Rakudo and NQP Internals Exercises

Jonathan Worthington

September 16, 2013

1 Exercise 1

This exercise will get you to do some simple tasks in NQP, to get used to the basic syntax.

1.1 Hello, world

Easiest thing ever:

```
say('Hello, world');
```

Now get it wrong:

```
say 'Hello, world';
```

Notice how you will be complained at. Loudly.

1.2 Variables

1. Bind a value to a scalar.
2. Use `say` to output it.
3. Try to use assignment instead. Observe that it fails.
4. Declare an array. Use indexing, like `@a[0]`, to bind and access elements. Note that in Perl 6, sigils are invariant; `$a[0]` complains about an undeclared `$a`!
5. Try binding an array literal, like `[1,2,3]` to an `@arr` variable. Now do the same to a `$scalar` variable. Note how indexing works just fine with both.
6. Declare a hash variable, `%h`. Try out both literal indexing (`%h<key>`) and the curly syntax (`%h{'key'}`).

1.3 Loops

1. Use `for` to iterate over an array one element at a time, first using `$_`, then with a pointy block (`for @a -> $val { ... }`).
2. Can you iterate over an array two elements at a time? Try it!
3. Iterate over a hash variable, printing out the `.key` and `.value` of each pair in the hash.

1.4 Subroutines

Implement a recursive factorial routine, `sub fac`. If you get it right, `fac(10)` will return 3628800.

1.5 Classes

1. Declare a `BarTab` class. Give it a scalar attribute for the table number and an array attribute for items.
2. Write a method `table` that returns the table number.
3. Write a method `add_order` that takes an item name and a price. Push a hash onto the items attribute with the keys `name` and `price`.
4. Write a method `render_tab` that produces a string describing what's on the tab (items, price for each item, total)
5. Create a tab (`BarTab.new(table => 42)`). Check the `table` method returns the 42. Add some items and make sure things work. You may even like to use `plan(...)` and `ok(...)` - built in to NQP - to make these tests!

1.6 Multi-methods

1. Add a `proto` method for `add_order`. Then make the current `add_order` a multi-method (just add the keyword `multi` before it). Ensure that all still works.
2. Add another multi candidate to `add_order` that takes 3 arguments, the third being a quantity. For this candidate, add an item hash to the array the specified number of times.

1.7 If time allows...

1. Write a lexical class `Item` (a class declared with `my` inside of your `BarTab` class. Give it attributes for `name` and `price`, and methods to access them.
2. Refactor your code to use this inner class, so you have an array of `Item` rather than an array of hashes.
3. Refactor your code so that instead of an item appearing in the items list multiple times if it is ordered more than once, each item has a quantity. (You may like to switch to using a hash instead of an array in your `BarTab`, so you can look up items by name and just update the quantity of ones that were already ordered.)

2 Exercise 2

2.1 Grammar

In plain text form, a log from the Perl 6 IRC channel looks like this:

```
14:30 colomon     r: say 2**100
14:30 camelia     rakudo 624ff7: OUTPUT1267650600228229401496703205376
14:30 colomon     sweet!!!!!
14:30 TimToady    bet that took a lot of electrons...
```

That is, a time, a nick, and some text. Start out by writing a grammar that parses any number of lines like this, capturing the time, nick and text. Go over the match object and check it has the right output.

2.2 Actions

Declare an `Utterance` class to represent a line of chat. It should have three attributes: time, nick, and text, along with methods that access them. You can rely on the default constructor for setting them.

Now write an actions class that builds an instance of this class for each line of text. Finally, add a `TOP` action method that produces an array.

2.3 Bonus Task: Actions

As well as normal chat lines, there are action lines, which have a `*` before the nick:

```
07:26 moritz      r: say (1, 2, 3).map({()}).flat.perl
07:26 camelia     rakudo 099f0b: OUTPUT().list
07:26 * moritz    wonders if .grep itself flattens
```

Add support for parsing this, and recording it in the `Utterance` as an extra attribute.

2.4 Bonus Task: Collect lines

Replace the text attribute in `Utterance` with an array of lines. Then, if a single nick speaks multiple consecutive lines of text, put them into a single array and create a single instance of `Utterance` for them.

3 Exercise 3

3.1 Familiarization

Take the SlowDB implementation. Review the code we've seen so far, and run a few simple `INSERT` and `SELECT` queries.

3.2 DELETE

Implement functionality for deleting rows. Examples of `DELETE` queries are:

```
DELETE WHERE name = 'jnthn'
DELETE WHERE name = 'jnthn', is_action = 1
```

That is, there can be one condition or a conjunction of conditions separated by commas.

3.3 UPDATE

Implement functionality for updating existing rows. Examples of `UPDATE` queries are:

```
UPDATE WHERE type = 'stout' SET tasty = 1
UPDATE WHERE type = 'stout' SET tasty = 1, dark = 1
UPDATE WHERE type = 'stotu' SET type = 'stout'
```

3.4 Refactor

Now there are three things that support `WHERE` clauses: `SELECT`, `DELETE` and `UPDATE`. Can you factor that out?

3.5 If you're working really fast...

Extend the range of operators usable in a `WHERE` clause to include `!=`, `<`, `<=`, `>` and `>=`.

4 Exercise 4

Create a minimal PHP compiler that simply supports `echo` statements:

```
echo "<h2>PHP is fun!</h2>";
echo "<blink>So 1990s!</blink>";
```

4.1 The basics

1. Create a new NQP source file, and `use NQPHLL`.
2. Create subclasses of `HLL::Grammar`, `HLL::Actions` and `HLL::Compiler`.
3. Write a `MAIN` sub that, just like the Rubyish one, sets up the compiler object and calls `command_line`. Verify that running your program with no arguments now enters the REPL, but that attempting to run any code fails because the grammar has no `TOP` rule.
4. Do Just Enough in the grammar to be able to parse a list of `echo` statements. The final one need not be followed by a semicolon. Run the REPL again and ensure your grammar parses, and that the error you get is because no AST is being built.
5. Write action methods to make QAST. Make sure your program now runs.

4.2 Going further

Time to be more PHP-ish!

1. The `echo` statement doesn't automatically put a new line at the end of the output, so use the `print` op instead of `say`.
2. Turn on parsing of backslash sequences in your quote parser. This is done by <`quote_EXPR: ':q', ':b'`> (note the `:b`). Make sure that you can now `echo "stuff\n"` with a newline on the end or not.
3. It is also possible to use `echo("with parentheses")`. Implement it.

5 Exercise 5

In this exercise, you'll add a few features to PHPish, in order to explore the QAST nodes we've been studying. By the end, you should be able to run the following program:

5.1 Values

So far, `echo` can only handle a string argument. It's time to change that! Add a `value` protoregex, and candidates for parsing integers, floating point numbers and strings. Be sure that your string token looks as follows:

```
token value:sym<string> { <?["]> <quote_EXPR: ':q', ':b'> }
```

So that you'll be able to handle the `\n`. Add the appropriate action methods (remembering that the protoregex itself doesn't need one). Update the `echo` rule and action method to take a `value`, not just a string. Ensure you can run the same things as before, and also:

```
echo 42;
echo 1.5;
```

5.2 Operators

Study the operator precedence parsing material. Set up two precedence levels, one for additive and one for multiplicative operators. Add the four usual arithmetic operations (`+`, `-`, `*`, `/`) as in Rubyish. Also add `.` for string concatenation (the NQP op for that is called `concat`).

Next, add a candidate to the `term` protoregex, which should call `value` to parse a value. This needs a matching action method.

Finally, update `echo` to parse an `EXPR` instead of a `value`. You should now be able to run:

```
echo 2 * 5 . "\n";
```

5.3 Variables

Curiously enough, PHP locals have similar declaration semantics to Ruby: first assignment declares. There are some more interesting rules surrounding globals, but we'll put those aside for now.

First, add a precedence level for assignment, and an assignment operator that is mapped to the `bind` op. Also, add a grammar rule and action method so that an expression can serve as a statement. After this, you should be able to parse:

```
1 = 1
```

However, it's nonsense so it will give an error during code generation.

We're going to need to care about block structure a little more now. Update your `TOP` grammar rule to declare a `$*CUR_BLOCK` like we did in Rubyish, and do the matching update to the action method.

Next, implement variable parsing. Once again, you can take inspiration from what we did in Rubyish, but the variable naming rules are as follows:

- A variable starts with the $ sign, followed by the name of the variable
- A variable name must start with a letter or the underscore character (not a number)
- A variable name can only contain alphanumeric characters and underscores (A..Z, a..z, 0..9, and _)

Also remember that PHP is not line-oriented. Once done, you should be able to execute the following program:

```
$emotion = "awesome";
echo "PHP is " . $emotion . "\n";
$emotion = $emotion . "st";
echo "Perl 6 is " . $emotion . "\n";
```

5.4 Bonus task: functions

Implement functions with parameters and argument passing! Once done, you should be able to run:

```
function greet($name) {
    echo "Good day, " . $name . "\n";
}
greet("Lena");
```

You can steal some inspiration from the Rubyish example. However, you'll also need to take some care to make sure that function declarations do not need a semicolon after them.

6 Exercise 6

This exercise is to fill time at the end of the first day, if we somehow make it through everything else! Here are some ideas of what you could do.

6.1 Basic numeric relational operators

For now, we'll ignore the type juggling done by PHP's relational operators to cope with both strings and numerics, and just handle the numeric case. PHP has two precedence levels for comparision operators:

```
Tighter level:    < <= > >=
Looser level:     == !=
```

These are simultaneously looser than additive and tighter than assignment. No action methods should be needed, since `:op<...>` can be used in the `<O(...)>`.

6.2 if/else if/else

Now that we have relational operators, it makes sense to implement `if` statements. Start out simple:

```
if ($a > $b) {
  echo "a is bigger than b";
}
```

Then build up to having `else`:

```
if ($a == $b) {
  echo "a is the same as b";
}
else {
    echo "a and b differ";
}
```

Finally, add `elseif`:

```
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
```

To better understand how `elseif` can be handled, read the applicable code in NQP's grammar and actions. Search for `statement_control:sym<if>` in both.

Last but not least, you may even like to try implementing the dangling form:

```
if ($a > $b)
  echo "a is bigger than b";
```

It appears that there are other wonderful irregularities to be had in the `elseif` and `else` forms too, involving colons and who knows what. But it's probably better to go drinking if you got this for. Or, do...

6.3 while loops

Parse and implement while loops:

```
$i = 1;
while ($i <= 10) {
    $i = $i + 1;
    echo $i;
}
```

Congratulations. Your PHP implementation is now Turing Complete, and all is now possible with it!

(Aside: It was actually Turing complete sooner, since recursive functions and `while` loops are equivalent.)

7 Exercise 7

In this exercise, you'll add basic support for classes, methods, the `new` operator and method calls to PHPish. When you're done, you should be able to run:

```
class Greeter {
    function greet($who) {
        echo "Greetings, " . $who . "\n";
    }
}
$g = new Greeter();
$g->greet("Anna");
```

7.1 First, a meta-object

Create a simple meta-object, much like the Rubyish one, that can support adding and looking up methods.

7.2 Next, parse classes and generate code for them

The basics are similar to Rubyish: functions within a class become methods on the class. Thus, you can steal the approach, making sure to use PHP syntax for it.

7.3 Add the new term

This is so similar, you can steal it from Rubyish almost directly. Just be aware of the different whitespace rules (line breaks aren't significant).

7.4 Add method calls

Once again, a similar approach works, modulo syntax: the method call operator is `->`. After adding this, you should be able to run the program at the start of this section - but it will still fail. The problem is that we did not make methods accept the invocant (the object they are called on).

7.5 Support $this

The first parameter of a method is the object it is called on. In PHP, this is assigned to `$this`. Therefore, when in a class, ensure that a function always takes a `$this` parameter *first*.

7.6 Well, that was a lot of copy-paste...

Yup. Most class-based object systems are relatively similar at this level. Still, hopefully being able to put it together for yourself will have made the whole thing seem less magical!

By the way, now you support `$this`, this should work too:

```
class Greeter {
    function greet($who) {
        $this->greet_custom("Hej", $who);
    }
    function greet_custom($greeting, $who) {
        echo $greeting . ", " . $who . "\n";
    }
}
$g = new Greeter();
$g->greet("masak");
```

8 Exercise 8

In this exercise, you'll extend the PHPish object system a little... or maybe a lot.

8.1 Method cache

First, create a new PHP class, manually, in NQP, with a method. It will look something like this:

```
my $c := PHPClassHOW.new_type(:name('Foo'));
$c.HOW.add_method($c, 'bar', -> $obj { 1 });
```

Time how long 100,000 method calls take.

```
my $start := nqp::time_n();
my int $i := 0;
while $i < 100000 {
    $obj.bar();
    $i++;
}
say(nqp::time_n() - $start);
```

Add a `compose` method to your `PHPClassHOW`. In it, use the `setmethcache` instruction. You can use `%!methods` to serve as the method cache hash for now. Add a call to your `compose` method. Repeat the timings. It should be faster with the cache (on the exercise writer's machine, 0.194s without and 0.066s with).

Now, update the code in the `statement:sym<class>` action method so that the `compose` method is called after all the methods have been added.

8.2 Inheritance

Here's the example we want to make work:

```
class Dog {
    function bark()  { echo "woof!\n"; }
    function smell() { echo "terrible\n"; }
}
class Puppy extends Dog {
    function bark()  { echo "yap!\n"; }
}
$d = new Dog();
$d->bark();
$d->smell();
$p = new Puppy();
$p->bark();
$p->smell();
```

First, start out with the parsing. It's a very small addition to parsing a class declaration, to optionally accept an `extends` keyword. Once it parses, you should be able to run the program, but it will fail to do the final call to `smell`, as that is inherited and we don't have inheritance yet.

Next, update the action method for a class. Before the code for adding methods, if there is an `extends`, add a call on the meta-object to `add_parent` (which you will write in a moment). To look up the parent, use a `QAST::Var` node with the scope set to `lexical`.

You should now get an error saying that `add_parent` is not yet implemented. From here, all the work will be in the `PHPClassHOW`.

Add an attribute `@!parents`. Implement an `add_parent` that will add to this array, but only if there is not a parent there already. If there is, then `nqp::die` about it. This should clear up the error, but again, the final call to `smell` does not yet work.

Next, update `find_method`. If the method to search for is not in `%!methods`, it should look at the method tables of all parents. For this, you will need to implement methods `parents` and `method_table` to enable you to get at the required information from base classes.

By this point, things should work - but not efficiently! For that, you should now also update the method cache. You can make sure you did so correctly by adding a debugging statement in your `find_method`, which should not be called if the cache is working.

8.3 If time allows: interfaces

In PHP, an interface represents a demand that a certain list of methods are provided by any class that declares it implements it. For example, given:

```
interface iBarkable {
    function bark();
}
```

Then this class is fine:

```
class Dog implements iBarkable {
    function bark()  { echo "woof!\n"; }
    function smell() { echo "terrible\n"; }
}
```

While this should cause an error:

```
class BadDog implements iBarkable {
    function smell() { echo "terrible\n"; }
}
```

Since it is missing the `bark` method. Your goal is to implement this feature. Here are a few hints.

1. Start by adding parsing for interfaces and method demands inside of them.
2. Once that works, write a `PHPInterfaceHOW` that can be used to keep a list of required methods. Use the `Uninstantiable` REPR, since you should not be allowed to make an instance of an interface.
3. Add the `implements` keyword parsing to `class` declarations. Set up the action methods to call an `add_interface` method on the `PHPClassHOW`, which should add the interface to an `@!interfaces` attribute.
4. You should now find that the above examples both compile, but the second one is not rejected. To make this happen, in the `PHPClassHOW`'s `compose` method, write code that gets the list of demanded methods for each of the interfaces and check that the class provides them.

9 Exercise 9

This exercise will take you on a short tour of the regex implementation.

9.1 Parsing and actions

Run the following:

```
nqp --rxtrace -e "/a+/"
```

Now, let's try to make sense of it.

In the NQP repository, open `src/NQP/Gramamr.nqp` and locate `quote:sym</ />`. Observe that it calls `LANG`, which does a language switch. This can be found in `src/HLL/Grammar.nqp`. Notice how it uses the braid, and switches actions as well as language type.

Next, open `src/QRegex/P6Regex/Grammar.nqp` and locate `nibbler`. Try to pick out the call to `termaltseq`. Look at `termaltseq` itself, then look in the Actions file for its action method. Work your way down to `termish`, and try to understand what it's doing here: handling the relative precedence of the different alternations and conjunctions.

Next, look down towards `atom`. Notice how the `a` from our regex is parsed here. Then look back up to `quantified_atom`. Read the action method and see how the atom is placed into the quantifier node.

You may wish to explore how some other constructs are handled.

9.2 Exploring embedded code blocks

In the NQP grammar, look for `NQP::Regex`. Notice that it is a subclass of the basic `QRegex::P6Regex::Grammar` you were just looking at. This is part of the power of grammars really being classes: we can subclass them to specialize them.

Let's consider how `/[a { say('here') }]+/` is handled. This is interesting, as we start out in NQP, then parse regex syntax, then end up back parsing NQP code again inside the block. Locate `assertion:sym<{ }>`, and then look at `codeblock`. Enjoy the relative simplicity of what is going on!

Take a look at the `codeblock` action method, in `NQP::RegexActions`. Can you work out what is going on? Ask your teacher, or those around you, if not!

9.3 Exploring NFAs

The saved form of an NFA is stored within a rule's code object. Therefore, we can obtain it and get back to an object as follows:

```
grammar Example {
    token t { 'foo' }
}
```

```
my $t   := nqp::findmethod(Example, 't');
my $nfa := QRegex::NFA.from_saved($t.NFA());
```

But what to do with it? Next, open up `src/QRegex/NFA.nqp`. Look at the list of edge type constants near the top. The overall structure of an NFA is a list of lists. The outer lists represent states - nodes in a graph. Each node is made up of a list of its outgoing edges, which are represented by 3 values:

- An action, which is one of the edge type constants at the top of NFA.nqp
- An argument to the action
- The index of the state to go to next if we match

We can write some code to dump this:

```
my $i := 0;
for $nfa.states -> @edges {
    say("State $i");
    for @edges -> $act, $arg, $to {
        say("    $act ($arg) ==> $to");
    }
    $i++;
}
```

For our simple example we get:

```
State 0
State 1
    2 (102) ==> 2
State 2
    2 (111) ==> 3
State 3
    2 (111) ==> 0
```

Edge type 2 means codepoint, and here the argument is the character code that we should match if we're to go to the next state. See how we start in state 1, and reaching state 0 is success.

Experiment with some different things inside the token to see what the NFA looks like. Here are some ideas.

```
a+
<[abc]>
<[abc]>*
<foo>?
\w*
:i abc
:i a+
```