler we worked on yesterday and add very basic OO support:

- Declaring a class
- Giving it methods
- Creating a class instance with a `new` statement
- Calling the methods on the class

We'll put inheritance and attributes aside for now; in fact, we'll use the NQP or Perl 6 meta-objects to study those.

Parsing a class definition

The parsing is relatively easy, however we set up a couple of extra dynamic variables related to methods. We'll see their usage next.

```
token statement:sym<class> {
    :my $*IN_CLASS := 1;
    :my @*METHODS;
    'class' \h+ <classbody>
}
rule classbody {
    :my $*CUR_BLOCK := QAST::Block.new(QAST::Stmts.new());
    <ident> \n
    <statementlist>
    'end'
}
```

Updating def for methods

Methods are declared just like functions. Inside the scope of a class, a function definition should be added to the surrounding class. The action method for `def` can be updated as follows:

```
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
    ));
    if $*IN_CLASS {
        @*METHODS.push($install);
    }
    make QAST::Op.new( :op('null') );
}
```

That is, push the `QAST::Block` onto `@*METHODS` in a class.

A simple meta-object

The meta-object we wrote before is just about good enough. Here it is with a couple of minor tweaks.

```
class RubyishClassHOW {
    has $!name;
    has %!methods;

    method new_type(:$name!) {
        nqp::newtype(self.new(:$name), 'HashAttrStore')
    }

    method add_method($obj, $name, $code) {
        %!methods{$name} := $code;
    }

    method find_method($obj, $name) {
        %!methods{$name}
    }
}
```

Building up the meta-object (1)

In Rubyish, we'll generate code that builds up the meta-object. First of all, let's take care of the `classbody` action method.

```
method classbody($/) {
    $*CUR_BLOCK.push($<statementlist>.ast);
    $*CUR_BLOCK.blocktype('immediate');
    make $*CUR_BLOCK;
}
```

Note how the `blocktype` is set to `immediate`, since we want code in the class body to run as part of the program mainline.

Building up the meta-object (2)

We mangle the name, then use `RubyishClassHOW` to create a new type object to represent it. Note that `QAST::WVal` is a way to refer to an object; we'll see much more on this later.

```
method statement:sym<class>($/) {
    my $body_block  := $<classbody>.ast;
    my $class_stmts := QAST::Stmts.new( $body_block );
    my $ins_name    := '::' ~ $<classbody><ident>;
    $class_stmts.push(QAST::Op.new(
        :op('bind'),
        QAST::Var.new( :name($ins_name), :scope('lexical'),
            :decl('var') ),
        QAST::Op.new(
            :op('callmethod'), :name('new_type'),
            QAST::WVal.new( :value(RubyishClassHOW) ),
            QAST::SVal.new( :value(~$<classbody><ident>),
                :named('name') ) )
        ));

    # <Method code comes here>
    make $class_stmts;
}
```

Building up the meta-object (3)

We also emit method calls to `add_method` to build up the method table for the class. Recall that `QAST::BVal` lets us reference a `QAST::Block` that was installed elsewhere in the tree.

```
my $class_var := QAST::Var.new( :name($ins_name), :scope('lexical') );
for @*METHODS {
    $class_stmts.push(QAST::Op.new(
        :op('callmethod'), :name('add_method'),
        QAST::Op.new( :op('how'), $class_var ),
        $class_var,
        QAST::SVal.new( :value($_.name) ),
        QAST::BVal.new( :value($_) )));
}
```

And with that, we've got classes and methods.

The new keyword

Parsing `new` is unsurprising (we skip constructor arguments):

```
token term:sym<new> {
    'new' \h+ :s <ident> '(' ')'
}
```

The actions mangle the class name to look it up, and then use the `create` NQP op to create an instance of it.

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

Method calls (1)

Last but not least, we need to parse method calls. These can be handled as a kind of postfix, with a very tight precedence. First, we add the level:

```
Rubyish::Grammar.O(':prec<y=>, :assoc<unary>', '%methodop');
```

And then the parsing, which is not too unlike how a function call was parsed.

```
token postfix:sym<.>  {
    '.' <ident> '(' :s <EXPR>* % [ ',' ] ')'
    <O('%methodop')>
}
```

Method calls (2)

The actions for a method call are relatively straightforward.

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

The key bit of "magic" that happens is that the `EXPR` action method will unshift the term the postfix was applied to, meaning it becomes the first child (and thus the invocant).

Exercise 7

In this exercise, you'll add basic support for classes and methods to PHPish. This will involve:

- Writing a basic meta-object for a class with methods
- Checking it works stand-alone
- Adding parsing for classes, methods, new statements and method calls
- Adding the relevant action methods to make things work

See the exercise sheet for more information.

STables

Each object has a meta-object and a representation. However, it does not point directly to them. Instead, each object points to an **s-table**, short for **shared table**.

STables **represent a type**, and exist per `HOW`/`REPR` combination. Here is a cut-down version of the `MVMSTable` struct from MoarVM:

```
struct MVMSTable {
    MVMREPROps *REPR;   /* The representation operation table. */
    MVMObject  *HOW;    /* The meta-object. */
    MVMObject  *WHAT;   /* The type-object. */
    MVMObject  *WHO;    /* The underlying package stash. */
    /* More... */
};
```

Representation Operations

The representation operations are broken down into:

- **Common things:** creating a new type based on the representation, composing that type (which may then compute a memory layout), allocation, cloning, changing type (used for mixins), serialization and deserialization
- **Boxing:** for types that serve as boxes of native types (int/str/num), get/set the boxed value
- **Attributes:** for types that can do storage of object attributes, get/bind attribute values as well as compute access hints
- **Positional:** for types that provide array-like storage, get and bind by index, push/pop/shift/unshift, splice, set elements
- **Associative:** for types that provide hash-like storage, get and bind by key, exists by key, delete by key

A representation can choose which of these it supports.

Common Representations

The most common representations you'll encounter while working with NQP and rakudo are:

P6opaque	Opaque attribute storage; default in Perl 6
P6int	A native integer; flattens into a P6opaque
P6num	A native float; flattens into a P6opaque
P6str	A native string reference; flattens into a P6opaque
P6bigint	Big integer; flattens into a P6opaque
VMArray	Automatically resizing array, type-parametric
VMHash	Hash table
Uninstantiable	Type object only; used for module, role, etc.

Type setup

The nqp::newtype operation is central to type creation. For example, here is the new_type method from NQPModuleHOW. It creates a new meta-object, makes a new type based upon it and the Uninstantiable representation, and gives it an empty Hash as its stash.

```
method new_type(:$name = '<anon>') {
    my $metaobj := self.new(:name($name));
    nqp::setwho(nqp::newtype($metaobj, 'Uninstantiable'), {});
}
```

nqp::newtype creates a new type object and STable. It points the type object at the STable, and the WHAT field of the STable back at the type object. It then sets the HOW field of the STable to the specified meta-object, and the REPROps to the operation table for Uninstantiable.

Type composition

Various representations need types to go through a **composition** phase. For others it is optional.

Representation composition typically happens at class composition time (which is usually done at the point of the closing } of a class declaration). It is when a meta-object has a chance to configure an underlying representation.

For example, `P6opaque` must be configured with the attributes that it should compute a layout for.

```
# <build attribute info array up into @repr_info>
my %info := nqp::hash();
%info<attribute> := @repr_info;
nqp::composetype($obj, %info)
```

`repr-compose-protocol.markdown` documents this in detail.

Method caches

If every method call really involved a call to `find_method`, method dispatch would be way too slow. Therefore, many types publish a **method cache**, which is a hash table mapping a method name to the thing to call. Here it is done by walking the method resolution order in reverse (so we get overrides correct).

```
method publish_method_cache($obj) {
    my %cache;
    my @mro_reversed := reverse(@!mro);
    for @mro_reversed {
        for $_.HOW.method_table($_) {
            %cache{nqp::iterkey_s($_)} := nqp::iterval($_);
        }
    }
    nqp::setmethcache($obj, %cache);
    nqp::setmethcacheauth($obj, 1);
}
```

Method caches hang off an `STable`.

Authoritative method caches

We can choose if the method cache is authoritative or not:

```
nqp::setmethcacheauth($obj, 0);     # Non-authoritative; default
nqp::setmethcacheauth($obj, 1);     # Authoritative
```

This really just controls what happens if the method in question is not found in the method cache. In authoritative mode, the cache is taken as having the complete set of methods. In non-authoritative mode, if the method is not found in the cache, we fall back to calling `find_method`.

It's nice to have authoritative method caches when possible, since it can give a fast answer to `nqp::can(...)`. However, any type that wants to do fallback handling cannot have this. Rakudo decides on a type-by-type basis.

Type checking

Type checks show up in many places in Perl 6:

```
if $obj ~~ SomeType { ... }          # Explicit check
my SomeType $obj = ...;              # Variable assignment
sub foo(SomeType $obj) { ... }       # Parameter binding
```

These all eventually boil down to the same operation, `nqp::istype`. However, there are many things that `SomeType` could be one of the many kinds of type:

```
class SomeType { }                   # Class type
role SomeType { }                    # Role type
subset SomeType where { ... }        # Subset type
```

Left-side-knows checks

For some kinds of type, the object being checked has the answer.
This is the case with subtyping relationships.

```
Int ~~ Mu           # Int knows it inherits from Mu
Block ~~ Callable   # Block knows it does Callable
```

These cases are handled by a `type_check` method.

```
method type_check($obj, $checkee) {
    for self.mro($obj) {
        return 1 if $_ =:= $checkee;
        if nqp::can($_.HOW, 'role_typecheck_list') {
            for $_.HOW.role_typecheck_list($_) {
                return 1 if $_ =:= $checkee;
            }
        }
    }
    return 0;
}
```

Type check caches

Once again, really iterating the MRO and the roles composed in at each level would be really slow. Therefore, left-side-knows checks are typically handled by the meta-object publishing a type-check cache.

```
method publish_type_cache($obj) {
    my @tc;
    for self.mro($obj) {
        @tc.push($_);
        if nqp::can($_.HOW, 'role_typecheck_list') {
            for $_.HOW.role_typecheck_list($_) {
                @tc.push($_);
            }
        }
    }
    nqp::settypecache($obj, @tc)
}
```

Right-side-knows checks (1)

There are other kinds of type where it's the type that we're checking against that needs to drive the checking. For example, subset types are this way:

```
subset Even of Int where * % 2 == 0;
```

We need to invoke the code associated with the `Even` subset type as part of the type check:

```
say 11 ~~ Even    # False
say 42 ~~ Even    # True
```

Right-side-knows checks (2)

These kinds of type implement an `accepts_type` method. For example, here is the one from Perl 6's `SubsetHOW`:

```
method accepts_type($obj, $checkee) {
    nqp::istype($checkee, $!refinee) &&
        nqp::istrue($!refinement.ACCEPTS($checkee))
}
```

It must also set up the appropriate type check mode for this to work:

```
nqp::settypecheckmode($type, 2)
```

Boolification

One relatively hot-path operation, it turns out, is deciding if an object will evaluate to true or false in boolean context. The `nqp::istrue` operation is used to test an object for truthiness. There's also an `nqp::isfalse`.

How an object boolifies is set through `nqp::setboolspec`, which takes a flag from the list below and an optional code object.

```
0   Call the specified code object, passing the object to test
1   Unbox as an int; non-zero is true
2   Unbox as a float; non-zero is true
3   Unbox as a string; non-empty is true
4   As above, but "0" is considered false
5   False if type object, true otherwise
6   Unbox or treat as a big integer; non-zero is true
7   For iterator objects; true if there are more items available
8   For VMArray/VMHash based objects; true if elems is non-zero
```

Invocation

There is also an invocation specification mechanism, which indicates what happens if an object is invoked (called).

In Rakudo, and often in NQP too, we have code objects. These in turn hold a VM level code object. When we invoke a code object, the invocation needs to be forwarded to the contained code object.

Here's an example from NQP's setting:

```
my knowhow NQPRoutine {
    has $!do;
    ...
}
nqp::setinvokespec(NQPRoutine, NQPRoutine, '$!do', nqp::null);
```

In Rakudo, see `Perl6::Metamodel::InvocationProtocol`.

NQP's meta-objects

NQP's meta-objects are all implemented using the `knowhow` meta-object. They also cannot assume the presence of the NQP setting, meaning you'll find some slightly odd code in there.

- The NQP iterator types for hashes that enable `.key` and `.value` methods are not yet set up, so this code uses `nqp::iterkey_s` and `nqp::iterval`.
- There is no `NQPMu` default for scalars to take yet, so an empty scalar will be null; `nqp::isnull` is therefore used for often.

Thankfully, your chances of needing to work on this code are fairly low. It's also relatively compact; `NQPClassHOW`, the most complex meta-object, is only around 800 lines of largely straightforward code.

Rakudo's meta-objects: overview

The story is much different in Rakudo. Rakudo's meta-objects are implemented in terms of NQP's classes and roles. This means that inheritance and role composition are available.

Therefore, while **Rakudo's meta-objects must handle much more** due to the richness of the Perl 6 object system, they are **very neatly factored**.

There is **a meta-object per declarator** (so `class` maps to `ClassHOW`), and a few extra bits for roles (which are rather complex to implement due to their type parametricity).

However, much functionality is **factored out into roles**, which are re-used amongst the different meta-objects.

Example: ClassHOW

Here are the roles that are done by `Perl6::Metamodel::ClassHOW`:

```
Naming                      Documenting
Versioning                  Stashing
AttributeContainer          MethodContainer
PrivateMethodContainer      MultiMethodContainer
RoleContainer               MultipleInheritance
DefaultParent               C3MRO
MROBasedMethodDispatch      MROBasedTypeChecking
Trusting                    BUILDPLAN
Mixins                      ArrayType
BoolificationProtocol       REPRComposeProtocol
InvocationProtocol
```

Amongst the names, you'll recognize many Perl 6 features, as well as some of the 6model concepts we've covered in this section.

Example: EnumHOW

If we look at `Perl6::Metamodel::EnumHOW`, we'll see that it re-uses a number of these roles:

```
Naming                      Stashing
AttributeContainer          MethodContainer
MultiMethodContainer        RoleContainer
MROBasedMethodDispatch      MROBasedTypeChecking
BUILDPLAN                   BoolificationProtocol
REPRComposeProtocol         InvocationProtocol
```

In fact, it has just one extra role that it composes:

```
BaseType
```

The roles aside, `ClassHOW` is 250 lines of code, and `EnumHOW` about 150. Thus, most interesting stuff lives in the roles.

Example: Naming

Some of the roles are extremely simple. For example, all of the meta-objects compose the `Naming` role, which simply provides two methods and a `$!name` attribute:

```
role Perl6::Metamodel::Naming {
    has $!name;
    method set_name($obj, $name) {
        $!name := $name
    }
    method name($obj) {
        $!name
    }
}
```

The role with most code is `C3MRO`, which computes the C3 method resolution order. It's still only 150 lines of code, though. Takeaway: things are divided into quite manageable pieces.

Example: GrammarHOW

This is the simplest meta-object:

```
class Perl6::Metamodel::GrammarHOW
    is Perl6::Metamodel::ClassHOW
    does Perl6::Metamodel::DefaultParent
{
}
```

Essentially, a `grammar` does everything that a `class` does, but composes the `DefaultParent` role so as to enable grammars to be configured with a different default parent in `BOOTSTRAP`:

```
Perl6::Metamodel::ClassHOW.set_default_parent_type(Any);
Perl6::Metamodel::GrammarHOW.set_default_parent_type(Grammar);
```

Container handling

So far, we've seen that a type can be given a boolification spec and an invocation spec. There is one more of these: **container spec**. This is used in implementing the `Scalar` container type in Perl 6.

Several operations relate to this:

```
setcontspec          Configure a type as a scalar container type
iscont               Check if an object is a scalar container
decont               Get the value inside the container
assign               Assign a value into the container
assignunchecked      Assign, assuming no type-check needed
```

For example, Rakudo's `BOOTSTRAP` does:

```
nqp::setcontspec(Scalar, 'rakudo_scalar', nqp::null());
```

Auto-decontainerization

One may wonder why `nqp::decont` doesn't need to show up absolutely everywhere in Perl 6. The answer is that a range of `nqp::ops` will automatically do a `nqp::decont` operation for you.

One commonly encountered exception is that **attribute access doesn't decontainerize**. This means `nqp::getattr` and friends may need an explicit `nqp::decont` on their first argument.

```
nqp::getattr(nqp::decont(@list.Parcel), Parcel, '$!storage')
```

However, since `self` is defined to always be decontainerized anyway, this is not normally a problem.

Exercise 8

As time allows, extend the PHPish object system to have:

- A method cache (you may like to time if it makes a difference)
- Single inheritance of classes (which will need updates to your method cache code)
- Interfaces (these will need a different meta-object, and you will need to add a compose-time to the class, to check all named methods in the interface are provided)

As usual, the exercise sheet has more hints.

Bounded Serialization and Module Loading

Let's save the World!

A problem

When we built object support into Rubyish, we did so by emitting code to make calls on the meta-objects. Doing this clearly has downsides for startup time. In Perl 6, however, there are much more serious challenges to this approach. Consider the following example:

```
class ABoringExample {
    method yawn() { say "This is at compile time!"; }
}
BEGIN { ABoringExample.yawn }
```

A `BEGIN` block runs while we are compiling. Therefore, the type object and meta-object for `ABoringExample` needs to be available at the point we run the `BEGIN` block.

Also, this must work for user-defined meta-objects.

This problem is everywhere

A subroutine declaration produces a `Sub` object, which in turn refers to a `Signature` object which in turn has `Parameter` objects inside of it.

All of these need constructing at compile time. Not only since we could call the sub, but also because traits may need to mix into it:

```
role StoredProcWrapper { has $.sp_name }
multi trait_mod:<is>(Routine:D $r, :sp_wrapper($sp_name)!) {
    $r does StoredProcName($sp_name)
}
# ...
sub LoadStuffAsObjects($id) is sp_wrapper('LoadStuff') {
    call_sp($id).map({ Stuff.new(|%($_)) })
}
```

Compile-time vs. runtime

The problem, in general, is that we need to be able to build up objects and meta-objects at compile time, then refer to them at runtime. Moreover, this is a very common case, so we need to do so efficiently.

That in itself wouldn't be too bad. However, module pre-compilation makes this a good bit trickier: **the objects created at compile time may need to cross a process boundary**, being saved to disk, then loaded at some future point.

This is where serialization contexts, bounded serialization and `Worlds` come in to play.

The World

One concept our small `Rubyish` language lacked, but that both NQP and Rakudo have, is a `World` class. While the `Actions` class is focused on QAST trees, and thus the runtime semantics of a program, a `World` class is focused on **managing declarations and meta-objects** during the compile.

A world always has a unique handle per compilation unit. This may be based on the original source text, such as in Rakudo.

```
my $file := nqp::getlexdyn('$?FILES');
my $source_id := nqp::sha1(
    nqp::defined(%*COMPILING<%?OPTIONS><outer_ctx>)
        ?? self.target() ~ $sc_id++                  # REPL/eval case
        !! self.target());                           # Common case
my $*W := Perl6::World.new(:handle($source_id), :description($file));
```

Serialization contexts

The key data structure at the heart of compile-time/runtime object exchange is a **serialization context**. Really, a serialization context is just three arrays, one each for:

- **Objects:** any 6model object can appear in this list, though it only makes sense to put those that are sensible to serialize in there
- **Code objects:** VM-level code objects that objects in the serialization context may refer to (or refer to through indirectly, due to a closure cloning)
- **STables:** the existence of this array is an implementation detail, and its contents is never directly manipulated outside of VM-specific code, so you can forget about it

There is one `World` per compilation unit, and a `World` in turn holds a serialization context. In fact, the `handle` given to `World.new(...)` is actually used for the SC.

Placing objects in a serialization context

Both `NQP::World` and `Perl6::World` inherit from `HLL::World`. It includes a method named `add_object`, which adds an object into the serialization context for the current compilation unit. Here is how it is used in `NQP::World`, for example:

```
method pkg_create_mo($how, :$name, :$repr) {
    my %args;
    if nqp::defined($name) { %args<name> := $name; }
    if nqp::defined($repr) { %args<repr> := $repr; }
    my $type_obj := $how.new_type(|%args);
    self.add_object($type_obj);
    return $type_obj;
}
```

Referencing objects in a serialization context

Any object that is in a serialization context - either the one currently being compiled or from one in another module or setting - can be referenced using the `QAST::WVal` node type.

For example, here is a utility method from `Perl6::World`:

```
method add_constant_folded_result($r) {
    self.add_object($r);
    QAST::WVal.new( :value($r) )
}
```

The `W` in `QAST::WVal` means "World", which should make a little more sense now than it did when we encountered it previously. :-)

Serialization

The compiler toolchain knows if the eventual target is to run code in-process or generate bytecode to write to disk.

In the first case, it's easy: we just make sure it is possible to see the serialization context from the running code, and compile a `QAST::WVal` to index into it.

The second case requires serializing all the objects in the serialization context, and in turn serializing the objects that they point to, traversing the object graph as needed.

They are dumped to a binary serialization format, documented in the NQP repository.

What's "bounded" about it

Consider pre-compiling the following module:

```
class Cache is Hash {
    has &!computer;
    submethod BUILD(:&!computer!) { }
    method at_key($key) is rw {
        callsame() //= &!computer($key)
    }
}
```

Here, `Hash` comes from Perl 6's `CORE.setting`. Clearly, we will encounter this type in the `@!parents` of the meta-object for `Cache`. However, we do not want to re-serialize the `Hash` type!

When an object is already owned by another SC, we just write a reference to it. **Ownership is the boundary of a compilation unit's serialization.**

Deserialization and fixups

The opposite of serialization is deserialization. This involes taking the binary blob representing objects and STables and recreating the objects from it.

In doing this, all references to object from other serialization contexts must be resolved. This means that they must have been loaded first. This implies that a module's dependencies must be loaded before it can be deserialized.

For this reason, `HLL::World` has an `add_load_dependency_task`, for adding code (specified as QAST) to execute before deserialization takes place.

There is also an `add_fixup_task`, which enables registration of code to run after deserialization has taken place.

Another tricky problem

One tricky issue is what happens if you try to pre-compile a module containing the following:

```
# Ooh! Let's pretend we're Ruby!
augment class Int {
    method times(&block) {
        for ^self { block($_) }
    }
}
```

The `Int` meta-object and STable are serialized in `CORE.setting`. But here, another module is modifying the meta-object, and the updated method cache is hung off the `STable`, meaning it too has changed.

So what do we do?

Repossession

When an object that belongs to a serialization context, we're at compile time, and the serialization context it belongs to is not one we're curerntly in the process of compiling, a write barrier is triggered.

This switches the ownership of the object to the serialization context of the compilation unit we're currently compiling. It also records that this happened.

At serialization, the updated version of the object is serialized.

At deserialization, the object to update is located and then overwritten with the new version of it.

Repossession conflicts

This leaves just one more issue: what happens if you load two pre-compiled modules that both want to augment the same class?

Once, "latest won". Thankfully, today this is detected as a repossession conflict, the resulting exception indicating two modules were loaded that may not be used together.

This should have been the end of the story. But it's not. It turns out that `Stash` objects started to conflict in interesting ways, when modules used nested packages. Therefore, there is now a conflict resolution mechanism that looks at the objects in conflict and tries to merge them. For `Stash`, that is easy enough.

SC write barrier control

Most of the `nqp::ops` related to serialization contexts are rarely seen, hidden away in `HLL::World`. However, two of them escape into regular code:

- `nqp::scwbdisable` disables the repossession detection write barrier, meaning that any changes done to an owned object will not cause it to be re-serialized. This is often done by meta-objects that want to keep caches.
- `nqp::scwbenable` re-enables repossession detection.

Note that this isn't a binary flag, but rather a counter that is incremented by the first op and decremented by the second. Repossession detection happens only when the counter is at zero.

Accidental Repossession

It's important to keep repossession in mind when working on Rakudo and NQP, as it can sometimes kick in when you might not have expected it.

For example, in Rakudo's CORE.setting, you'll find a `BEGIN` block that looks like this:

```
BEGIN {
    my Mu $methodcall     := nqp::hash('prec', 'y=');
    ...
    trait_mod:<is>(&postfix:<i>, :prec($methodcall));
    ...
}
```

If this were done in the setting mainline, it would cause a change to the `postfix:<i>` serialized in the CORE setting, which could as a result cause a repossession of this by whatever compilation unit triggers setting loading.

QAST::CompUnit, revisited

The various pieces assembled by the World are passed down to the backend using `QAST::CompUnit`.

```
my $compunit := QAST::CompUnit.new(
    :hll('perl6'),
    :sc($*W.sc()),
    :code_ref_blocks($*W.code_ref_blocks()),
    :compilation_mode($*W.is_precompilation_mode()),
    :pre_deserialize($*W.load_dependency_tasks()),
    :post_deserialize($*W.fixup_tasks()),
    :repo_conflict_resolver(QAST::Op.new(
        :op('callmethod'), :name('resolve_repossession_conflicts'),
        QAST::Op.new(
            :op('getcurhllsym'),
            QAST::SVal.new( :value('ModuleLoader') )
        )
    )),
    ...);
```

How module loading works (1)

When a `use` statement is encountered in Perl 6 code:

```
use Term::ANSIColor;
```

The module name is parsed, any adverbs extracted (such as `:from`) and then control is passed on to the `load_module` method in `Perl6::World`:

```
my $lnd    := $*W.dissect_longname($longname);
my $name   := $lnd.name;
my %cp     := $lnd.colonpairs_hash('use');
my $module := $*W.load_module($/, $name, %cp, $*GLOBALish);
```

How module loading works (2)

This `load_module` method first delegates to `Perl6::ModuleLoader` to load the module right away (required as it will probably introduce types or do other changes that we need to continue parsing). Once the module is loaded, it also registers a load dependency task to make sure the module is loaded if we are in a pre-compiled situation before deserialization takes place.

```
method load_module($/, $module_name, %opts, $cur_GLOBALish) {
    my $line   := HLL::Compiler.lineof($/.orig, $/.from, :cache(1));
    my $module := Perl6::ModuleLoader.load_module($module_name, %opts,
        $cur_GLOBALish, :$line);

    if self.is_precompilation_mode() {
        self.add_load_dependency_task(:deserialize_past(...));
    }

    return $module;
}
```

How module loading works (3)

Inside `Perl6::ModuleLoader`, some work is done to locate where the module is on disk. If it exists in a pre-compiled form, the `nqp::loadbytecode` op is used to load it. Otherwise, the source is slurped from disk and compiled.

Loading a pre-compiled module automatically triggers its deserialization.

A couple of odd lines that are executed on both code paths deserve some explanation, however:

```
my $*CTXSAVE := self;
my $*MAIN_CTX;
nqp::loadbytecode(%chosen<load>);
%modules_loaded{%chosen<key>} := $module_ctx := $*MAIN_CTX;
```

How module loading works (4)

When the mainline of the module is run, its lexical scope is captured by some code equivalent to:

```
if $*CTXSAVE && nqp::can($*CTXSAVE, 'ctxsave') {
    $*CTXSAVE.ctxsave();
}
```

The ModuleLoader has such a method:

```
method ctxsave() {
    $*MAIN_CTX := nqp::ctxcaller(nqp::ctx());
    $*CTXSAVE := 0;
}
```

This is how the `UNIT` (outer lexical scope) of a module being loaded is obtained. This is in turn used to locate `EXPORT`.

How module loading works (5)

Finally, `ModuleLoader` triggers global merging. This involves taking the symbols the module wishes to contribute to `GLOBAL` and incorporating them into the current view of `GLOBAL`.

If this sounds strange, note that Perl 6 has separate compilation, meaning all modules start out with a completely clean and empty view of `GLOBAL`. These views are reconciled (and conflicts whined about) as modules are loaded.

Finally, the `UNIT` lexpad is returned.

```
my $UNIT := nqp::ctxlexpad($module_ctx);
if +@GLOBALish {
    unless nqp::isnull($UNIT<GLOBALish>) {
        merge_globals(@GLOBALish[0], $UNIT<GLOBALish>);
    }
}
return $UNIT;
```

How module loading works (6)

What we have seen so far is what a `need` would do. A `use` then goes on to import. This is not implemented in the module loader, but rather lives in the `import` method in `Perl6::World`.

It does the following things:

- Locates the symbols that need to be imported
- If there are multiple dispatch candidates exported and there also exist some in the target scope, merges the candidate lists
- For other symbols, installs them directly into the target scope, complaining if there is a conflict
- If any operators are imported, makes sure the current language is augmented so as to be able to parse them

The regex and grammar engine

Inside how Perl 6 is parsed

The pieces involved

Regex and grammar handling involves a number of components:

- The **Perl 6 Regex grammar/actions**, from `src/QRegex/P6Regex`, which parse the Perl 6 regex syntax and produce a QAST tree from it. These are not used directly by NQP and Rakudo, but instead subclassed (so, for example, nested code blocks will be parsed in the correct main language)
- The `QAST::Regex` QAST node, which represents the whole range of regex constructs we can compile
- `Cursor` **objects**, which keep state as we parse
- `Match` **objects**, which represent the result of a parse
- **NFA construction and evaluation**, used for Longest Token Matching

The QAST::Regex node

This node covers all of the regex constructs. It has an `rxtype` property that is used to indicate the kind of regex operation to perform.

It can be placed at any point in a QAST tree, though typically expects to find itself inside of a `QAST::Block`. Furthermore, it expects the lexical `$` to have been declared.

With a few exceptions, once you reach a `QAST::Regex` node, the QAST compiler will expect to find only other `QAST::Regex` nodes beneath it. There is an explicit `qastnode` rxtype for escaping back to the rest of QAST.

We'll now study the rxtypes available.

literal

The `literal` rxtype indicates a literal string that should be matched in a regex. The string to match is passed as a child to the node.

```
QAST::Regex.new( :rxtype<literal>, 'meerkat' )
```

It has one subtype, `ignorecase`, which makes matching of the literal be case insensitive.

```
QAST::Regex.new( :rxtype<literal>, :subtype<ignorecase>, 'meerkat' )
```

concat

The concat subtype is used to match a sequence of QAST::Regex nodes one after the other. It expects these nodes as its children.

This will do the same as the previous slide, though will be a little less efficient:

```
QAST::Regex.new(
    :rxtype<concat>,
    QAST::Regex.new( :rxtype<literal>, 'meer' ),
    QAST::Regex.new( :rxtype<literal>, 'kat' )
)
```

scan and pass

Regexes tend to start with a `scan` node and end with a `pass` node.

- `scan` will generate code to work through the string, trying to match the pattern at each offset, until either a match is successful or it runs out of string to try. This is what makes `'slaughter' ~~ /laughter/` match, even though `laughter` is not at the start of the string. Note it will only do this if the match is not anchored (which it will be if called by another rule).
- `pass` will generate a call to `!cursor_pass` on the current `Cursor` object, indicating that the regex has matched. For named regexes, tokens and rules, this node conveys the name of the action method to invoke also.

A simple example

If we give NQP the following regex:

```
/meerkat/
```

And use `--target=ast`, the resulting `QAST::Regex` nodes contain all of the things we have covered so far:

```
- QAST::Regex(:rxtype(concat))
  - QAST::Regex(:rxtype(scan))
  - QAST::Regex(:rxtype(concat)) meerkat
    - QAST::Regex(:rxtype(literal)) meerkat
      - meerkat
  - QAST::Regex(:rxtype(pass))
```

cclass

Used for the various common built-in character classes, typically expressed through backslash sequences. For example, \d and \W respectively become:

```
QAST::Regex.new( :rxtype<cclass>, :name<d> )
QAST::Regex.new( :rxtype<cclass>, :name<w>, :negate(1) )
```

The available values for name are as follows:

```
Code    Meaning
.       Any character (really, any)
d       Any numeric character (Unicode aware)
s       Any whitespace character (Unicode aware)
w       Any word character or the underscore (Unicode aware)
n       A literal \n, a \r\n sequence, or a Unicode LINE_SEPARATOR
```

enumcharlist

Used for user-defiend character classes. Requires that the current character class be any of those specified in the child string.

For example, \v (which matches any vertical whitespace character) compiles into:

```
QAST::Regex.new(
    :rxtype<enumcharlist>,
    "\x[0a,0b,0c,0d,85,2028,2029]"
)
```

enumcharlist and user defined character classes

The `enumcharlist` node is also used in things like:

```
/<[A..Z]>/
```

Which, as `--target=ast` shows, becomes:

```
- QAST::Regex(:rxtype(concat))
  - QAST::Regex(:rxtype(scan))
  - QAST::Regex(:rxtype(concat)) <[A..Z]>
    - QAST::Regex(:rxtype(enumcharlist)) [A..Z]
      - ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - QAST::Regex(:rxtype(pass))
```

anchor

Used for various zero-width assertions. For example, ^ (start of string) compiles into:

```
QAST::Regex.new( :rxtype<anchor>, :subtype<bos> )
```

The available subtypes are:

```
bos     Beginning of string (^)
eos     End of string ($)
bol     Beginning of line (^^)
eol     End of line ($$)
lwb     Left word boundary (<<)
rwb     Right word boundary (>>)
fail    Always fails
pass    Always passes
```

quant

Used for quantifiers. The `min` and `max` properties are used to indicate how many types the child node may match. A `max` of `-1` means "unlimited". Thus, the regex `\d+` compiles into:

```
QAST::Regex.new(
    :rxtype<quant>, :min(1), :max(-1),
    QAST::Regex.new( :rxtype<concat>, :name<d> )
)
```

The `backtrack` property can also be set to one of:

```
g       Greedy matching (\d+:, the default)
f       Frugal (minimal) matching (\d+?)
r       Ratchet (non-backtracking) matching (\d+:)
```

altseq

Tries to match its children in order, until it finds one that matches. This provides || semantics in Perl 6, which are the same as | semantics in Perl 5. Thus:

```
the || them
```

Compiles into:

```
QAST::Regex.new(
    :rxtype<altseq>,
    QAST::Regex.new( :rxtype<literal>, 'the' ),
    QAST::Regex.new( :rxtype<literal>, 'them' )
)
```

There is also `conjseq` for Perl 6's `&&`.

alt

Support Perl 6 LTM-based alternation. The regex:

```
the | them
```

Compiles into:

```
QAST::Regex.new(
    :rxtype<alt>,
    QAST::Regex.new( :rxtype<literal>, 'the' ),
    QAST::Regex.new( :rxtype<literal>, 'them' )
)
```

This will always match `them` if it can, because it goes for the branch with the longest declarative prefix first.

subrule (1)

Used to call another rule, optionally capturing. For example:

```
<ident>
```

Will compile into:

```
QAST::Regex.new(
    :rxtype<subrule>, :subtype<capture>, :name<ident>,
    QAST::Node.new( QAST::SVal.new( :value('ident') ) )
)
```

The `name` property is the name to capture as, while the `QAST::SVal` node is taken as the name of the method to call. Extra children may be given to the `QAST::Node`, which will be taken as arguments for the call.

subrule (2)

There are a few other things worth noting about subrule. First, it need not capture. For example:

```
<.ws>
```

Will compile into:

```
QAST::Regex.new(
    :rxtype<subrule>, :subtype<method>,
    QAST::Node.new( QAST::SVal.new( :value('ws') ) )
)
```

subrule (3)

The `subrule` rxtype is also capable of handling zero-width assertions. For example:

```
<?alpha>
```

Will compile into:

```
QAST::Regex.new(
    :rxtype<subrule>, :subtype<zerowidth>,
    QAST::Node.new( QAST::SVal.new( :value('ws') ) )
)
```

subrule (4)

Finally, there are two other properties that apply to `subrule`:

- `backtrack` being set to `r` will prevent the subrule call being backtracked into. This is set in `token` and `rule`, and avoids keeping a lot of state around.
- `negate` can also be set on this node. It is probably most useful in combination with the `zerowidth` subtype, since that is how ‘ is compiled.

Last but not least, `subrule` is also used for positional captures. Instead of specifying a method to call, the contents of the capture is compiled inside a nested `QAST::Block` and that is called. This is to make sure positional matches get their own `Match` object.

subcapture

This is used for implementing named captures that are not subrules. That is:

```
$<num>=[\d+]
```

Will compile into:

```
QAST::Regex.new(
    :rx<subcapture>, :name<num>,
    QAST::Regex.new(
        :rxtype<quant>, :min(1), :max(-1),
        QAST::Regex.new(
            :rxtype<cclass>, :name<d>
        )
    )
)
```

Cursor

A `Cursor` is an object that **holds the current state of a match**. `Cursors` are created at the point of entry to a `token`/`rule`/`regex`, and either pass or fail. From that point on, a `Cursor` is immutable.

The state inside a `Cursor` includes:

- The target string
- The position we're matching from in the current rule (-1 indicates scan)
- The current position reached by the match
- A stack of backtrack marks (more later)
- A stack of captured cursors (more later)
- Potentially, a cached `Match` object produced from the `Cursor`
- For a passed `Cursor` that we may backtrack into later, the code object to invoke to restart matching

NQPCursorRole

Both NQP and Rakudo have their own cursor objects, named `NQPCursor` and `Cursor` respectively. However, they both compose `NQPCursorRole`, which provides most of their methods.

The methods can be categorized as follows:

- Common introspection methods: `orig`, `target`, `from` and `pos`
- Built-in rules: `before`, `after`, `ws`, `ww`, `wb`, `ident`, `alpha`, `alnum`, `upper`, `lower`, `digit`, `xdigit`, `space`, `blank`, `cntrl`, `punct`
- Infrastructure methods: all have a name starting with a `!` and are called mostly by code generated from compiling `QAST::Regex` nodes or as part of implementing the built-in rules

It starts with !cursor_init

Parsing a grammar or matching a string against a regex always starts with a call to `!cursor_init`, which creates a `Cursor` and initializes it with the target string, setting up options (such as whether to scan or not).

For example, here is how `NQPCursor`'s `parse` method is implemented:

```
method parse($target, :$rule = 'TOP', :$actions, *%options) {
    my $*ACTIONS := $actions;
    my $cur := self.'!cursor_init'($target, |%options);
    nqp::isinvokable($rule) ??
        $rule($cur).MATCH() !!
        nqp::findmethod($cur, $rule)($cur).MATCH()
}
```

Inside a rule (1)

The first thing that happens on entry to a `token`, `rule` or `regex` is the creation of a new `Cursor` to track its work. This is done by calling the `!cursor_start_all` method, which returns an array of state, including:

- The newly created `Cursor`
- The target string
- The position to start matching from (-1 indicates scan)
- The current `Cursor` type (generic `$?CLASS`)
- The backtracking mark stack
- A restart flag: 1 if it is a restart, 0 otherwise

Aside: this exact factoring will likely change in the future, for performance reasons.

Inside a rule (2)

The `Cursor` returned by `!cursor_start_all` may have various methods call on it as a match proceeds:

- `!cursor_start_subcapture` to produce a `Cursor` that will represent a sub-capture
- `!cursor_capture` pushes a `Cursor` onto the capture stack (either one returned by calling a subrule or one created for a subcapture)
- `!cursor_pos` updates the match position in the `Cursor` (it's only synchronized when needed)
- `!cursor_pass` if the match is successful; the position reached must be passed, and if it is a named regex then the name can be passed; this also triggers a call to an action method
- `!cursor_fail` if the match fails

Inside a rule (3)

Once a `token`, `rule` or `regex` has finished matching, either passing or failing, it should return the `Cursor` that it worked against.

In fact, this is the protocol: anything that is called as a `subrule` should return a `Cursor` to its caller. Failing to do so will cause an error.

At the point a `Cursor` is failed, any backtracking and capture state will be discarded. If it passes, but can not be backtracked in to, then backtracking state can be thrown away too.

The cstack and capturing

The `cstack` (either **Capture stack** or **Cursor stack**) is where `Cursor` objects that correspond to captures (positional or named) are stored. It may also be used to store non-captured `Cursors` for subrules we could backtrack in to.

In something like:

```
token xblock {
    <EXPR> <.ws> <pblock>
}
```

The `cstack` will end up with two `Cursors` on it by the end of the match: one returned by the call to `EXPR` and another returned by the call to `pblock`.

The bstack and backtracking

The `bstack` is a stack of integers. Each "mark" actually consists of four integers (so it only makes sense to talk about groups of 4 entries, not the individual integers):

- The location in the regex to jump back to (typically interpreted by a jump table); if 0, then the backtracker should just go on looking at the next entry
- The position in the string to go back to
- Optionally, a repetition count (used by quantifiers)
- The height of the `cstack` at the point the mark was made. This is used to throw away any captures that we backtrack over.

Match object production

The `MATCH` method on a `Cursor` or `NQPCursor` takes the `Cursor` and makes a `Match` or `NQPMatch` object. These are the things our action methods were passed as their `$/` argument.

They are produced by looking at the `cstack`, observing the names of each of the entries, and building up an array of positional captures and a hash of named captures. Positional captures just have an integer name.

Any capture qauntified with `*`, `+` or `**` will produce an array of captured results.

Most of this work is factored out by `CAPHASH` from `NQPCursorRole`.

Longest Token Matching

All of this leaves one important regex related topic: Longest Token Matching. We've already seen it in action, but now we'll take a few moments to consider how it works.

Every regex or branch alternation has a (possibly zero-length) declarative prefix. It covers the region from the start of a regex up to a construct that is deemed imperative (such as a code block, positive lookahead, etc.)

```
token even { \d+ { +$/ % 2 == 0 } }
             DDD IIIIIIIIIIIIII
```

The declarative prefix always forms a **regular language**, and as a result can be translated into a **finite automata**.

NFA fragments

Once an individual `token`, `rule` or `regex` has been compiled to QAST, the QAST tree is passed to `QRegex::NFA`.

This explores the QAST, identifies the declarative prefix, and builds an NFA (Non-deterministic Finite Automata) out of it.

If the QAST tree contains any alternations, then each branch of these also has an NFA build and stored.

At this point, the NFAs are not ready to evaluate. Whenever there is a subrule call, they simply name the call. In that sense, they are **generic with regard to the grammar as a whole**, and may need to be made concrete many times (due to grammar inheritance).

Protoregex and alternation NFAs

A protoregex decides which candidate to call by building an NFA representing the alternation of all the candidate NFAs.

This protoregex NFA is always specific to a particular type of grammar. As a part of producing it, any subrule calls have their NFA substituted in for the call.

Alternations go through a similar process, except this time the NFA is built up out of the NFAs of the branches.

The result of either of these is an NFA that can be executed against a target string.

NFA execution

There are two `nqp::ops` that relate to executing NFAs:

- `nfarunproto` evaluates the NFA from a given offset in the target string. It returns an array indicating the order in which the candidates should be tried, excluding any that could never possibly match.
- `nfarunalt` evaluates the NFA from a given offset in the target string. It then pushes marks for all the branches that could possibly match onto the `bstack` in reverse order, so the best possible candidate is at the top. The regex engine then just immediately "backtracks" to start trying the possible candidates.

These two really are just thin wrappers around the same underlying NFA evaluator.

Exercise 9

In this exercise, you'll explore some of the regex engine implementation. Of note, you'll encounter (time-allowing):

- The Perl 6 regex grammar and actions
- How embedded code blocks are implemented in NQP and Rakudo
- Where NFAs are stored and how they look

See the exercise sheet for guidance.

The JVM backend

Bringing Perl 6 to the land of Java

The JVM

Virtual machine originally built to execute the Java language, and now host to a large number of languages spanning many paradigms, static, dynamic, etc.

Instruction set of **around 200 instructions**, but many class library methods are provided natively by the VM also

Instruction set and execution model are **stack based**; values are loaded on to the stack to be operated on, passed as method arguments, etc.

The **bytecode** lives in a **class file**, which represents a single class with fields and methods.

JVM instruction set: constants

Various instructions load constants on to the stack:

- `aconst_null` loads a null reference
- `iconst_m1`, `iconst_0`, `iconst_1`, ... `iconst_5` load 32-bit integer -1, 0, 1, ... 5 onto the stack
- `lconst_0`, `lconst_1` load 64-bit integer 0 and 1 onto the stack
- `fconst_0`, `fconst_1`, `fconst_2` load 32-bit floating point 0.0, 1.0, 2.0 onto the stack
- `dconst_0`, `dconst_1` load 64-bit flaoting point 0.0, 1.0 onto the stack
- `bipush` takes a 1-byte argument and loads it as a 32-bit integer
- `sipush` takes a 2-byte argument and loads it as a 16-bit integer
- `ldc`, `ldc_w` **and** `ldc2_w` load constants from the constant pool (int, float or String)

JVM instruction set: locals

Local variables are either integers (32-bit), longs (64-bit), floats (32-bit), doubles (64-bit) or object reference. There are instructions to load and store them.

- `iload`, `lload`, `fload`, `dload` **and** `aload` take an index and load that local variable onto the stack
- `istore`, `lstore`, `fstore`, `dstore` **and** `astore` take an index and store what is currently on the stack top to that local variable
- The first four local variables (indexes 0 through 3) can be accessed using special instructions of the form `<prefix>[load|store]_[0..3]`, for example `iload_0`, `lload_3`, `astore_2`, `dstore_0`

Both longs and doubles count as two slots, so two adjacent longs might be in index 4 and 6; trying to access something at 5 will complain about splitting a value!

JVM instruction set: arrays

Arrays can be created with types `int`, `long`, `float`, `double` `byte`, `char`, `short` or any reference type. They are not resizable.

- `newarray` creates an array of any of the native types, with a byte to indicate type and taking the length to allocate from the stack top
- `anewarray` is for creating arrays of a reference type; the type is specified as a constant pool entry
- `arraylength` gets the length of an array
- Loading an element from an array involves putting the array on the stack, the index on the stack, and then using one of `iaload`, `laload`, `faload`, `daload`, `aaload`, `baload`, `caload` or `saload`
- Storing an element to an array involves putting the array on the stack, the index on the stack, the value on the stack, and then using one of `iastore`, `lastore`, `fastore`, `dastore`, `aastore`, `bastore`, `castore`, or `sastore`

JVM instruction set: arithmetic

The usual set of arithmetic and bitwise operations are available

- **Addition:** `iadd`, `ladd`, `fadd`, `dadd`
- **Subtraction:** `isub`, `lsub`, `fsub`, `dsub`
- **Multiplication:** `imul`, `lmul`, `fmul`, `dmul`
- **Division:** `idiv`, `ldiv`, `fdiv`, `ddiv`
- **Modulo:** `irem`, `lrem`, `frem`, `drem`
- **Negation:** `ineg`, `lneg`, `fneg`, `dneg`
- **Bit shifts**: `ishl`, `lshl`, `ishr`, `lshr`, `iushr`, `lushr`
- **Bitwise:** `iand`, `land`, `ior`, `lor`, `ixor`, `lxor`

JVM instruction set: compare/branch

Bloody irregular!

For longs, floats and doubles, you use one of `lcmp`, `fcmpl`, `fcmpg`, `dcmpl`, or `dcmpg`, which give -1, 0 or 1 (like a `cmp` in various languages). You then branch with one of `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, or `ifle`.

32-bit integer comparisons are special enough to get their own instructions that compare and branch all in one: `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpge`, `if_icmpgt` and `if_icmple`.

References can be compared for equality or inequality and branched on with `if_acmpeq` and `if_acmpne`. For nullness check and branch, there are `ifnull` and `ifnonnull`.

Finally, there's an unconditional `goto` and a `tableswitch` for compiling a switch statement into.

JVM instruction set: objects and fields

An object is instantiated with the `new` instruction. Note that the bytecode validator will enforce that its constructors are called with `invokespecial` (see next slide).

There are four instructions for accessing fields (though we don't do this too often, as for 6model objects it's encapsulated inside the representation):

- `getstatic` takes a field reference from the constant pool and loads the static field's value onto the stack
- `getfield` is similar, but expects the object to access an instance field from to be on the stack
- `putstatic` takes a field reference from the constant pool and stores the current stack top value to the static field
- `putfield` is similar, but expects the object to store the instance field on to be on the stack beneath the value

JVM instruction set: method calls

Instance method calls expect the stack to contain the object to call a method on, followed by any extra arguments. Note that these operate on **Java objects** rather than 6model objects, so we don't use them for Perl 6's method dispatch!

- `invokevirtual` does a normal virtual method call
- `invokespecial` calls a method in an exact class (used for super, etc.)
- `invokeinterface` calls a method through an interface

There is also `invokestatic` which just expects the arguments to be on the stack. We use this very heavily, since most `nqp::` ops are static method calls.

Finally, there's `invokedynamic`, which is how actual Perl 6-level routine and method calls are wired up.

JVM instruction set: exceptions

There is only one instruction related to exceptions, `athrow`. It throws the exception object that is currently on the stack top.

Exception handlers are stored as a table rather than in the bytecode stream.

JVM instruction set: other bits

There are various coercion instructions that convert between the primitive types. They are of the form 2, with the same one-letter codes used for arrays. The available ones are `i2l`, `i2f`, `i2d`, `l2i`, `l2f`, `l2d`, `f2i`, `f2l`, `f2d`, `d2i`, `d2l`, `d2f`, `i2b`, `i2c`, and `i2s`.

There are also a number of instructions for manipulating the stack:

- **Popping:** `pop`, `pop2` (note that a long or double counts as 2 slots)
- **Duplicating:** `dup`, `dup2` (same rules)
- **Swapping:** `swap` (nope, doesn't exist for long/double)

Finally, a method can be returned from (taking the current stack top as the return value) with `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, or `return` (void).

JAST (1)

To generate JVM bytecode from NQP, we build a bunch of **JAST** nodes (short for JVM Abstract Syntax Tree).

There are some nodes for pushing constants:

```
JAST::PushIVal.new( :value(42) )        # 64-bit integer constant
JAST::PushIndex.new( :value(69) )       # 32-bit integer constant
JAST::PushNVal.new( :value(1.5) )       # 64-bit double constant
JAST::PushSVal.new( :value('beer') )    # String constant
```

There is also a `JAST::PushCVal`, used for pushing class literals.

JAST (2)

The top level structure is made up of a `JAST::Class` node. It exposes the methods `add_field`, which expects a `JAST::Field`, and `add_method`, which expects a `JAST::Method`.

A `JAST::Field` has methods (and named constructor parameters) to set a `name`, `type` and if it's `static`.

A `JAST::Method` is rather more complex. Along with the `name` and a flag to indicate if it's `static`, it also has lists of `locals` and `arguments`, along with a type that it `returns`. Additional fields capture the set of lexicals, NQP-level exception handlers, etc.

JAST (3)

An individual instruction is expressed as a `JAST::Instruction` node.

```
JAST::Instruction.new( :op('aconst_null') )
```

Typically, these are pushed onto a `JAST::InstructionList`, though they can also be pushed onto the instruction list inside a `JAST::Method` too.

```
my $il := JAST::InstructionList.new();
$il.append(JAST::PushIVal.new( :value($target) ));
$il.append(JAST::Instruction.new( :op('aload'), 'tc' ));
$il.append(JAST::Instruction.new( :op('invokestatic'), $TYPE_OPS,
    'lexotic_tc', $TYPE_SMO, 'Long', $TYPE_TC ));
# ...
```

JAST (4)

The `JAST::Label` node represents a label. Within a given `JAST::Method`, a label needs to be unique.

```
my $if_id    := $qastcomp.unique($op_name);
my $else_lbl := JAST::Label.new(:name($if_id ~ '_else'));
```

A `JAST::Label` can be used in a branch:

```
$il.append(JAST::Instruction.new($else_lbl,
    :op($op_name eq 'if' ?? 'ifeq' !! 'ifne')));
```

And its location is wherever it's pushed:

```
$il.append($else_lbl);
```

JAST (5)

Finally, there is `JAST::TryCatch`, which represents a (JVM-level) exception handler.

It expects two `JAST::InstructionLists`, one that makes up the `try`, and another that makes up the `catch`. It also needs to haven an exception `type` specified.

```
$il.append(JAST::TryCatch.new(
    :try($try_il),
    :catch($catch_il),
    :type($TYPE_EX_LEX)
));
```

The QAST to JAST translator

JAST provides a way to produce JVM bytecode from NQP. The frontend produces a QAST tree, however. Between them is a QAST to JAST translator, which lives in the `src/vm/jvm/QAST/` directory of NQP.

In all, including translation of all the QAST nodes (including regexes) and `nqp::ops`, it weighs in at about 5,400 lines. That may sound like a lot, yet it is only around a third of the size of the Perl 6 CORE.setting!

Its job is complicated by a couple of factors:

- It's doing a Continuation Passing Style transform on everything as it goes about doing the code generation - though this is relatively well isolated
- The JVM has stack-must-be-empty constraints on things that appear in the middle of a Perl 6 expression (like `try`)

Types and results

There are 4 primitive types that everything revolves around, which you will see everywhere in the compiler:

```
$RT_INT     a JVM long; maps to ‘int‘ in NQP
$RT_NUM     a JVM double; maps to ‘num‘ in NQP
$RT_STR     the Java String class; maps to ‘str‘ in NQP
$RT_OBJ     a 6model object (org.perl6.nqp.sixmodel.SixModelObject)
```

The code to compile a QAST node or an `nqp::op` always returns a `Result`. This type pairs together a `JAST::InstructionList` with one of the above types, indicating what it leaves behind on the stack.

`$RT_VOID` indicates the absence of a result.

Mapping nqp::ops

A handful of `nqp::ops` map directly to JVM ops:

```
QAST::OperationsJAST.map_jvm_core_op('neg_i', 'lneg', [$RT_INT], $RT_INT);
```

Some others get mapped to functions in the Java class library:

```
QAST::OperationsJAST.map_classlib_core_op('abs_i', $TYPE_MATH, 'abs',
    [$RT_INT], $RT_INT);
```

Most, however, are mapped to a a call on the `Ops` class in the NQP runtime, often passing the current `ThreadContext` object:

```
QAST::OperationsJAST.map_classlib_core_op('create', $TYPE_OPS, 'create',
    [$RT_OBJ], $RT_OBJ, :tc);
```

The runtime support library

Many of the `nqp::ops`, the JVM implementation of 6model, and various other support code lives in the `src/vm/jvm/runtime/` directory in NQP. It is built into the `nqp-runtime.jar` library.

If you modify this code, you typically need only re-build the JAR to see the effects, rather than re-building all of NQP. Copy it to its install location, and it can be updated for Rakudo too.

Rakudo also has a (much smaller) runtime support library, which is built to `rakudo-runtime.jar`.

In summary. . .

The JVM support consists of translating a QAST tree into JVM bytecode, through an intermediate form known as JAST, and a runtime support library.

Most effort goes into providing things that the JVM does not provide natively, such as the continuation support needed by gather/take, 6model, exception handlers that run on the stack top before unwinding, and so forth.

From here, work is needed on optimization, better code generation, and better use of `invokedynamic`.

The MoarVM backend

A VM built just for NQP and Rakudo

MoarVM in a nutshell

Uses **6model** as its native object system

Instruction set aligned with **nqp::ops**

Provides **generational GC** with two generations: a nursery (handled by semi-space copying) and gen2 (sized pools, except for large objects)

Includes **Unicode database** support, and working towards **NFG** strings

Support for **threads**, use of **lock-free** data structures where possible

MAST

MoarVM doesn't have an assembly or intermediate language; instead, MAST (the MoarVM Assembly Syntax Tree) is assembled directly into bytecode.

MAST is lower level than QAST, and rather different to JAST (due to the rather different design of the VMs and the fact that MoarVM isn't stack based). However, if you are familiar with either (or both) of these, you'll feel at home quite quickly with MAST. After all, MAST and JAST were designed by the same person, and said person also contributed to the QAST design. :-)

In fact, 5 MAST nodes share exactly the same name and role as the equivalent QAST nodes.

MAST nodes: literals

Literal integers, floating point numbers, and strings are unsurprising in their representation.

```
MAST::IVal.new( :value(42) )
MAST::NVal.new( :value(1.2) )
MAST::SVal.new( :value('cwrw') )
```

However, while their QAST equivalents can appear essentially anywhere, these MAST nodes can only be used as arguments to instructions that expect literals (for example, `const_i64`, `const_n64`, `const_s`, `argconst_[ins]`, parameter names in the named parameter binding instructions, etc.) Most instructions instead expect the argument to be in a local.

MAST: frames

A lexical scope, and the smallest invokable unit, in MoarVM is a frame. This is represented by a `MAST::Frame` node.

A frame has:

- A (high level) name and a (low level) compilation unit unique ID
- A list of locals/registers (there's no distinction, and the terms are used interchangeably). These just have integer indexes, not names.
- A list of lexicals. These have names and types.
- A reference to its static outer frame (used in lexical lookup)
- A list of instructions

Typically, a `QAST::Block` maps to a `MAST::Frame`.

MAST nodes: locals

There is a `MAST::Local` to represent locals (storage slots available as the frame executes):

```
MAST::Local.new( :index($!frame.add_local($type)) )
```

In reality, it's very rare to see this being directly constructed in the QAST to MAST compiler. It's all hidden behind some helpers:

```
fresh_o     Make a new object local
fresh_i     Make a new int64 local
fresh_n     Make a new num64 local
fresh_s     Make a new string local
```

After use, a local is typically released so it can be re-used elsewhere in the frame.

MAST nodes: lexicals

MoarVM also natively supports lexical variables, which (unlike locals) are visible from nested frames. Again, these nodes are rarely generated directly in the QAST compiler, but by a helper that resolves the lexical (calculating how many frames out to look for it):

```
method resolve_lexical($name) {
    my $block := self;
    my $out   := 0;
    while $block {
        if ($block.lexicals()){$name} -> $lex {
            return MAST::Lexical.new( :index($lex.index),
                                      :frames_out($out) );
        }
        $out++;
        $block := $block.outer;
    }
    nqp::die("Could not resolve lexical $name");
}
```

MAST nodes: ops

A `MAST::Op` node represents an operation from the MoarVM instruction set. These are often created and pushed onto an instruction list by the `push_op` helper sub:

```
sub push_op(@dest, $op, *@args) {
    nqp::push(@dest, MAST::Op.new(
        :op($op),
        |@args
    ));
}
```

The kinds of nodes expected as arguments varies with instruction. For example, the `push_o` instruction expects two `MAST::Local` nodes:

```
push_op($arr.instructions, 'push_o', $arr_reg, $item_reg);
```

MAST nodes: labels

A `MAST::Label` can be placed in an instruction list as the target of a branch and used as an argument to certain `MAST::Ops` that branch.

A label must be unique within a given `MAST::Frame`, which is why you'll often see code like:

```
my $if_id    := $qastcomp.unique($op_name);
my $else_lbl := MAST::Label.new(:name($if_id ~ '_else'));
my $end_lbl  := MAST::Label.new(:name($if_id ~ '_end'));
```

Here's some examples of using the labels:

```
push_op(@ins, 'goto', $end_lbl);
nqp::push(@ins, $else_lbl);
```

MAST nodes: calls

Making a call boils down to a number of steps: getting the arguments to pass into an arguments buffer, setting the callsite descriptor, indicating the result register, and making the call itself.

This is abstracted behind the `MAST::Call` node:

```
nqp::push(@ins, MAST::Call.new(
    :target($callee.result_reg),
    :flags(@arg_flags),
    |@arg_regs,
    :result($res_reg)
));
```

The flags indicate register type, as well as named and flattening arguments.

MAST nodes: exception handlers

Exception handlers are used both for control flow (such as next/redo/last in loops) or true exceptions (caught by `CATCH` blocks in NQP/Perl 6). Both of these are set up with a `MAST::HandlerScope`, which indicates the instructions covered by the handler, what kind of exception it's interested in and what to do if the handler is triggered.

```
MAST::HandlerScope.new(
    :instructions(@loop_il),
    :category_mask($HandlerCategory::redo),
    :action($HandlerAction::unwind_and_goto),
    :goto($redo_lbl)
)
```

Here, the action is to simply unwind the call stack and go to the specified label. By contrast, a `CATCH` block's action is to run a handler block on the stack top and unwind afterwards.

MAST nodes: compilation units

Finally, the top of a MAST assembly tree is always a `MAST::CompUnit`. This has a list of frames (each one added with the `add_frame` method).

Certain frames can be called out as special:

- `deserialize_frame` holds code that drives deserialization, and will always be run when the compilation unit is created or loaded
- `load_frame` holds code that should run when the compilation unit is loaded as a module
- `main_frame` holds code that should run when the compilation unit is the initial entry point

It also keeps track of the HLL that produced the compilation unit and the set of serialization contexts that it depends on.

The QAST to MAST translator

Spread over three files:

- `QASTOperationsMAST.nqp` handles compilation of `nqp::ops`
- `QASTRegexCompilerMAST.nqp` handles compilation of `QAST::Regex` nodes
- `QASTCompilerMAST.nqp` handles the rest

These reference:

- The MAST nodes
- Meta-data about all of the ops available and the kinds of registers they work on

MAST::InstructionList

Once again, there is a data structure used to convey the result of compiling a QAST node: `MAST::InstructionList`. It holds three pieces of information:

- A list of instructions (`$il.instructions`)
- The register (local) holding the result (`$il.result_reg`)
- The kind of result register it is (‘$il.result_kind)

There are constants for the four main kinds:

- `$MVM_reg_obj` (6model object)
- `$MVM_reg_int64` (int)
- `$MVM_reg_num64` (num)
- `$MVM_reg_str` (str, though it's actually a 6model object too)

Register/local allocation

There is a per-block `$*REGALLOC` that keeps track of register use. Despite the name, it's not doing register allocation in the traditional sense (such as by graph coloring). Rather, it keeps track of available temporaries, enabling them to be re-used.

Obtaining a new register to work with is typically done as:

```
my $callee_reg := $*REGALLOC.fresh_o(); # also _i, _n, _s
```

It can then be released when it's no longer needed:

```
$*REGALLOC.release_register($callee_reg, $MVM_reg_obj);
```

nqp::op mapping

Many `nqp::ops` have similar or identical names in the MoarVM instruction set. The operand type data is also readily available, so does not need to be specified in the mappings:

```
QAST::MASTOperations.add_core_moarop_mapping('atpos', 'atpos_o');
QAST::MASTOperations.add_core_moarop_mapping('atpos_i', 'atpos_i');
```

Some instructions in MoarVM are void, but are allowed in an r-value context as `nqp::ops`. Therefore, we pick one of the input operands as the result, if one is needed.

```
QAST::MASTOperations.add_core_moarop_mapping('bindpos', 'bindpos_o', 2);
QAST::MASTOperations.add_core_moarop_mapping('bindpos_i', 'bindpos_i', 2);
```

Inside MoarVM

The top-level `src` directory doesn't contain much directly; the code is categorized into sub-directories:

- `6model` contains 6model, implementations of the REPRs, serialization...
- `core` is the heart of the VM, containing the interpreter, argument handling, bytecode decoding, thread handling, invocation, exceptions...
- `gc` is where memory allocation and garbage collection lives
- `io` contains IO-related functionality, typically delegating the real work to `libuv`
- `mast` contains the MAST to bytecode compiler
- `math` contains the libtommath binding for big integer support
- `platform` is where platform-specific code goes (a different platforms do things differently)
- `strings` contains string operations, encoding/decoding of ASCII, UTF-8, etc.

In summary. . .

MoarVM **uses 6model** as its object model and has an instruction set that is **well aligned** with the `nqp::op` set. As a result, the mapping from QAST down to it is comparatively straightforward.

It will also be the first place that we support NFG strings, and should also get good Perl 5 interop.

Future developments will include 6model-aware JIT compilation, which should give a notable performance boost.

The Road Ahead

This isn't the end??!!!

This isn't all...

We've covered a lot of ground in these two days.

Naturally, there are things that have been put aside. We haven't looked at every line of code of every file!

And, of course, we didn't cover the things not invented yet because we didn't implement the bits of the Perl 6 spec that need them.

However, **we have covered all of the key parts that make up NQP and Rakudo**. With careful reading of code and a little digging, it should be possible to work out what most of NQP and Rakudo do, and where most things are found.

The toolchain will evolve

The NQP toolchain has **evolved in response to understanding Perl 6's needs**. As we continue to learn, this knowledge will be crunched into the tools.

In the past, there have been some fairly dramatic overhauls. These are very likely over, though there are surely more lessons that can be turned into **better abstractions and APIs**.

For example, the concurrency/parallelism work is currently done in terms of classes from the Java Class Library directly. However, in time, the key abstractions may well be captured into `nqp::` ops and so forth.

Remember. . .

Compilers aren't magical. They're just software.

Perl 6 is a large language, and implementing it is non-trivial. However, NQP and Rakudo have made a reasonable job of trying to **manage the complexity** by breaking the problem into decoupled pieces.

In fact, that's the only thing that keeps it manageable at all. Keep this in mind as you hack. **Good architecture takes discipline.** The first solution you think of will rarely be the best one. Things that feel wrong, usually are.

Take pride in solving implementation problems elegantly, ask questions, treat no code as sacred, and be sure to `-Ofun`.

Thank you!

Thanks for atttending the course!

Any final questions?

By the way, at Edument AB we've also built and deliver courses on...

- Perl 5
- Git
- Software architecture and Domain Driven Design
- JavaScript and other web technologies
- C# and .Net
- Test Driven Development

For more, see http://edument.se/courses/.