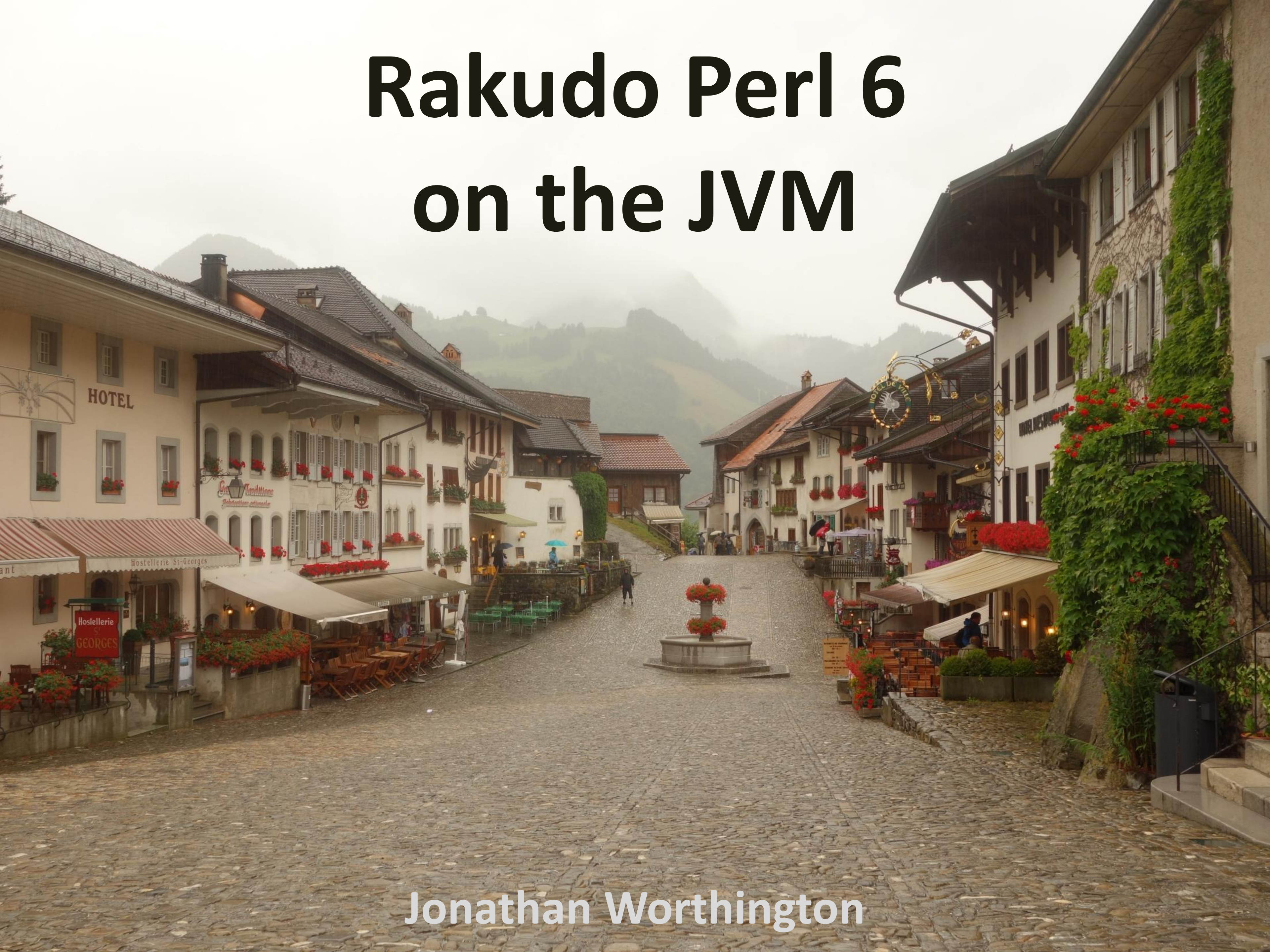
Rakudo Perl 6 on the JVM



Jonathan Worthington

About Rakudo

Most complete and most actively developed Perl 6 implementation

Compiler + built-ins

66 monthly releases to date

10-20 code contributors per release

(but we draw on many other contributions too: bug reports, test suite work, etc.)

About the JVM

JVM = Java Virtual Machine

Runtime originally built for the Java language, but now plays host to dozens of others

Heavily optimized, solid threading support, battle hardened, and widely deployed

Lots of libraries, frameworks, etc.

"Isn't the JVM for static languages?"

It's long been feasible, even if not convenient, to target the JVM for dynamic languages

These days, serious interest from JVM developers

Use `invokedynamic` instruction to teach the JVM how your language does dispatch, invocation, etc.

Perl 6 is neither statically nor dynamically typed, but rather gradually typed

What runs on the JVM?

✓ **Java (of course!)**

What runs on the JVM?

✓ **Java (of course!)**

✓ **COBOL**

What runs on the JVM?

- ✓ Java (of course!)
- ✓ COBOL
- ✓ JavaScript
- ✓ Python
- ✓ Ruby
- ✓ Tcl
- ✓ Lua

What runs on the JVM?

- ✓ Java (of course!)
- ✓ COBOL
- ✓ JavaScript
- ✓ Python
- ✓ Ruby
- ✓ Tcl
- ✓ Lua

☹ So, where is Perl? ☹

Not a new realization

Patrick Michaud, Rakudo Perl 6 pumpking, was speaking with Jesse Vincent, a former Perl 5 pumpking, at YAPC::NA in Pittsburg in 2009

All of the major scripting languages except Perl have implementations on JVM and .NET.

Perl 6 is Perl's best (only?) hope for running on JVM/.Net.

"Run anywhere"

Once, this was just about running on a wide array of operating systems and CPU architectures

Perl 5 is very good at this

However, today some of the "anywhere"s are virtual machines

Perl 6's split of specification and implementation are better suited to cope with this

Other motivations

Rakudo on Parrot is often annoyingly slow

Being able to run on well tuned VM with good profiling tools should provide either better performance and/or better understanding of performance problems (hopefully both!)

Also wanted a solid base to explore and solidify the spec around the parallel and asynchronous parts of the Perl 6 language ➔ JVM can help here

But how to get there?

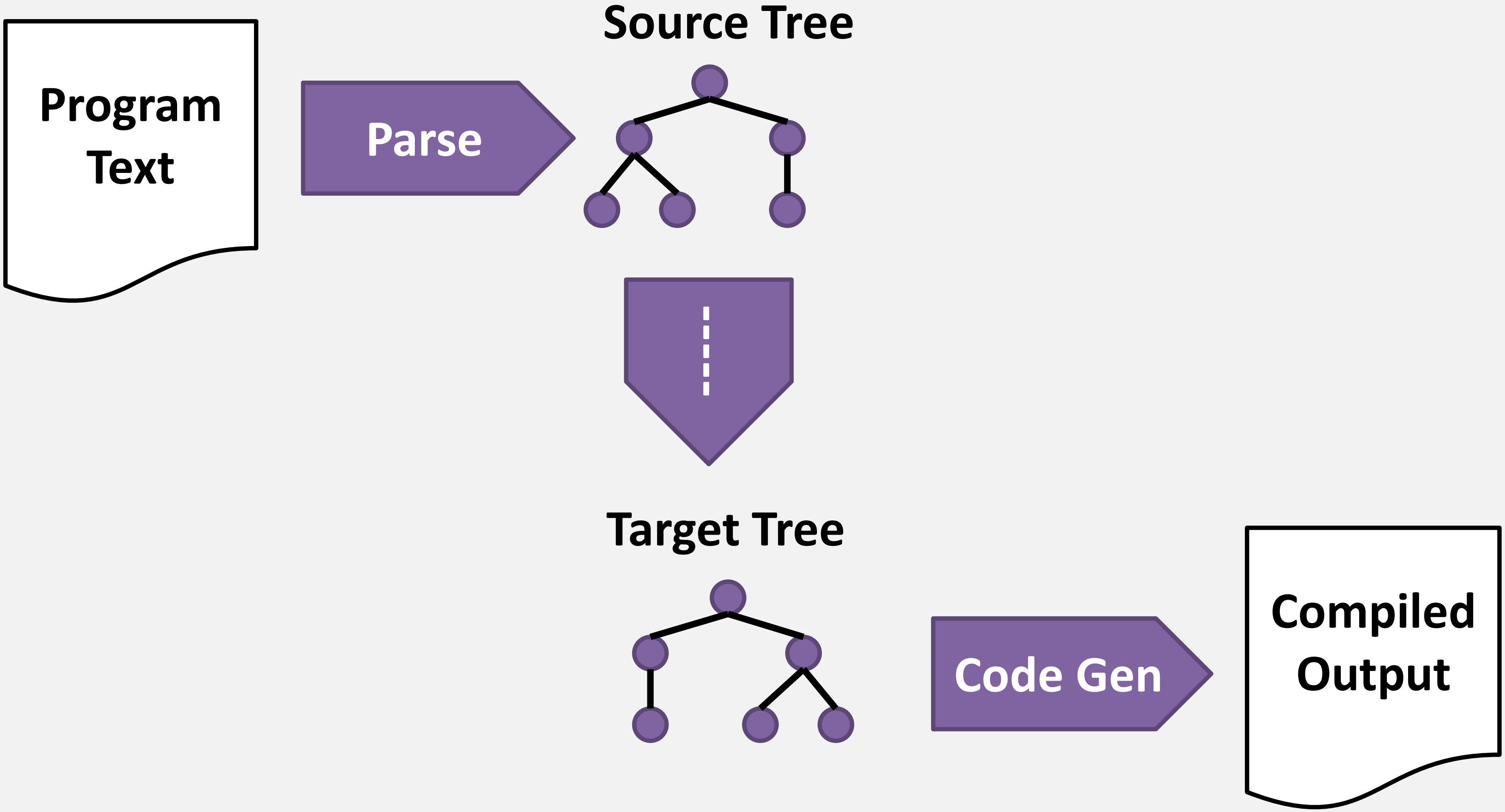
Ruby and Python both have "original" C implementations and separate JVM implementations (JRuby, Jython)

Perl 6, like Perl 5, is a large language and is not easy to implement

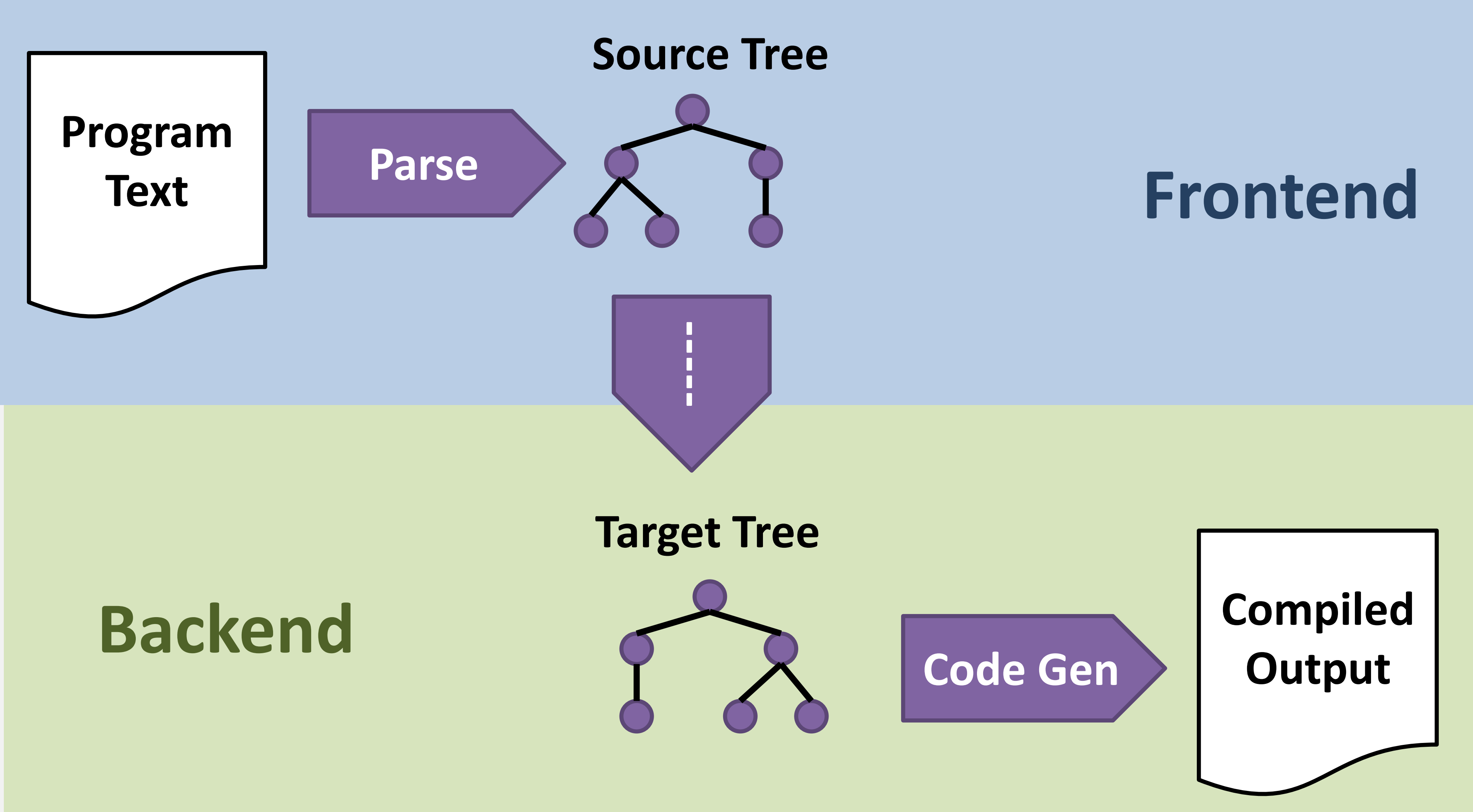
Starting from scratch is costly

What about making Rakudo target the JVM too?

What a compiler does



What a compiler does



Program Text

Parse

Source Tree

Frontend

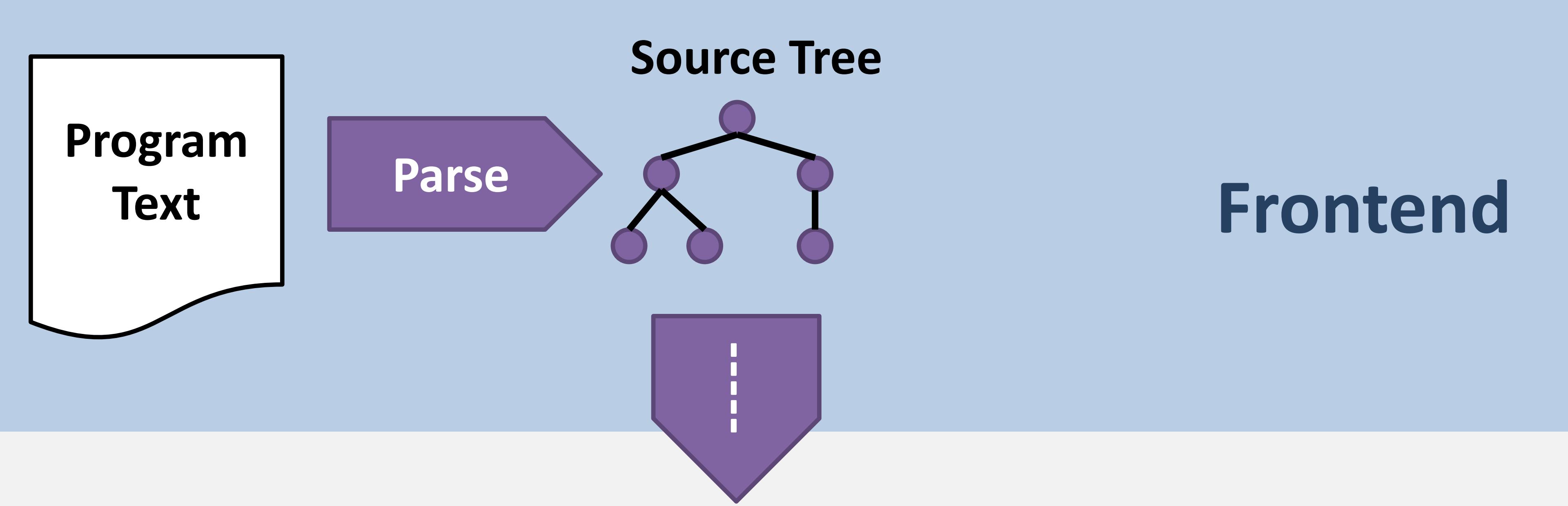
Target Tree

Backend

Code Gen

Compiled Output

The frontend



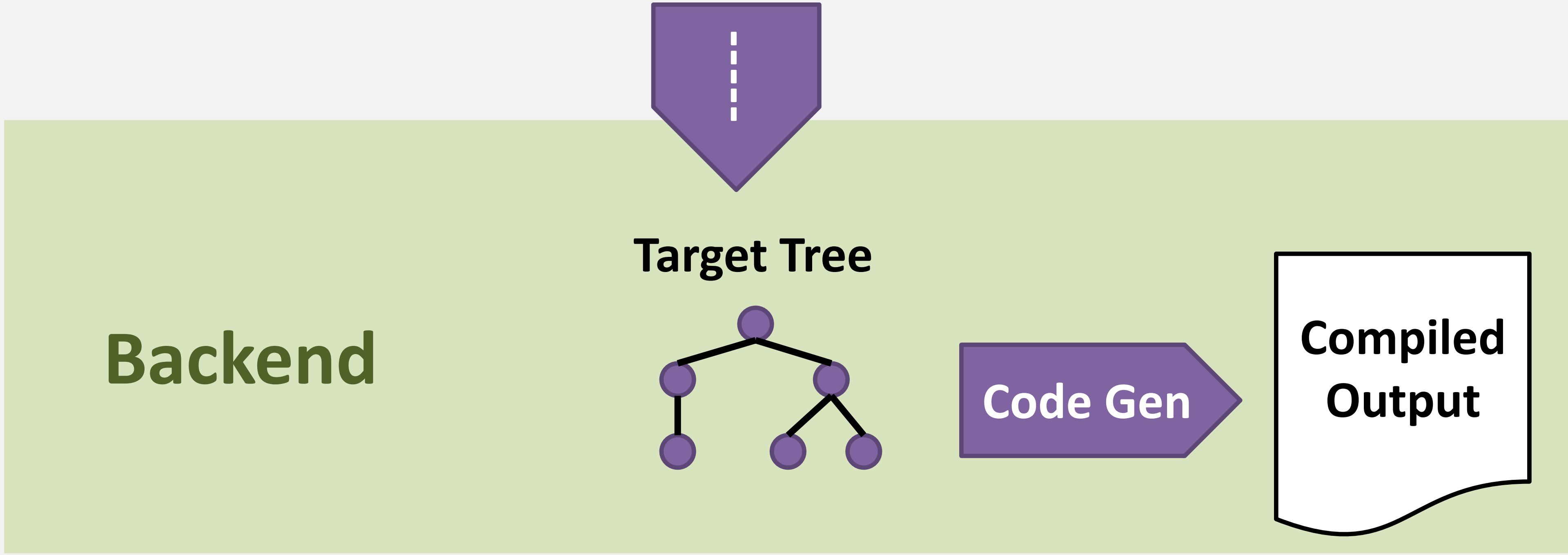
All about a specific language

Syntax, runtime semantics, declarations...

The backend

All about the target runtime

Map HLL concepts to runtime primitives



Rakudo compiler architecture

Loosely coupled sequence of stages that...

Take a well-defined data structure as input

and

Produce a well-defined data structure as output

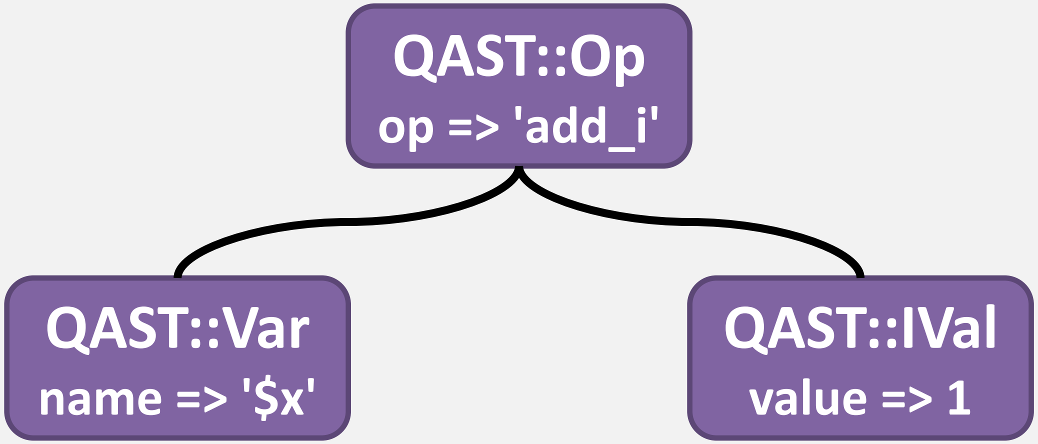
Each stage may be relatively complex. However, it is also completely self-contained.

An FP design, factored OO-ly.

QAST ("Q" Abstract Syntax Tree)

The data structure used to communicate between frontend and backend

A tree with around 15 node types



QAST::Op
op => 'add_i'

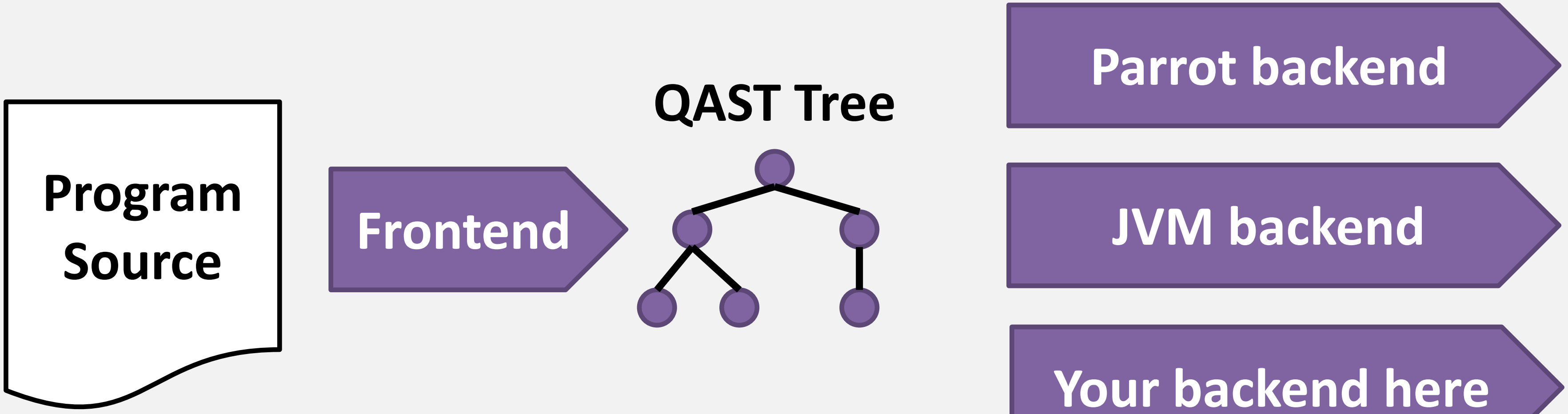
QAST::Var
name => '$x'

QAST::IVal
value => 1

The plan

Tease out places where the frontend was overly-coupled to the Parrot backend

Then add a JVM backend



But wait, what about the compiler?

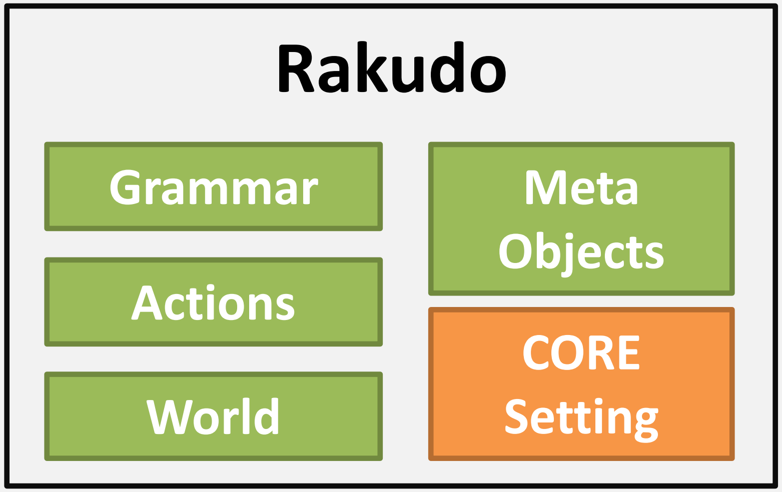
It's all well and good to get Rakudo to target the JVM, but what about `eval`?

Thankfully, Rakudo is written in NQP, a Perl 6 subset

Furthermore, NQP is written in itself

Can use an NQP to JVM compiler to build both NQP and Rakudo on the JVM!

Overall architecture



Rakudo

Grammar

Actions

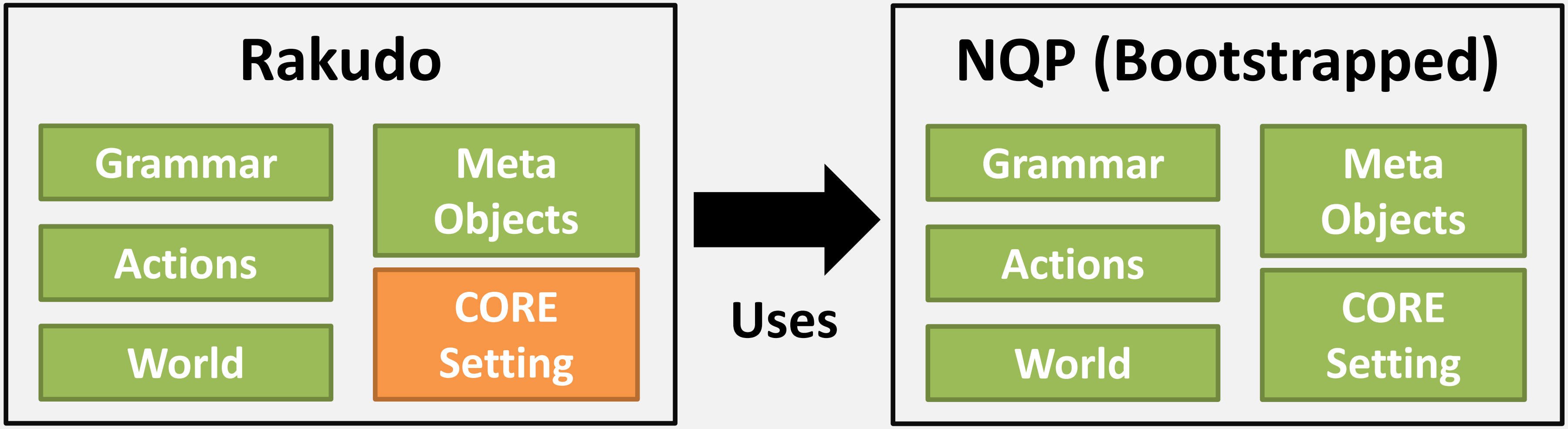
World

Meta Objects

CORE Setting

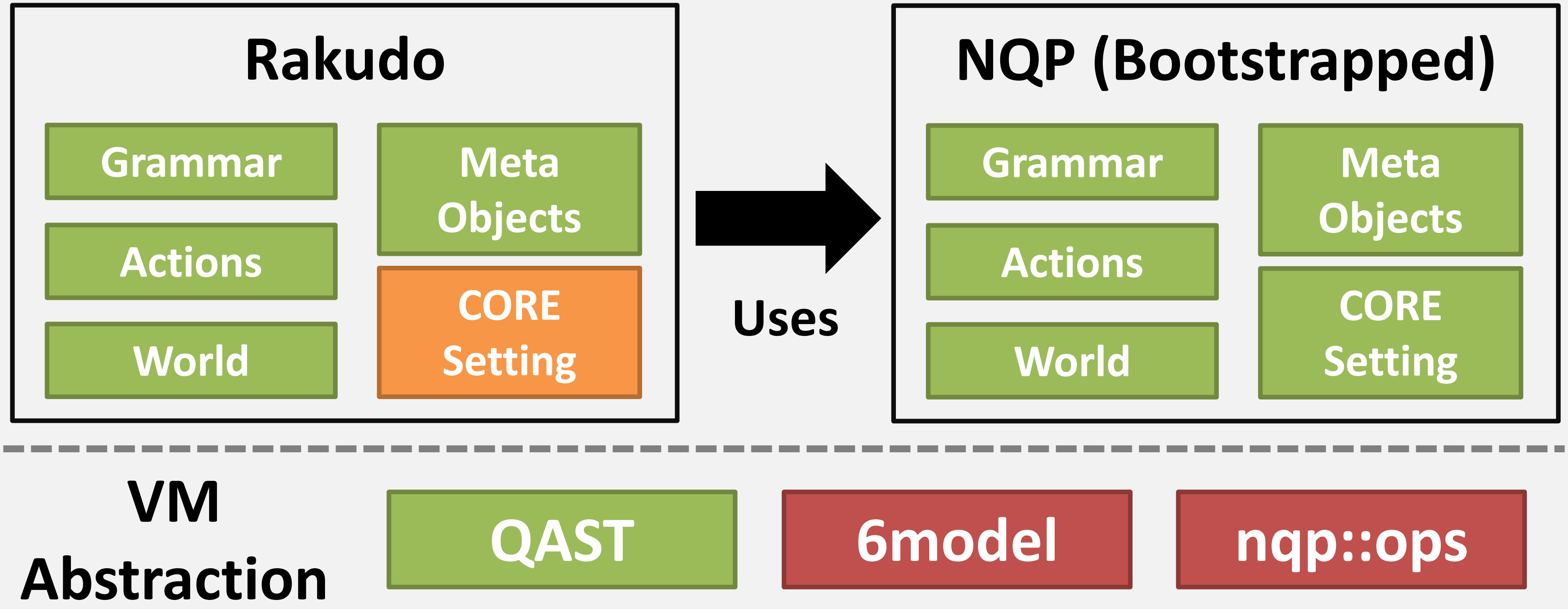
🟩 NQP 🟧 Perl 6 🟥 VM Specific Code ⬛ VM

Overall architecture



🟩 NQP 🟧 Perl 6 🟥 VM Specific Code ⬛ VM

Overall architecture



Rakudo

Grammar

Actions

World

Meta Objects

CORE Setting

Uses

NQP (Bootstrapped)

Grammar

Actions

World

Meta Objects

CORE Setting

VM Abstraction

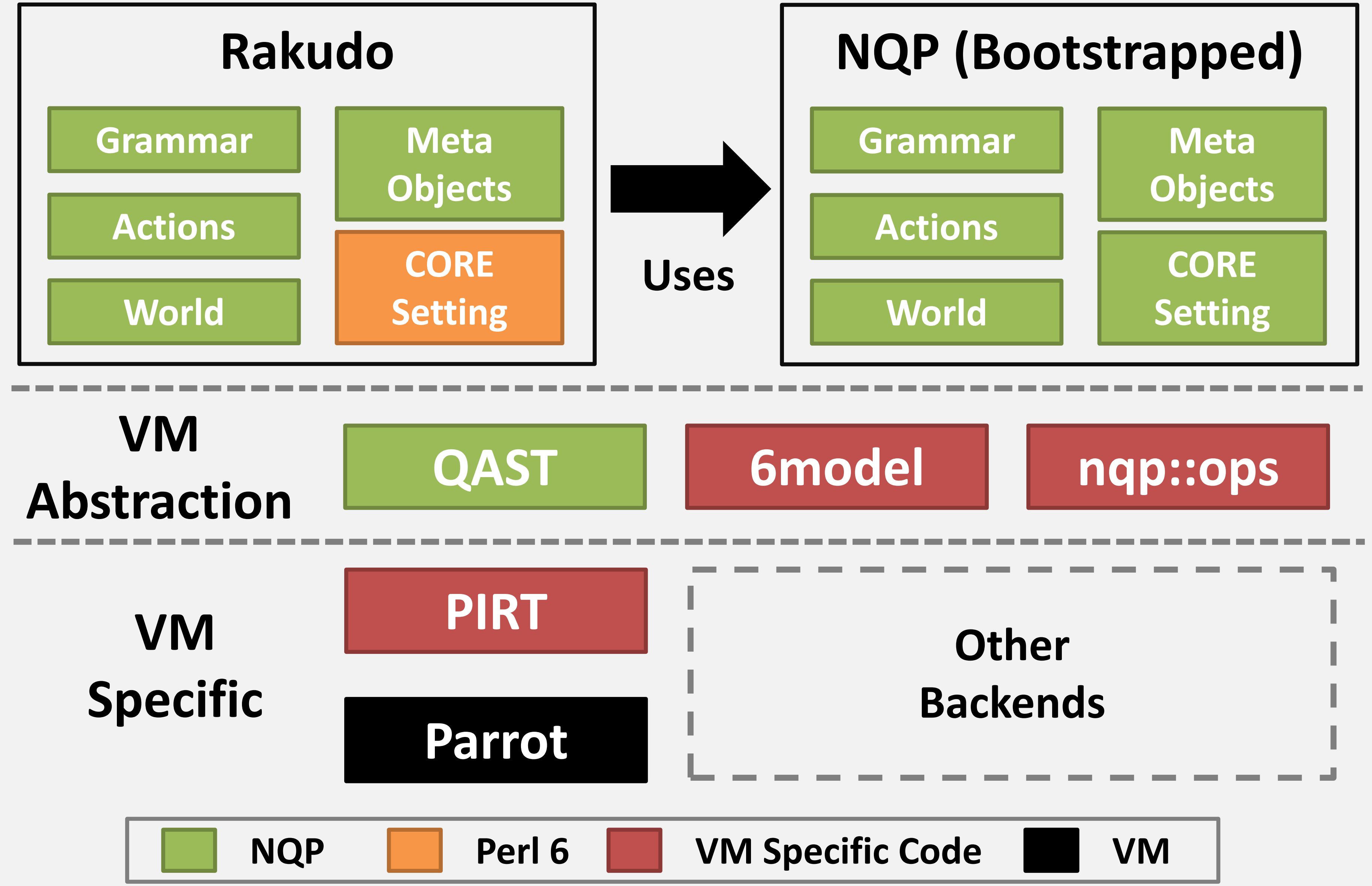
QAST

6model

nqp::ops

🟩 NQP 🟧 Perl 6 🟥 VM Specific Code ⬛ VM

Overall architecture



Rakudo

Grammar

Actions

World

Meta Objects

CORE Setting

Uses

NQP (Bootstrapped)

Grammar

Actions

World

Meta Objects

CORE Setting

VM Abstraction

QAST

6model

nqp::ops

VM Specific

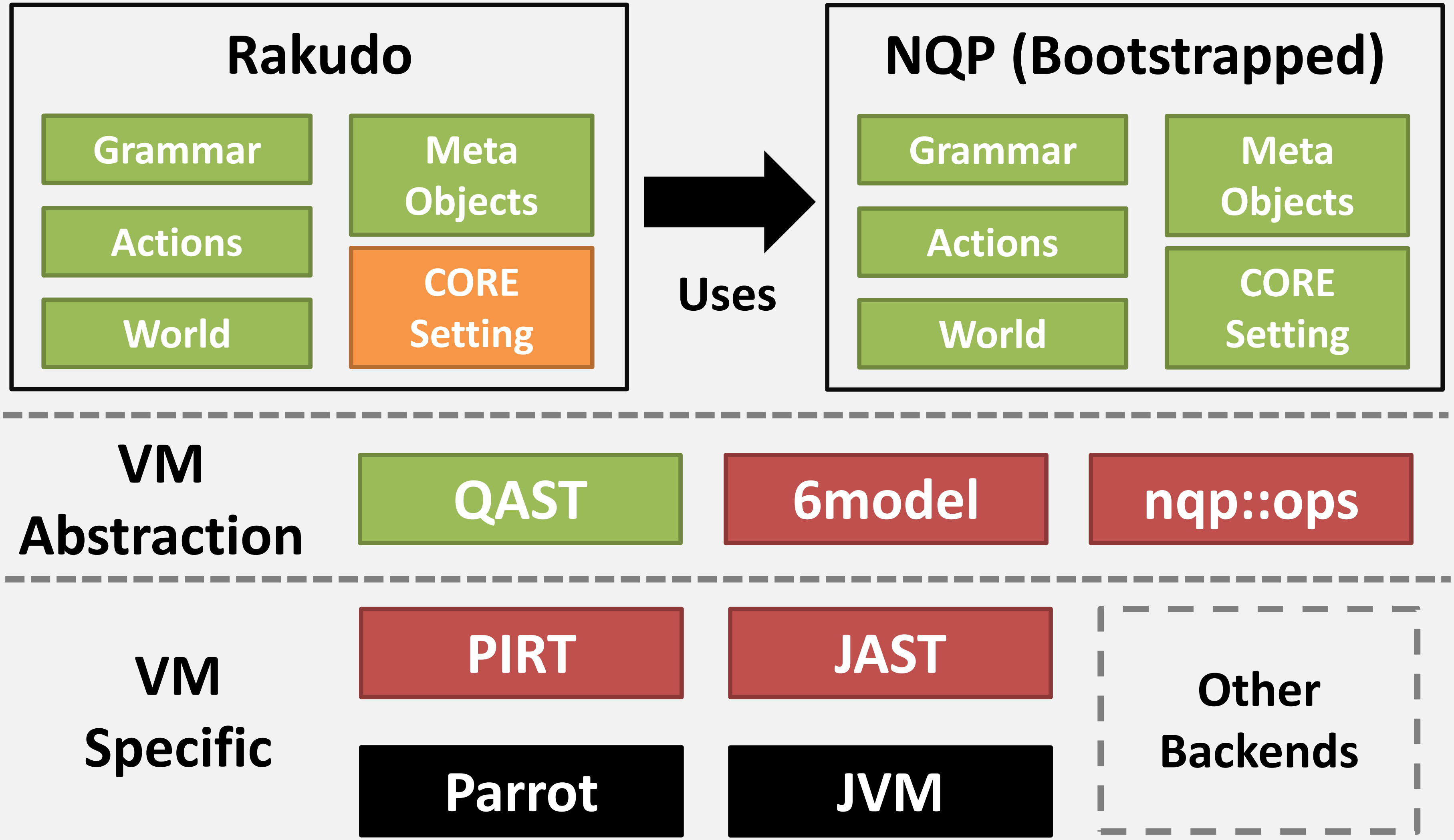
PIRT

Parrot

Other Backends

🟩 NQP 🟧 Perl 6 🟥 VM Specific Code ⬛ VM

Overall architecture: JVM plan



Rakudo

Grammar

Actions

World

Meta Objects

CORE Setting

Uses

NQP (Bootstrapped)

Grammar

Actions

World

Meta Objects

CORE Setting

VM Abstraction

QAST

6model

nqp::ops

VM Specific

PIRT

JAST

Parrot

JVM

Other Backends

🟩 NQP 🟧 Perl 6 🟥 VM Specific Code ⬛ VM

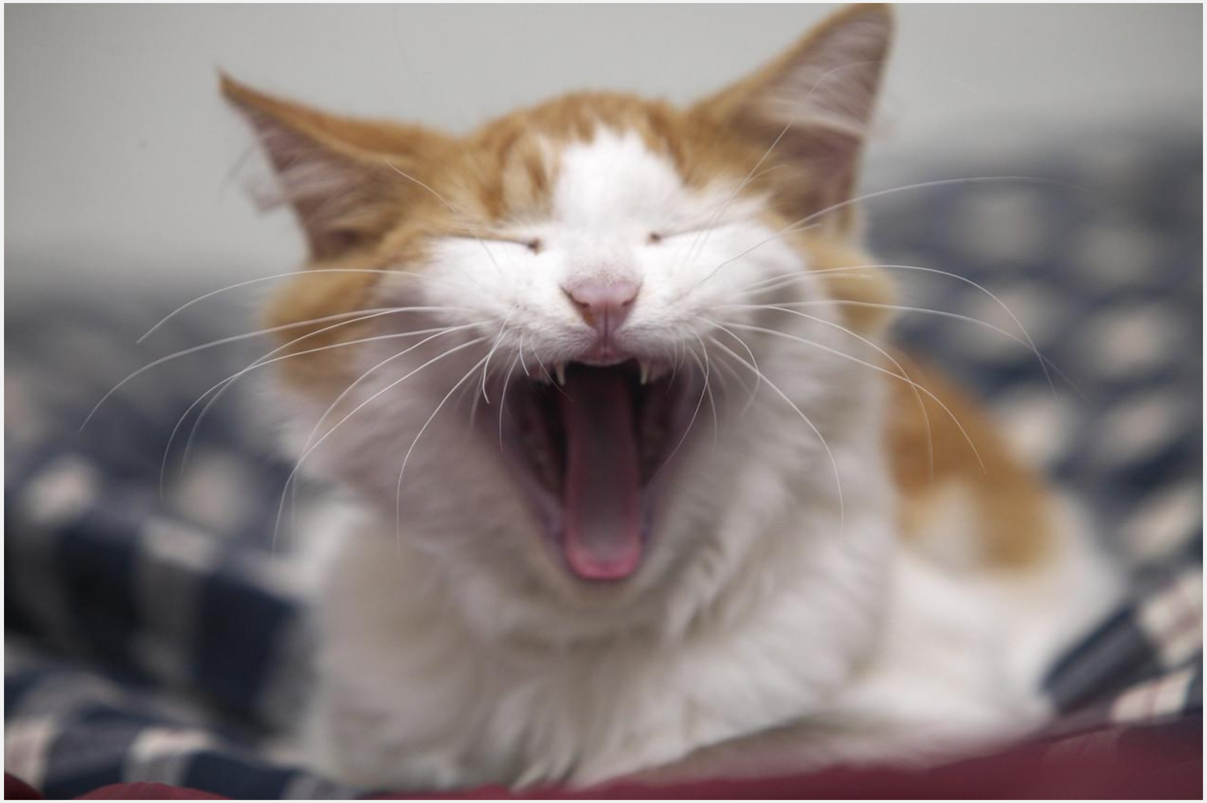
Step 1: JAST ➔ JVM bytecode

JVM Abstract Syntax Tree: a bunch of classes in NQP that can be used to describe Java bytecode

Steadily built up it up, test by test

```
jast_test(
    -> $c {
        my $m := JAST::Method.new(:name('one'), :returns('I'));
        $m.append(JAST::Instruction.new( :op('iconst_1') ));
        $m.append(JAST::Instruction.new( :op('ireturn') ));
        $c.add_method($m);
    },
    'System.out.println(new Integer(JASTTest.one()).toString());',
    "1\n",
    "Simple method returning a constant");
```

Bytecode generation? Boring!



Really, really did not want to have to do the actual class file writing. Thankfully, could re-use an existing library here (first BCEL, later ASM).

Step 2: basic QAST ➔ JAST

Now there was a way to produce Java bytecode from an NQP program, it was possible start writing a QAST to JAST translator

This also involved building out runtime support – including a JVM implementation of 6model

Also approached in a test driven way

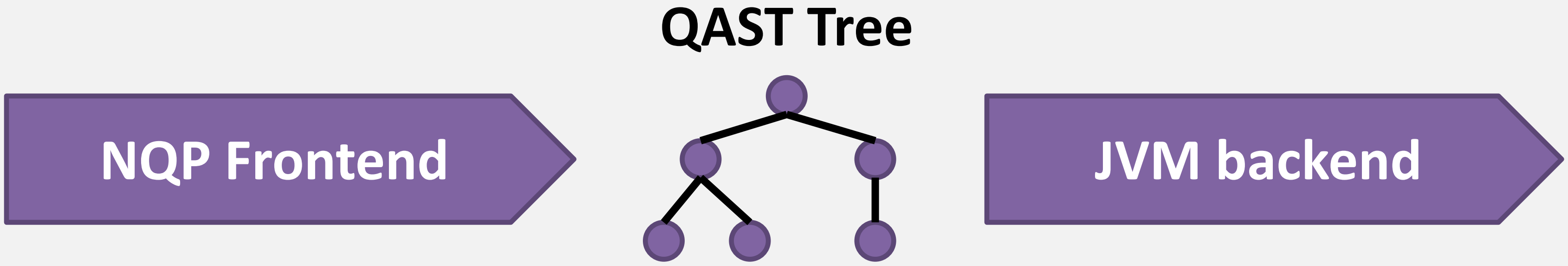
Test suite useful for future porting efforts

Step 2: basic QAST ➔ JAST

```
qast_test(
    -> {
        my $block := QAST::Block.new(
            QAST::Op.new(
                :op('say'),
                QAST::SVal.new( :value('QAST compiled to JVM!') )
            ));
        QAST::CompUnit.new(
            $block,
            :main(QAST::Op.new(
                :op('call'),
                QAST::BVal.new( :value($block) )
            )))
    },
    "QAST compiled to JVM!\n",
    "Basic block call and say of a string literal");
```

Step 3: NQP cross-compiler

Took existing grammar/actions/world from NQP on Parrot, and plugged in the JVM backend

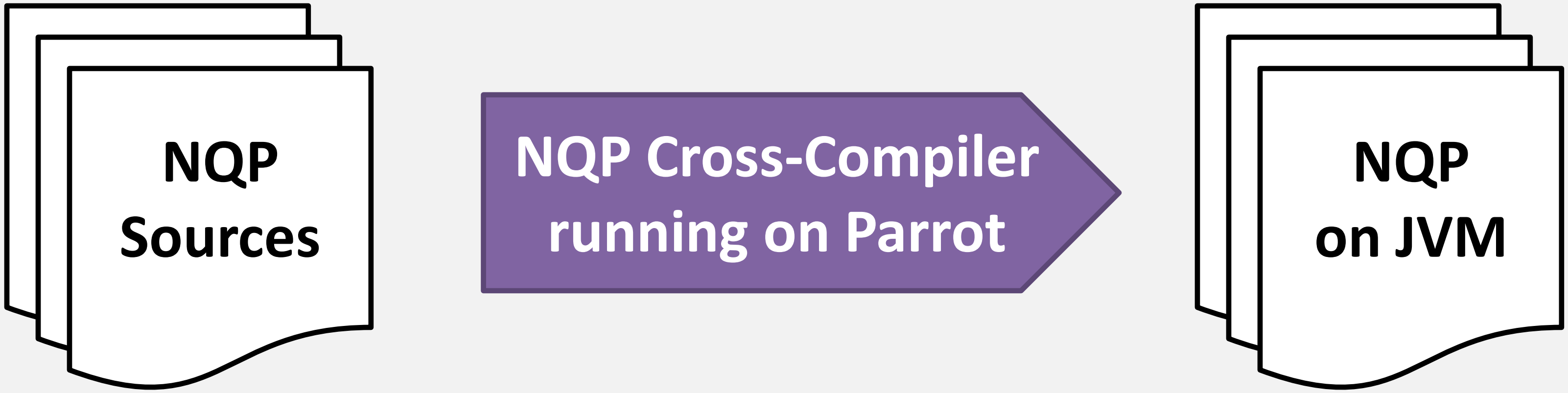


Took about 20 lines of code.

Design win!

Step 4: cross-compile NQP

Use the NQP cross-compiler to cross-compile NQP



NQP Sources

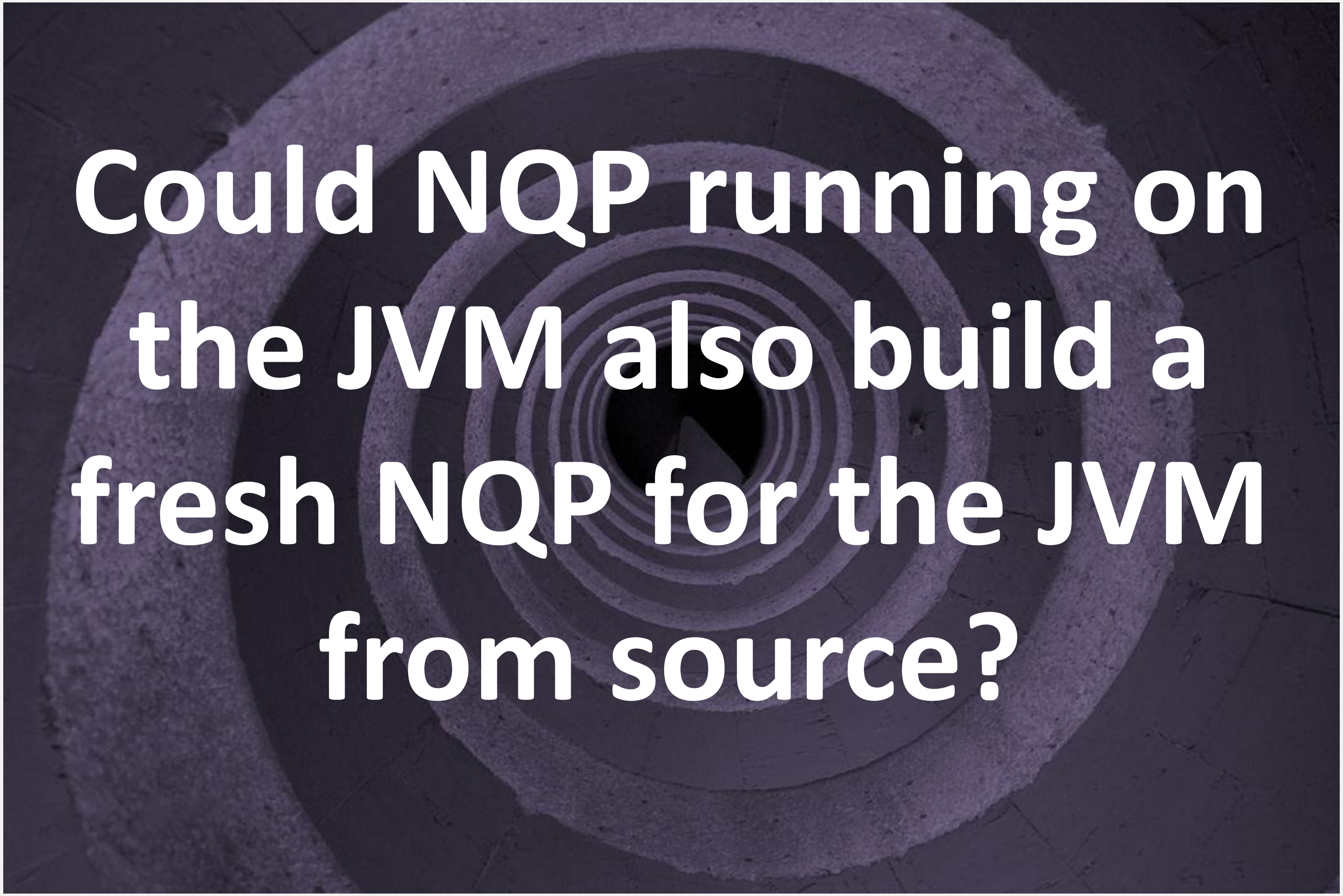
NQP Cross-Compiler running on Parrot

NQP on JVM

Hit various missing pieces, and some things that needed further abstraction

End result: a bunch of class files representing a standalone NQP on the JVM!

Step 5: close the bootstrap loop



Could NQP running on the JVM also build a fresh NQP for the JVM from source?

NQP on JVM

Answer: yes, once some missing pieces were completed (such as serialization)

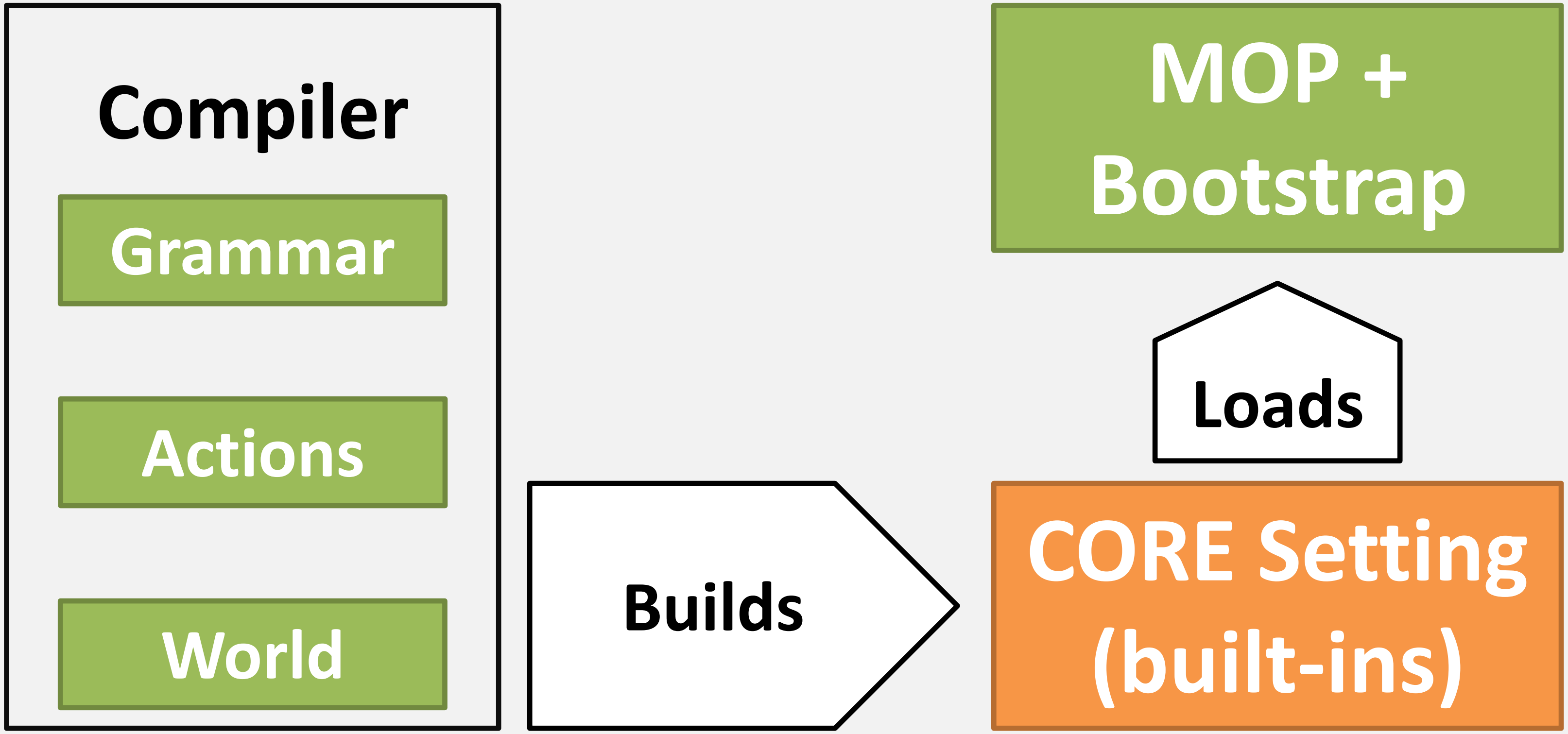
\☺/

Merged into NQP master in late April

Included in the May release of NQP

Rakudo: first port the compiler

Rakudo is broken into the compiler itself and various built-ins, including meta-objects. The compiler is used to build some of those built-ins.



Compiler, MOP and bootstrap

While the Perl 6 grammar and actions are much larger and more complex than their NQP equivalents, they don't really use anything new

Similar story for the various meta-objects

The bootstrap was a different story. It contains a huge BEGIN block that does a lot of setup work, piecing together the core Perl 6 types. This gets done at compile time, and is then serialized.

The setting: bit by bit, or all in one?

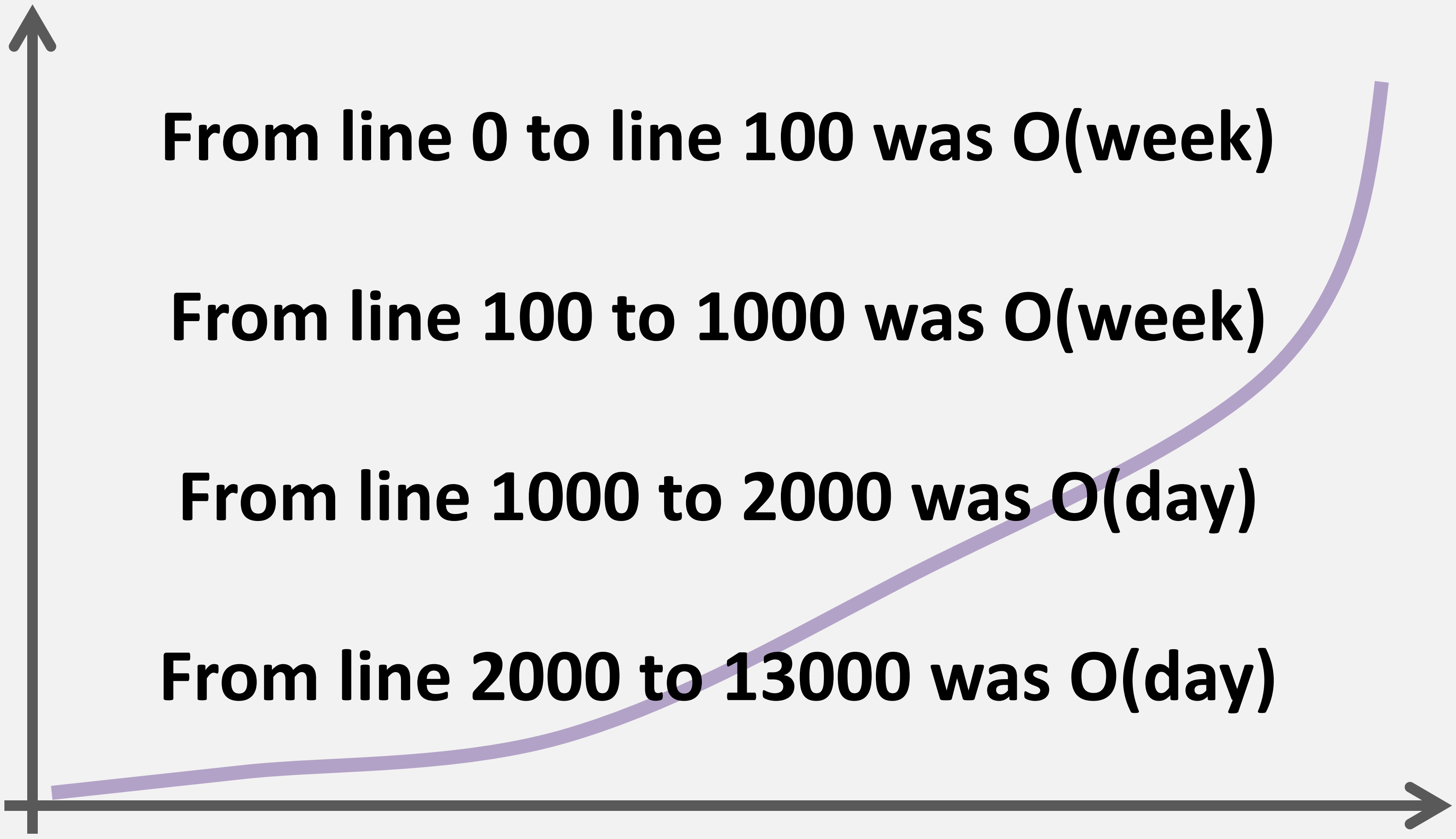
The CORE setting contains the built-in types and functions. It forms the outer scope of your program.

13,250

lines of Perl 6

That's a tough first test. ☺

Screw it, let's do it all anyway...



From line 0 to line 100 was O(week)

From line 100 to line 1000 was O(week)

From line 1000 to line 2000 was O(day)

From line 2000 to line 13000 was O(day)

What makes it hard?

Compiling the setting isn't just compiling

On line 137:

```
BEGIN &trait_mod:<is>.set_onlystar();
```

Yup, compiling the Perl 6 setting means running bits of Perl 6 code

Also traits, constants…

"Hello, JVM"

Finally...

```
$ perl6 -e "say 'Hello, JVM'"
Hello, JVM
```

Remember, this is running the compiler itself and loading just about all the core setting on the JVM; no Parrot required anywhere in the build!

Just "hello world", but not cheating at all ☺

(Well, apart from where we were...)

The specification test suite

The written Perl 6 specification is also expressed as a test suite (the "spectests")

Automated daily runs (thanks to Coke++)

So far, Rakudo on JVM is passing

99.28%

of the spectests that Rakudo on Parrot does

Java interoperability

So, now we can run much of Perl 6 on the JVM, but can we call into Java libraries?

?

Java interoperability

So, now we can run much of Perl 6 on the JVM, but can we call into Java libraries?

```
use java::util::zip::CRC32:from<java>;

my $crc = CRC32.new();

for 'Hello, Java'.encode('utf-8') {
    $crc.'method/update/(B)V'($_);
}

say $crc.getValue();
```

Java interop: SWT example (1)

The Standard Widget Toolkit is the library used by the Eclipse IDE to build its user interface

Not in the standard class library, so need to explicitly name the JAR file to loads the various classes we'll use from

```
constant SWTJAR = 'org.eclipse.swt.win32.jar';
use org::eclipse::swt::SWT:from<java>:jar(SWTJAR);
use org::eclipse::swt::widgets::Display:from<java>:jar(SWTJAR);
use org::eclipse::swt::widgets::Shell:from<java>:jar(SWTJAR);
use org::eclipse::swt::widgets::Text:from<java>:jar(SWTJAR);
```

Java interop: SWT example (2)

Use those types to display a window

```
my $display = Display.'constructor/new/()V'();
my $shell =
    Shell.'constructor/new/(Lorg/eclipse/swt/widgets/Display;)V'(
        $display);

my $helloWorldTest = Text.new($shell, SWT.'field/get_NONE/I'());
$helloWorldTest.setText("Hello from Perl 6");
$helloWorldTest.'method/pack/()V'();

$shell.'method/pack/()V'();
$shell.open();
until $shell.isDisposed() {
       $display.sleep unless $display.readAndDispatch();
}
$display.dispose();
```

Java interop: status

The basic things work

Plumbing layer by sorear++ is pretty capable

Sugar layer to make it convenient still needs plenty of improvements

Also need to work on calling into Perl 6 code from Java (or other JVM language) code

Is it any faster?

Startup time is awful. Such is the JVM.

Once it gets going and the JIT kicks in, it typically beats Rakudo on Parrot. How much it wins by depends on the nature of the work.

To put this in context, remember that we've been working at performance on Parrot for years, and the (largely unoptimized) JVM backend started 10 months ago is often coming out ahead anyway!

A real world result

...the script executed correctly in 11 minutes under Rakudo-JVM. ... It also executed correctly in Rakudo-Parrot -- but in 7 hours, 52 minutes.

Let me emphasize that. For this real-world task of significant size, Rakudo-JVM was 40 times faster than Rakudo-Parrot.

The script is pretty basic core stuff, mostly file I/O, grammar parsing, and hashes. The improvement is much smaller on a small data-set -- on my small test file, Rakudo-JVM is not even twice as fast as Rakudo-Parrot. But throw a big task (well, this big task, anyway) at Rakudo, and Rakudo-JVM crushes Rakudo-Parrot.

colomon

Threading?

Oh, yes. ☺

Come to tomorrow's talk!

Main room, same time.

What next?

Chip away at the remaining < 1% of specification tests that pass with Rakudo on Parrot, but fail with Rakudo on the JVM
(Goal: late August)

Get the module ecosystem and module installer (Panda) working well on Rakudo JVM, then create a JVM-based Rakudo Star distribution release
(Goal: September/October)

JVM backend weaknesses

Startup time is currently awful. Perfect storm of JVM startup being relatively slow, and us doing too much work at startup, before JIT kicks in.

➔ can be improved somewhat, with effort

While the commitment to `invokedynamic` seems serious, in reality it's new. I've run into bugs.

➔will very likely improve, with time

And, of course, ecosystem stuff is to come

➔ "just" needs more work ☺

There's more than one way to run it

Running on multiple backends is very much in the TMTOWTDI spirit of Perl

Contrast with how other languages are doing it: Rakudo is targeting multiple backends with a single implementation, rather than one per VM

Vision

Rakudo Perl 6 runs well on a number of platforms, and is fast and reliable enough for most tasks

Modules, debugger, etc. work reliably on the different backends

Most development effort goes into the things that are shared, rather than they VM specific stuff

Perl 6 users build great stuff, and enjoy doing so

Thank you!

Questions?

Blog: 6guts.wordpress.com
Twitter: @jnthnwrthngtn
Email: jnthn@jnthn.net