

Rakudo Perl 6: to the JVM and beyond!

Jonathan Worthington

Rakudo

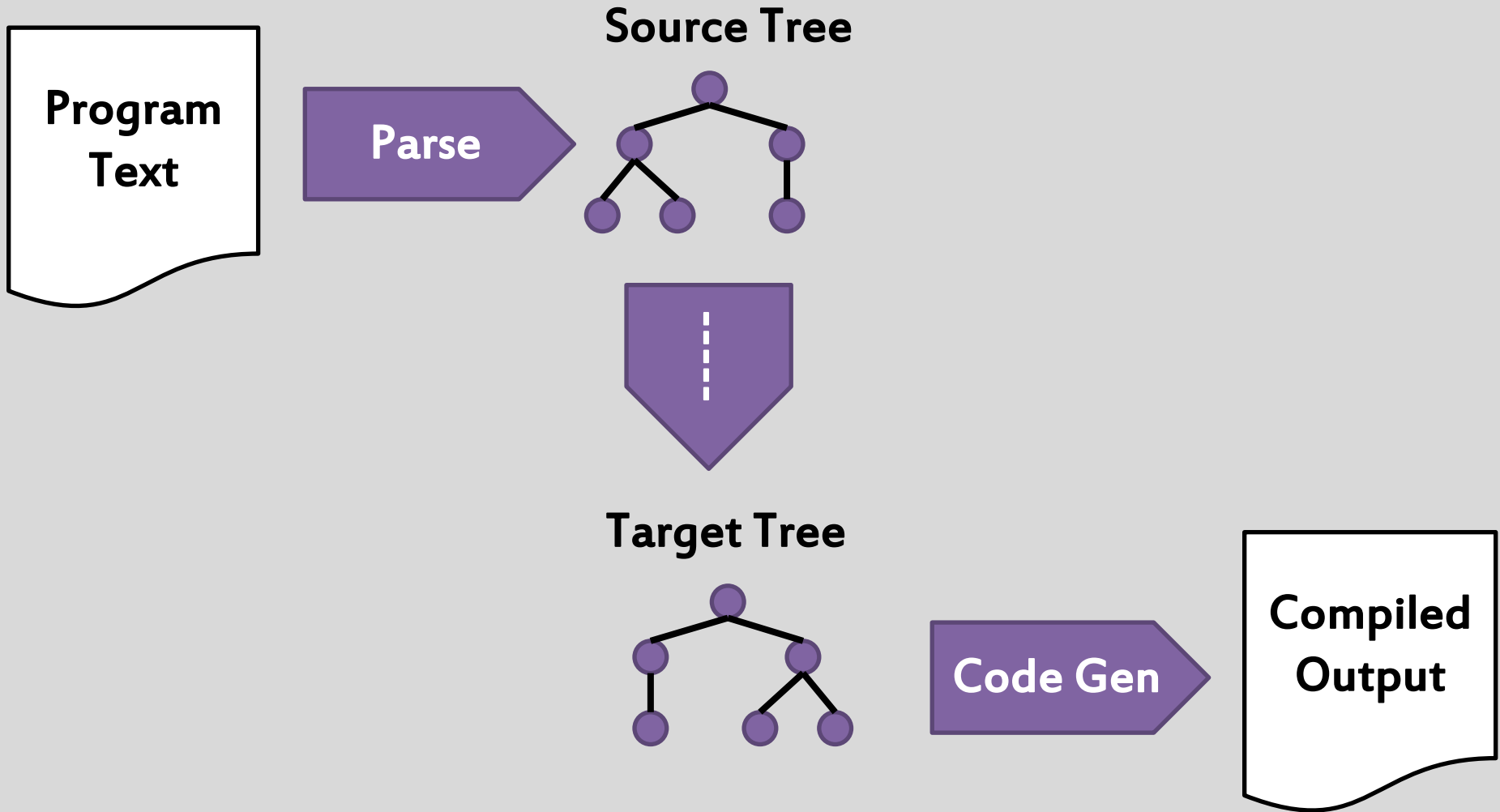
A Perl 6 implementation

Compiler + built-ins

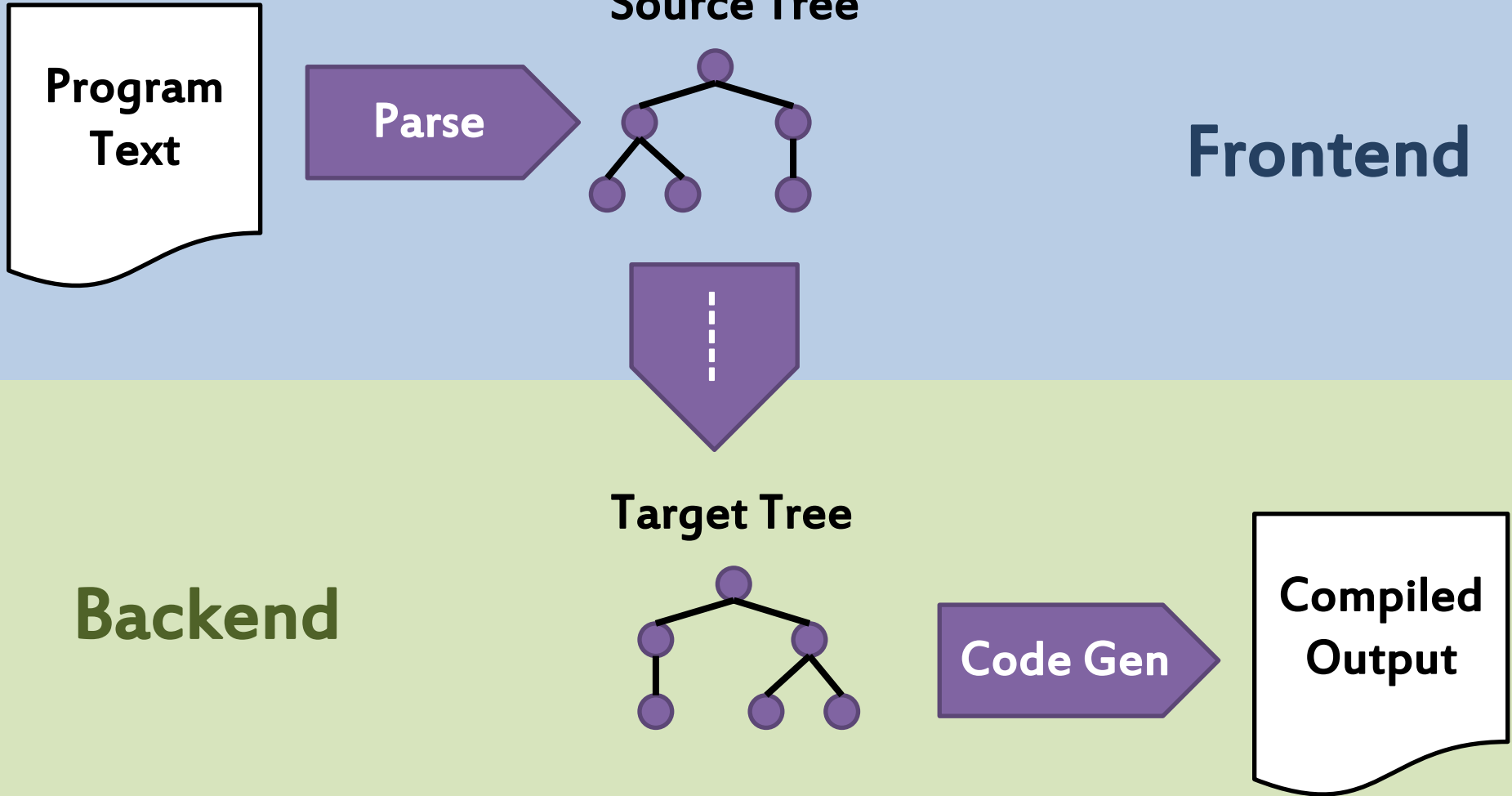
64 monthly releases to date

10-15 code contributors per release
(but we draw on many other contributions
too: bug reports, test suite work, etc.)

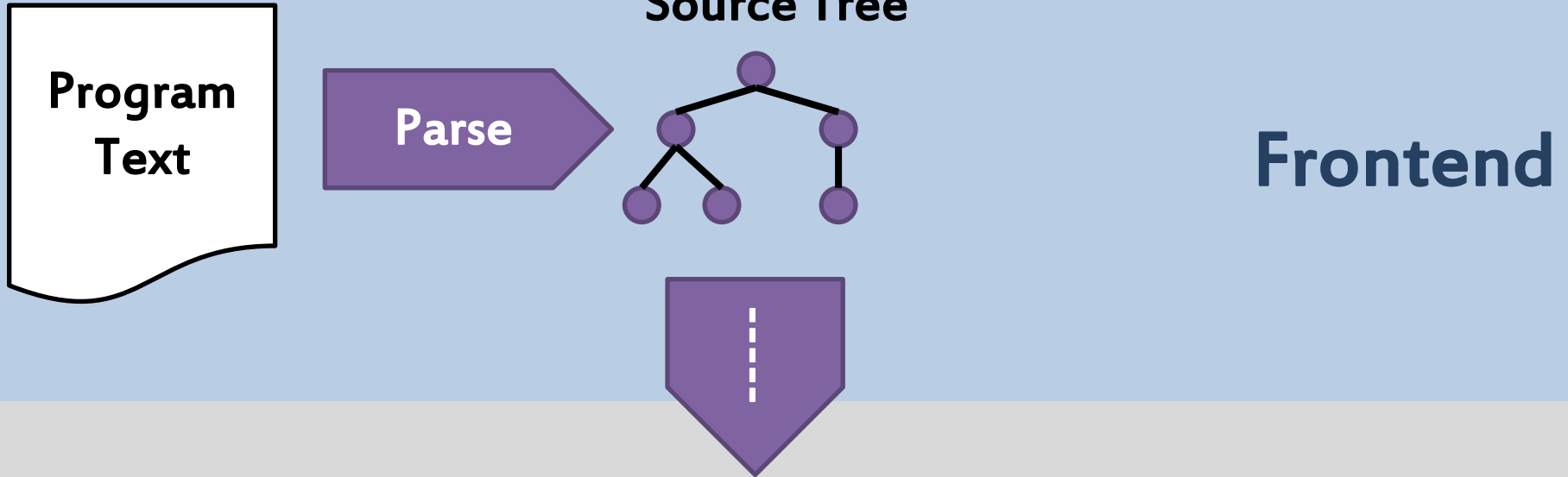
What a compiler does



What a compiler does



The frontend



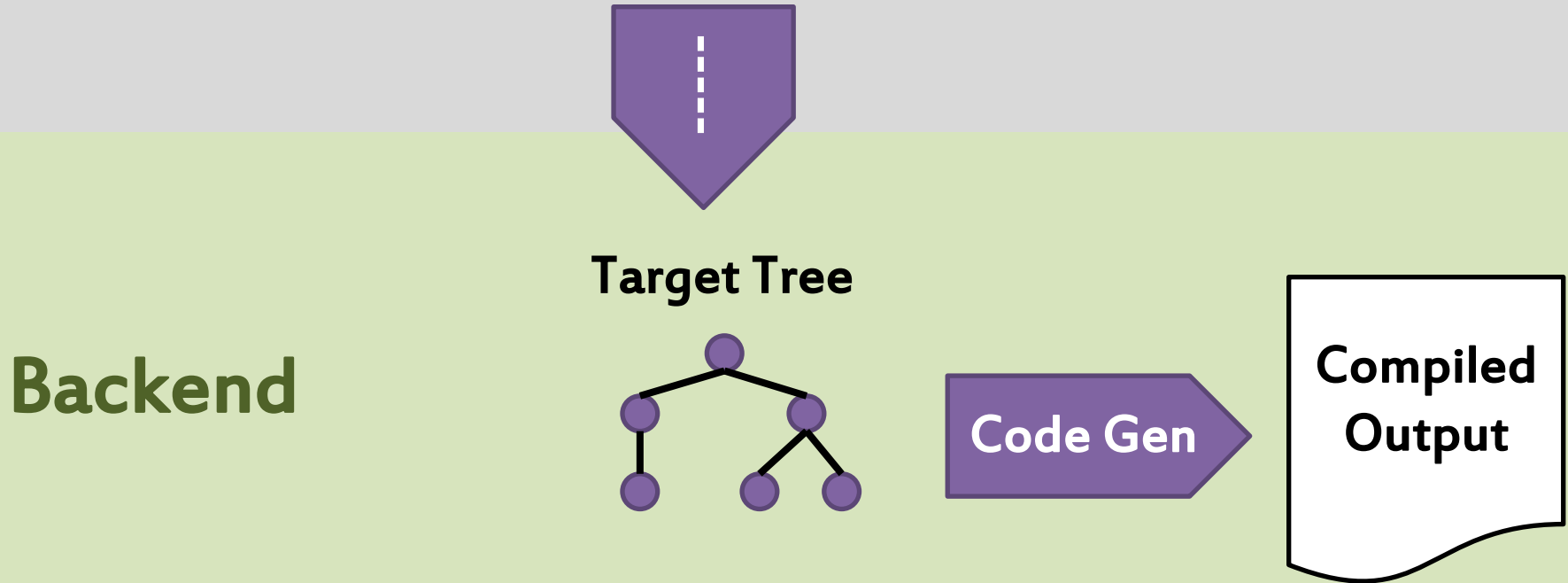
All about a specific language

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 some well-defined data structure as input

and

Produce some well-defined data structure as output

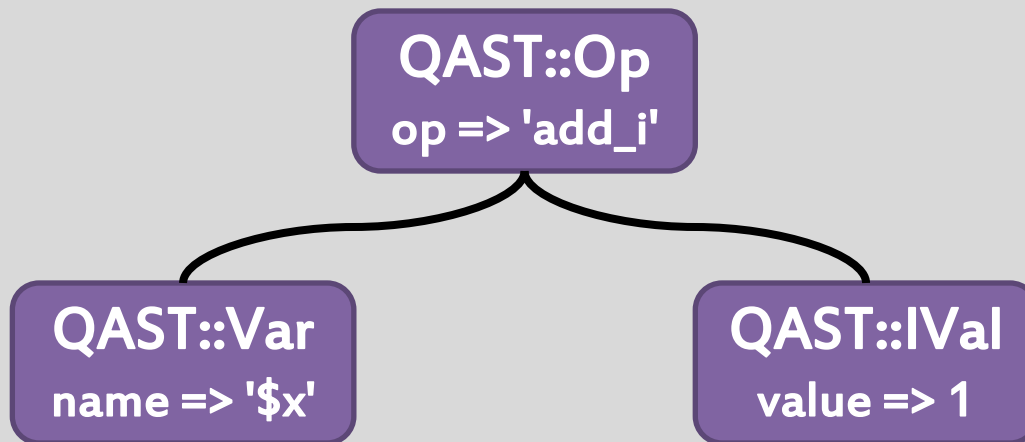
Each stage may be relatively complex. However, it is
also completely self-contained.

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



Building a Perl 6 compiler

The stuff you learn in compiler class on a typical computer science course only takes you so far

"Necessary, but not sufficient"

The overall assumption is source in, and – provided it is free of errors – translation to the target runtime

So what makes Perl 6 interesting?

Compile time at runtime

You might have to compile stuff while running

```
eval 'say "Here we go compiling..."';
```

```
my $pat = prompt 'Pattern: ';  
my $txt = prompt 'Text: ';  
say $txt ~~ /<$pat>/;
```

This one is fairly easy. You just make sure that the compiler is available at runtime.

Runtime at compile time

You might have to run stuff while compiling

```
CHECK say now - BEGIN now;
```

Who knows what will be called!

```
sub fac($n) { [*] 1..$n }  
constant fac20 = fac(20);
```

**It's not just that we have to run stuff while compiling.
It's that we have to run parts of the program we're in
the middle of compiling.**

Compiler must be re-entrant

After all, there's nothing to stop a bit of runtime at compile time doing a bit of compile time too...

```
BEGIN eval 'say q[oh, my]'
```

But really, that's what loading and compiling a module is.

In reality, just means not having global state. Which is good design anyway.

Mutable grammar

New operators and terms can be introduced during the parser, augmenting the parser

```
sub postfix:<!*>($n) { [*] 1..$n }  
say 20!
```

But only lexically!

```
{  
    sub postfix:<!*>($n) { [*] 1..$n }  
}  
say 20!; # Must be an error
```

Meta-programming

The meaning of the package declarators can be changed – also just for a lexical scope

```
use Grammar::Tracer;  
  
grammar JSON::Tiny {  
    # Rules in there should be traced  
}
```

In fact, most declarations boil down to creating objects rather than generating code

Serialization

Creating objects is fine, but what if we need to put the output in some kind of bytecode file?

The objects must be serialized! And they may reference objects from other compilation units.

```
BEGIN {  
    my $c = Metamodel::ClassHOW.new_type(  
        :name('Foo') );  
    EXPORT::DEFAULT::<Foo> = $c;  
}
```

Separate compilation, but...

So we serialize stuff per compilation unit. But wait, what about...

```
augment class Int {  
    method answer() { 42 }  
}
```

Need to detect such changes, and then re-serialize new versions of meta-objects from other compilation units!

Every operator is a multi-dispatch

Things like...

```
$a + $b
```

...compile down to...

```
&infix:<+>($a, $b)
```

...which is a multiple dispatch on type.

How can this ever be fast? Needs inlining →
compile time analysis of multiple dispatches.

Gradual typing, pluggable types

The Perl 6 type system is opt-in

```
my $name;  
my Int $age;
```

When you write types, the compiler should be able to make use of the information to do additional checks and generate better code

However, meta-programming means that the type checking method is user-overridable!

Rakudo: the early days

Built a relatively conventional compiler

**The grammar engine was decidedly innovative,
using a Perl 6 grammar to do the parsing**

**Turned class declarations into calls on meta-objects,
which we run during startup → costly, and not really
the right semantics either**

In fact, all BEGIN time was...icky. ☹

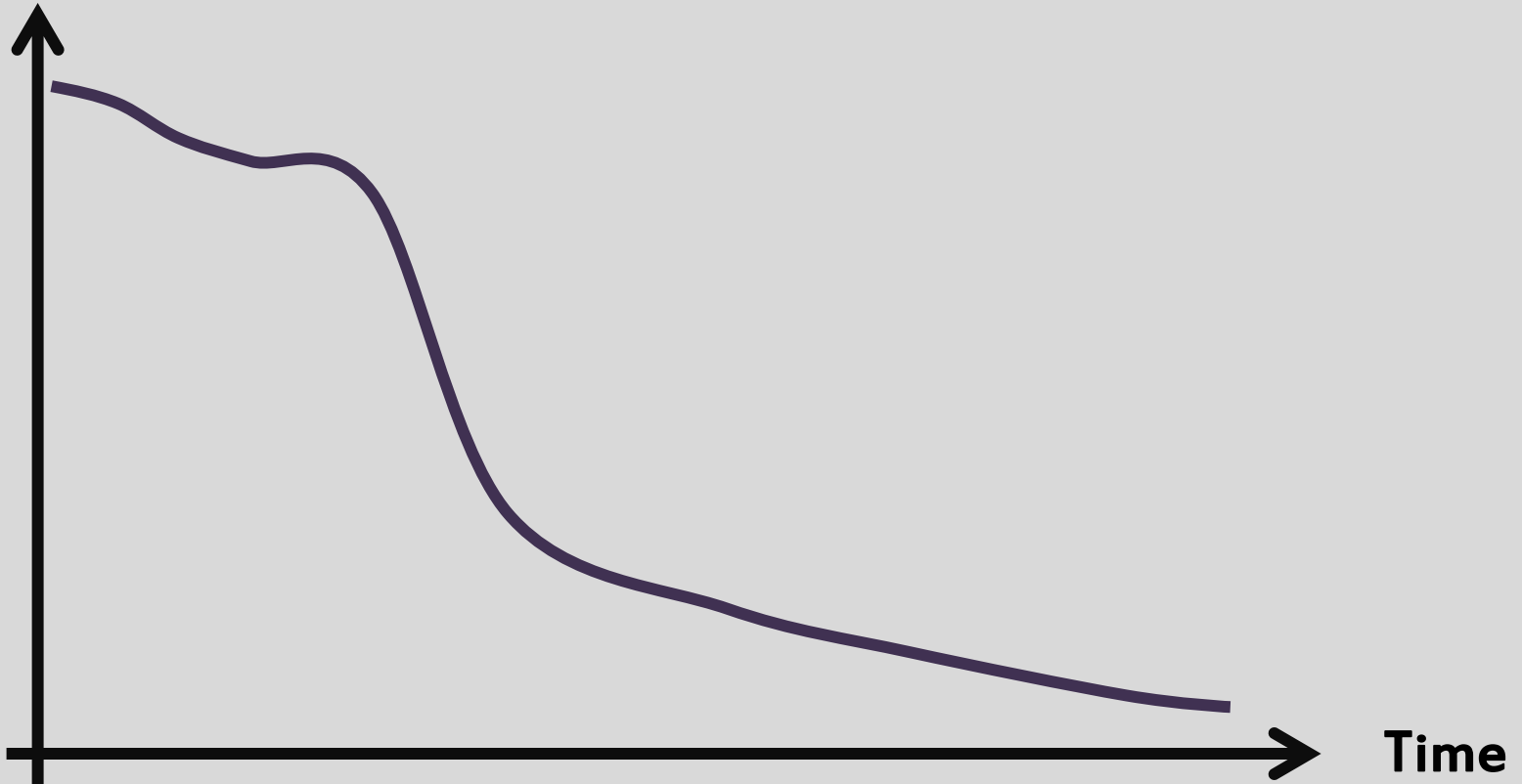
Result: impractical foundation

We made a lot of things work. On the surface, it looked promising. But inside, it felt like this:



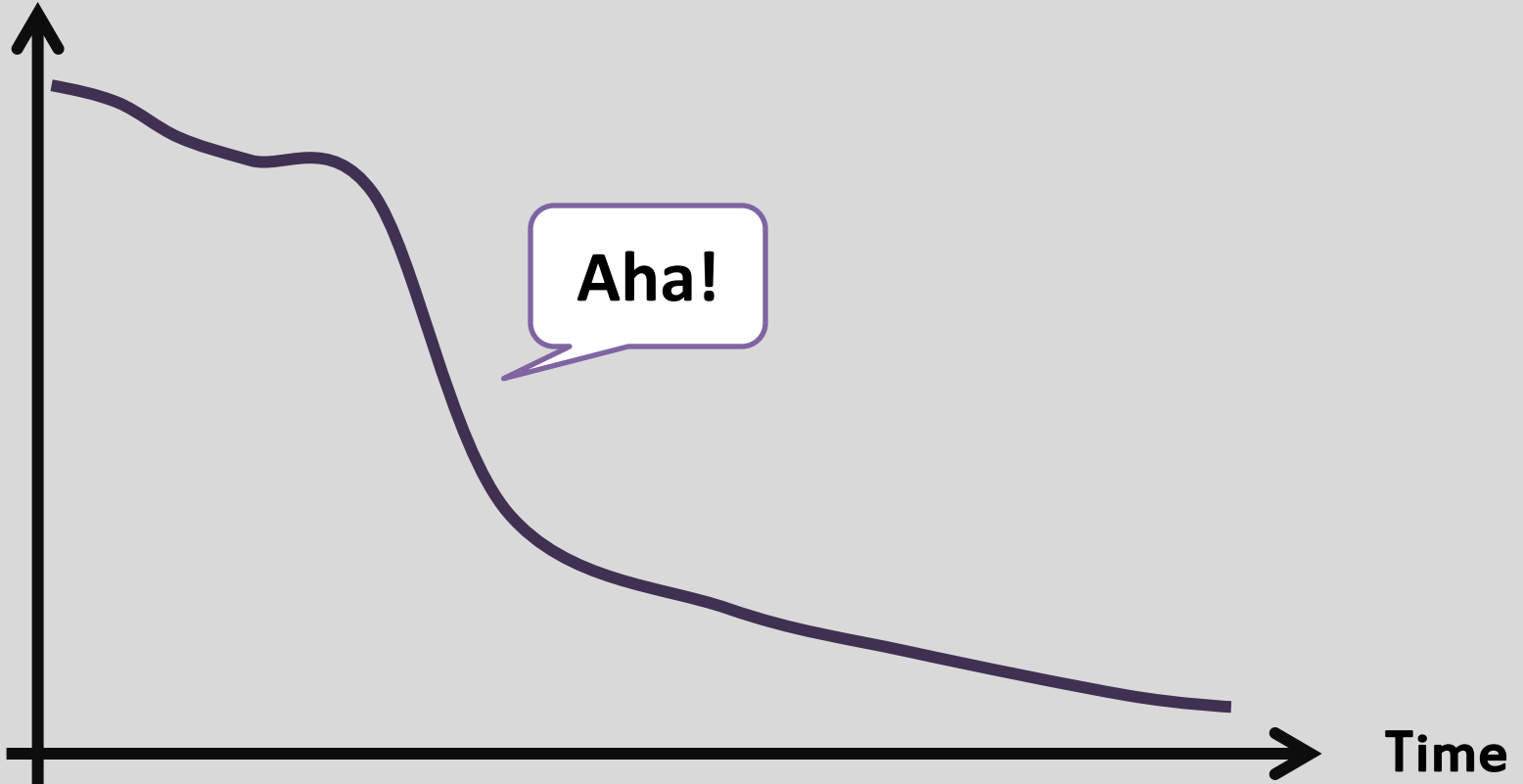
The ignorance curve

Ignorance

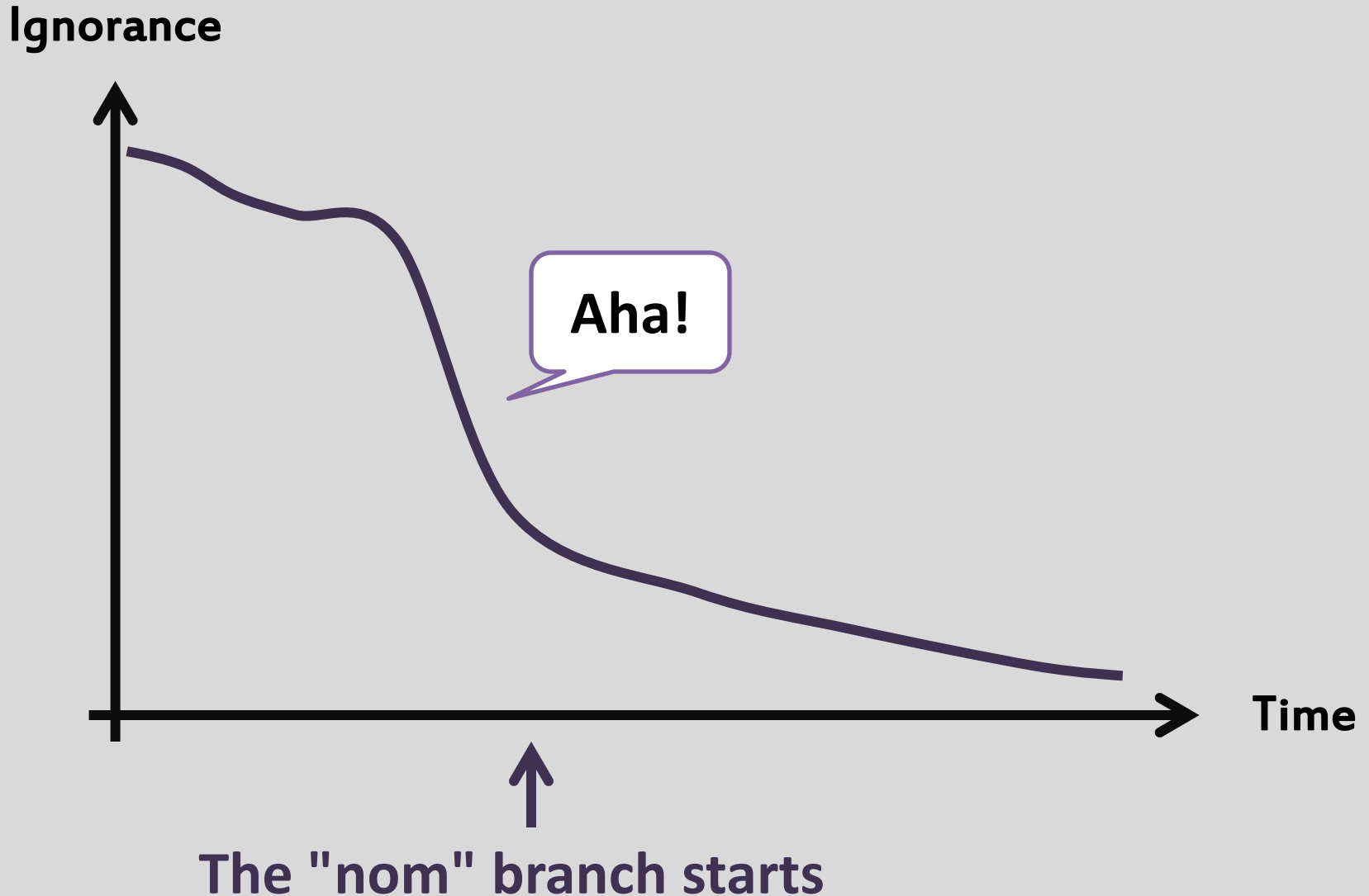


The ignorance curve

Ignorance



The ignorance curve



Realizations

The split of grammar (syntax) and actions (semantics) left the handling of declarations scattered all over the compiler

→ need a third thing (we called it World)

Creating objects during the parse/compilation and referring to them at runtime happens everywhere

→ must be cheap and easy

Must create meta-objects as we parse, and have robust ways to handle BEGIN time

Around the same time...

Ruby and Python are on multiple VMs. Why can't Perl do that also?

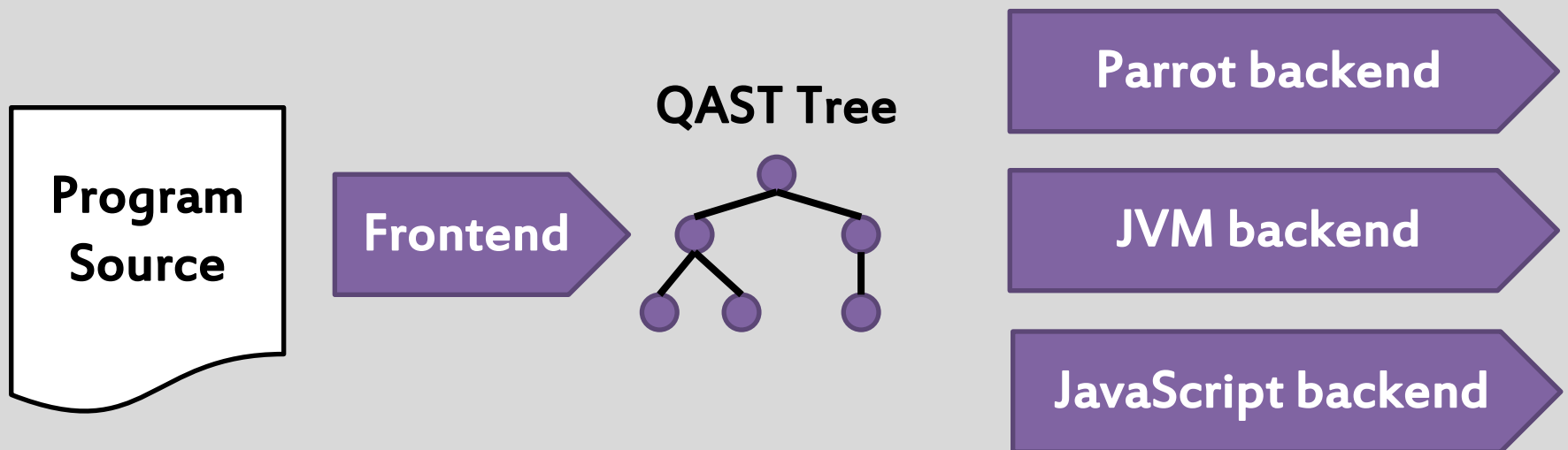
Perl 5 runs on lots of platforms in the CPU/OS sense. But these days, a lot of the interesting platforms are not physical machines. They're virtual machines.

Some organizations want to deploy everything on a particular virtual platform. A language doesn't run on that platform? Can't use it.

Additional realizations

Building the things needed to support a Perl 6 implementation is rather difficult

Thus, Rakudo should look at how this investment could be re-used when targeting new VMs



So, the grand plan

Extensive re-working of Rakudo and the compiler toolchain to better handle declarations, meta-objects, and have a much better foundation

Don't do all of the serialization and multi-VM stuff right away. Just make it possible in the future.

Took longer than imagined, for various reasons. In hindsight, it was still the right call. At the time, plenty of whining. Taking hard decisions is hard.

So, what do we write it in?

From very early on, various components of Perl 6 were written in **NQP**, a Perl 6 subset

Earlier work on Rakudo had extensive portions of the built-ins, OO stuff, etc. written in PIR, a thin layer on Parrot's assembly language.

Problem: to work on Rakudo required learning PIR. Which, frankly, was horrible. Even Parrot folks tend usually agree PIR is horrible.

Write all the things in NQP / Perl 6

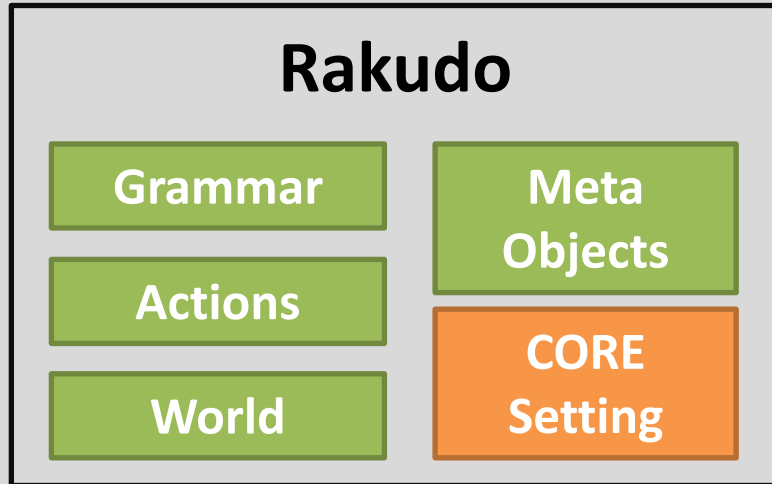
Thus, we gradually moved to writing almost everything in either NQP, or Perl 6 itself

More accessible to more contributors

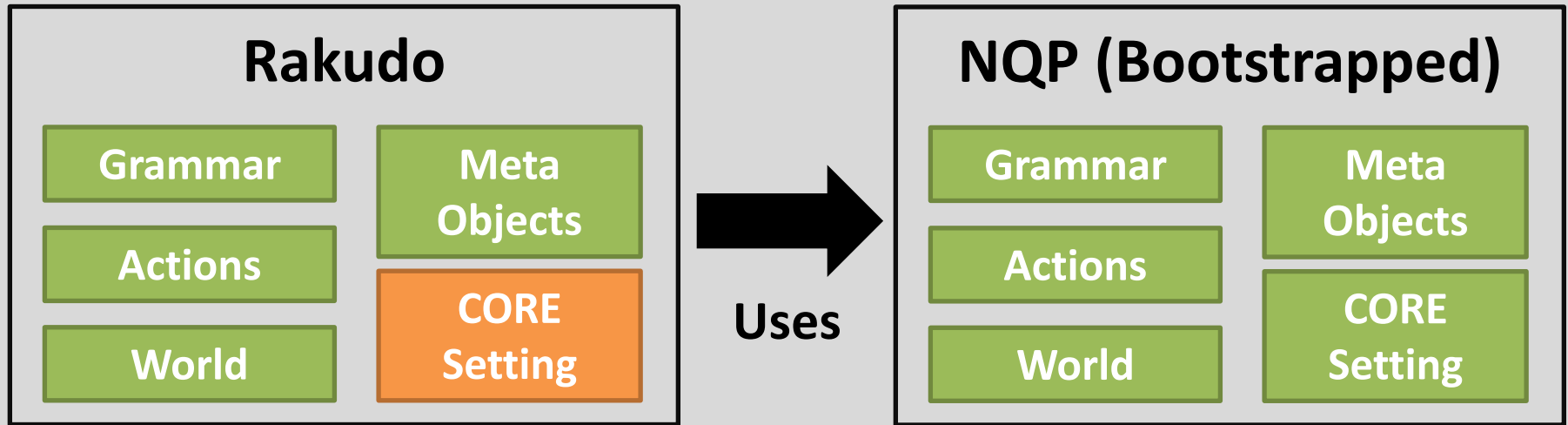
Even NQP came to be written entirely in NQP!

This would be important for porting...

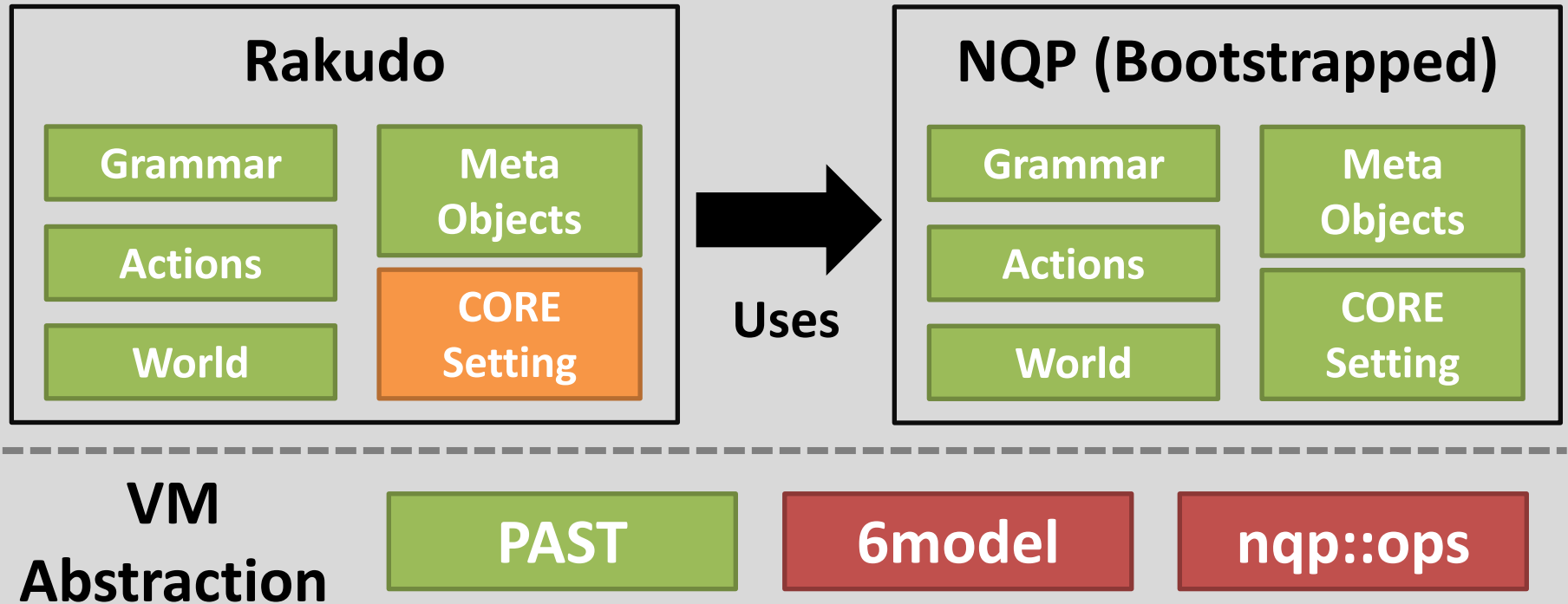
Overall architecture



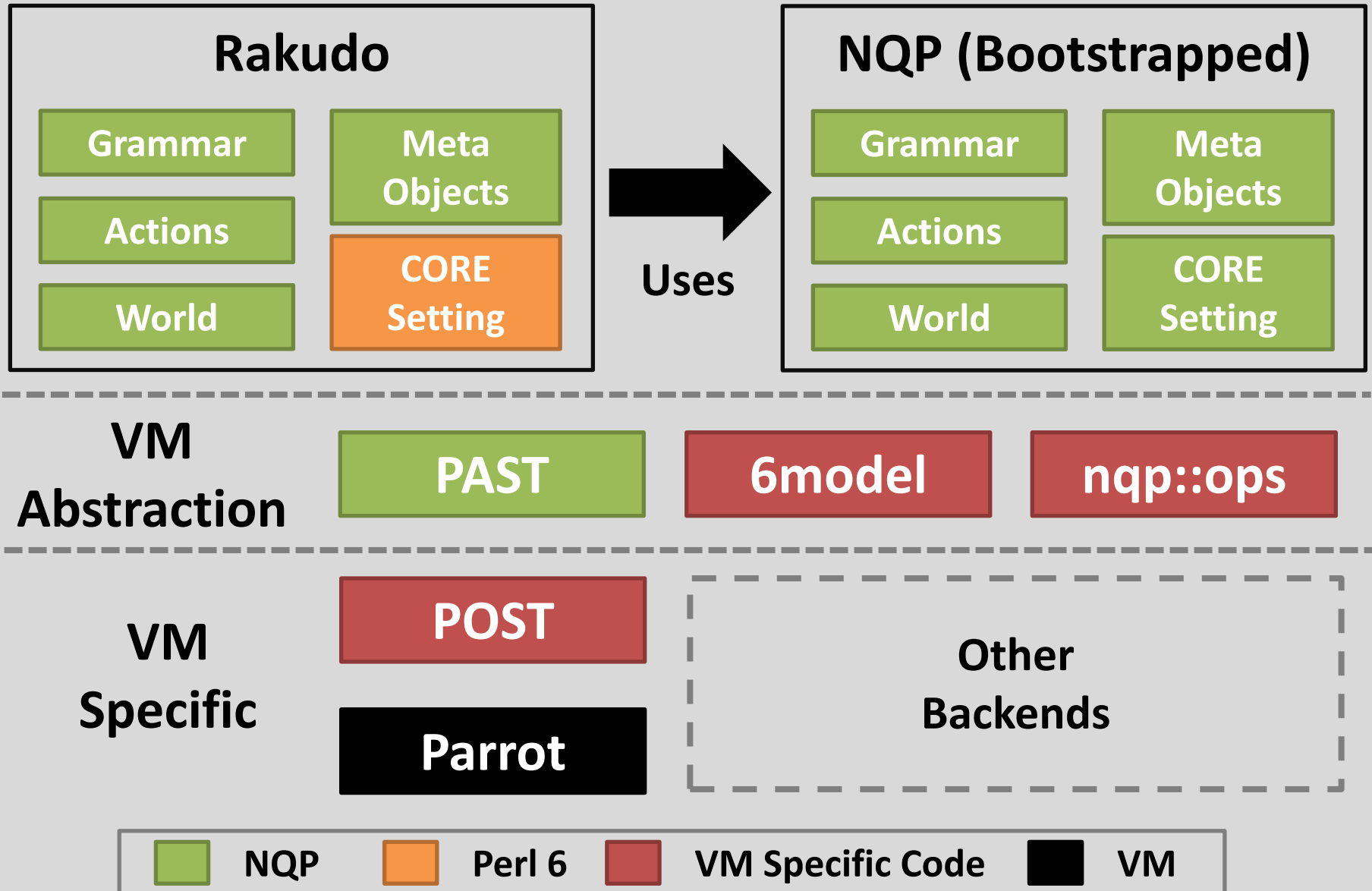
Overall architecture



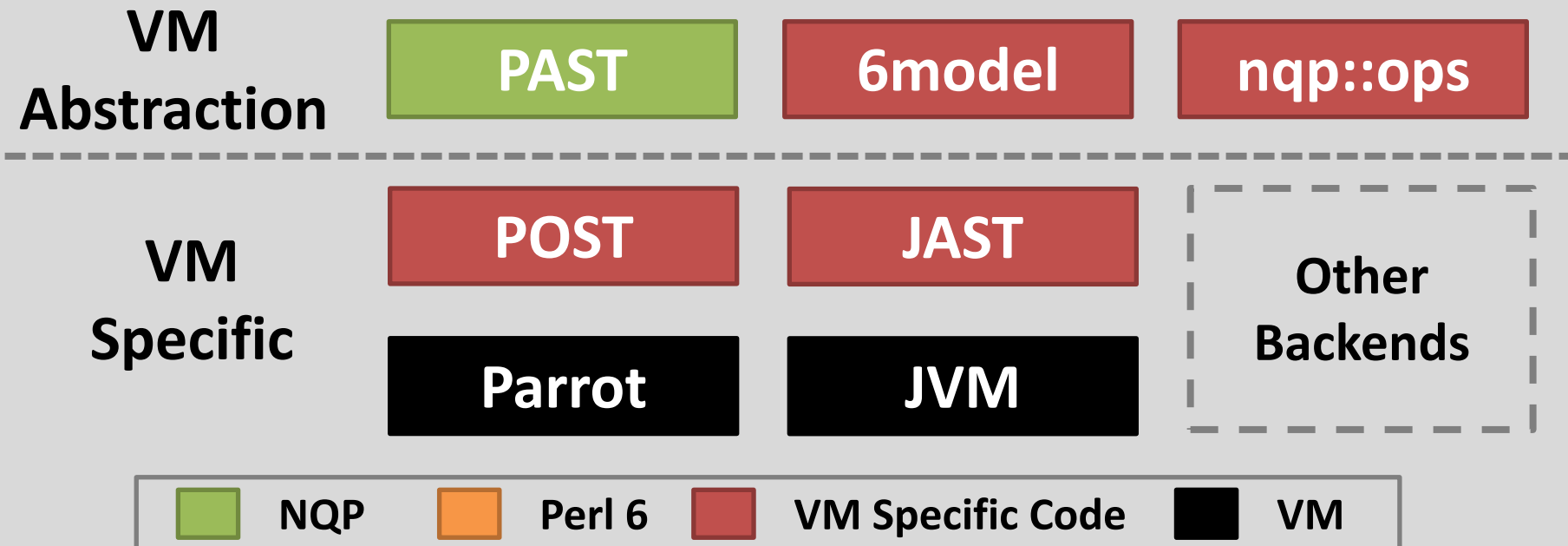
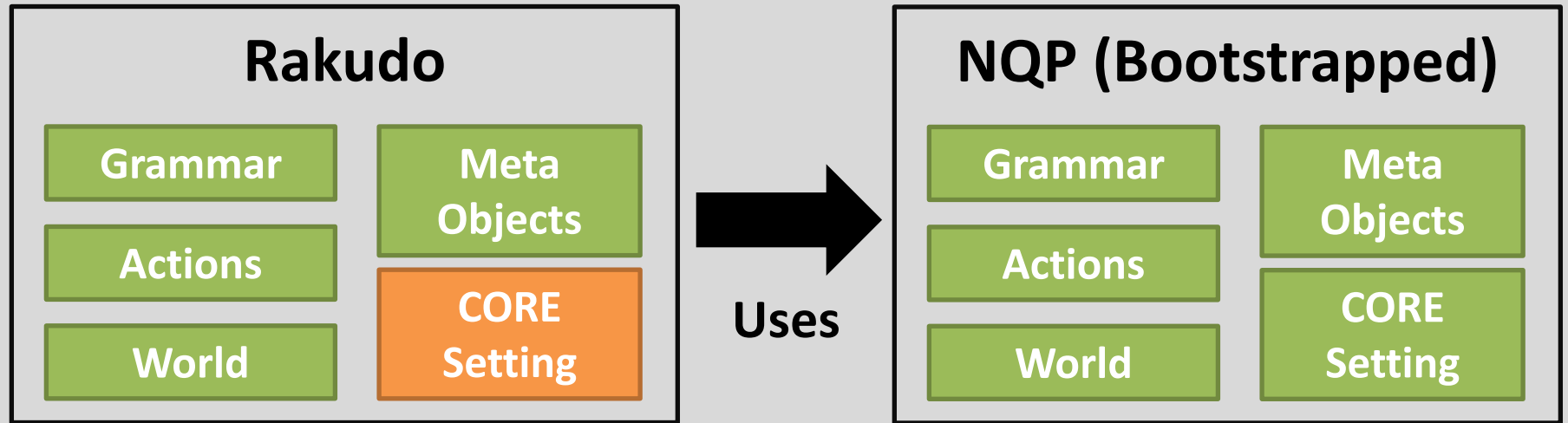
Overall architecture



Overall architecture



Overall architecture: JVM plan



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

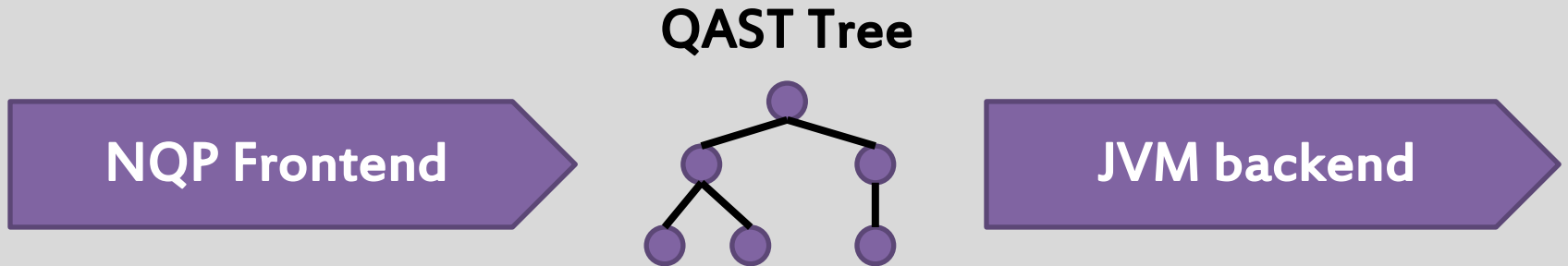
Test suite useful for future porting efforts

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

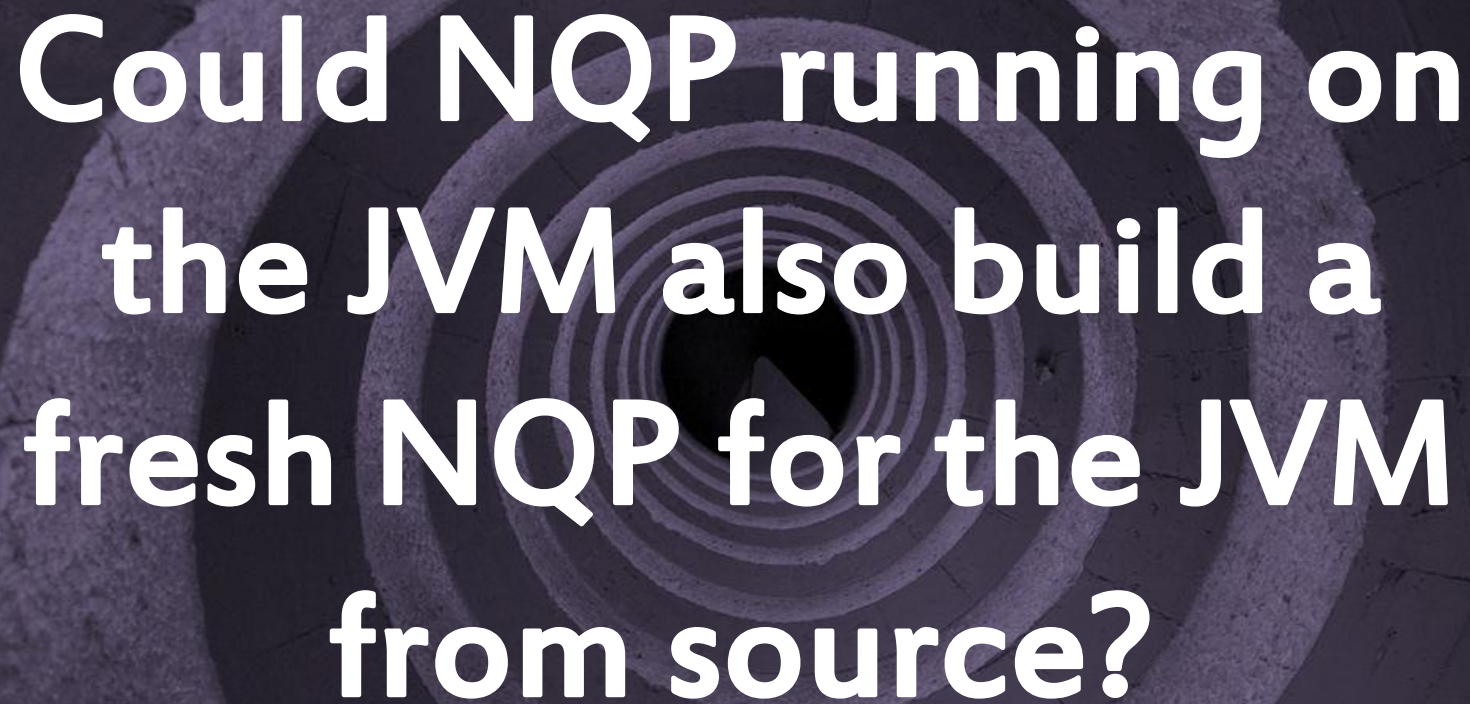
Use the NQP cross-compiler to cross-compile NQP



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)

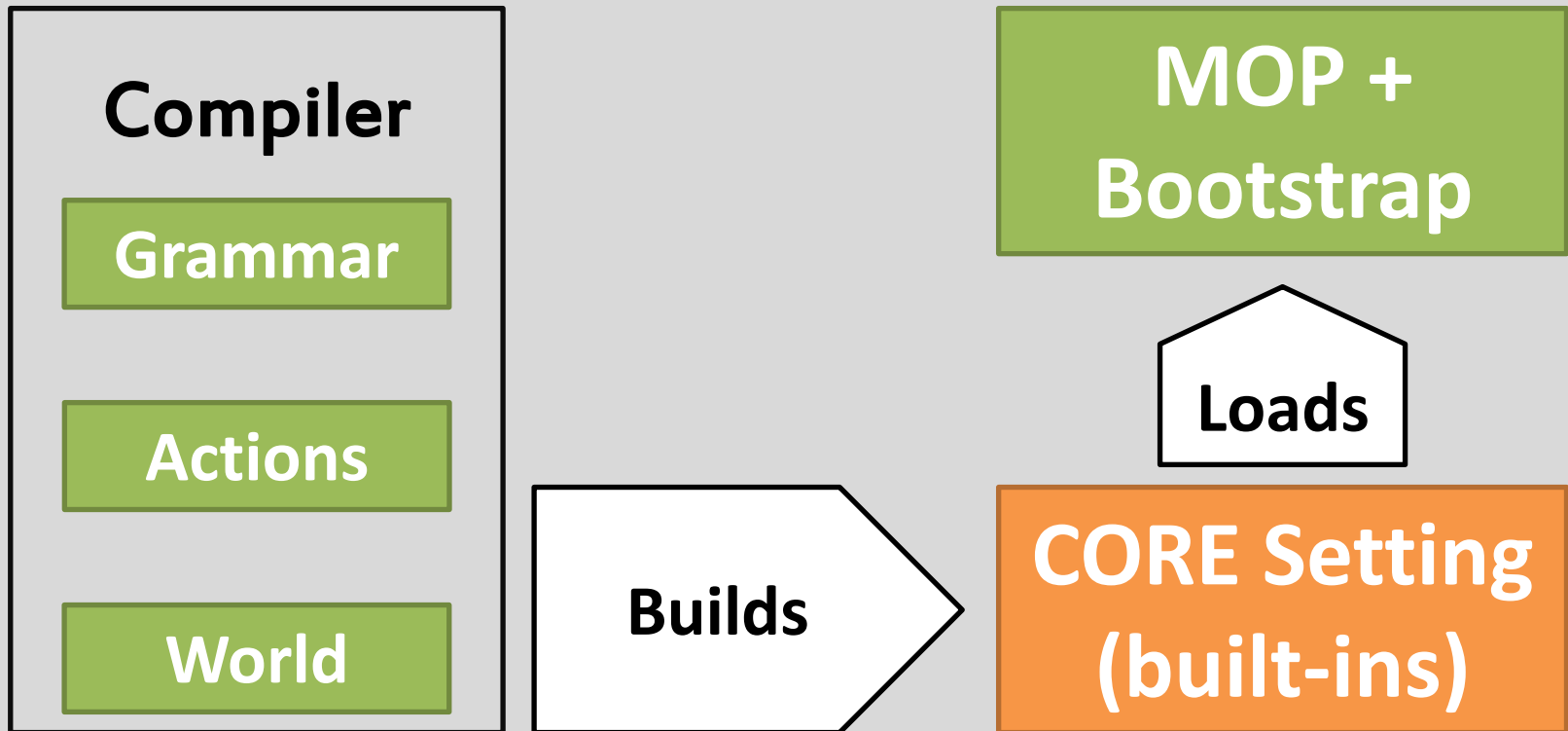


Merged into NQP master in late April

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?

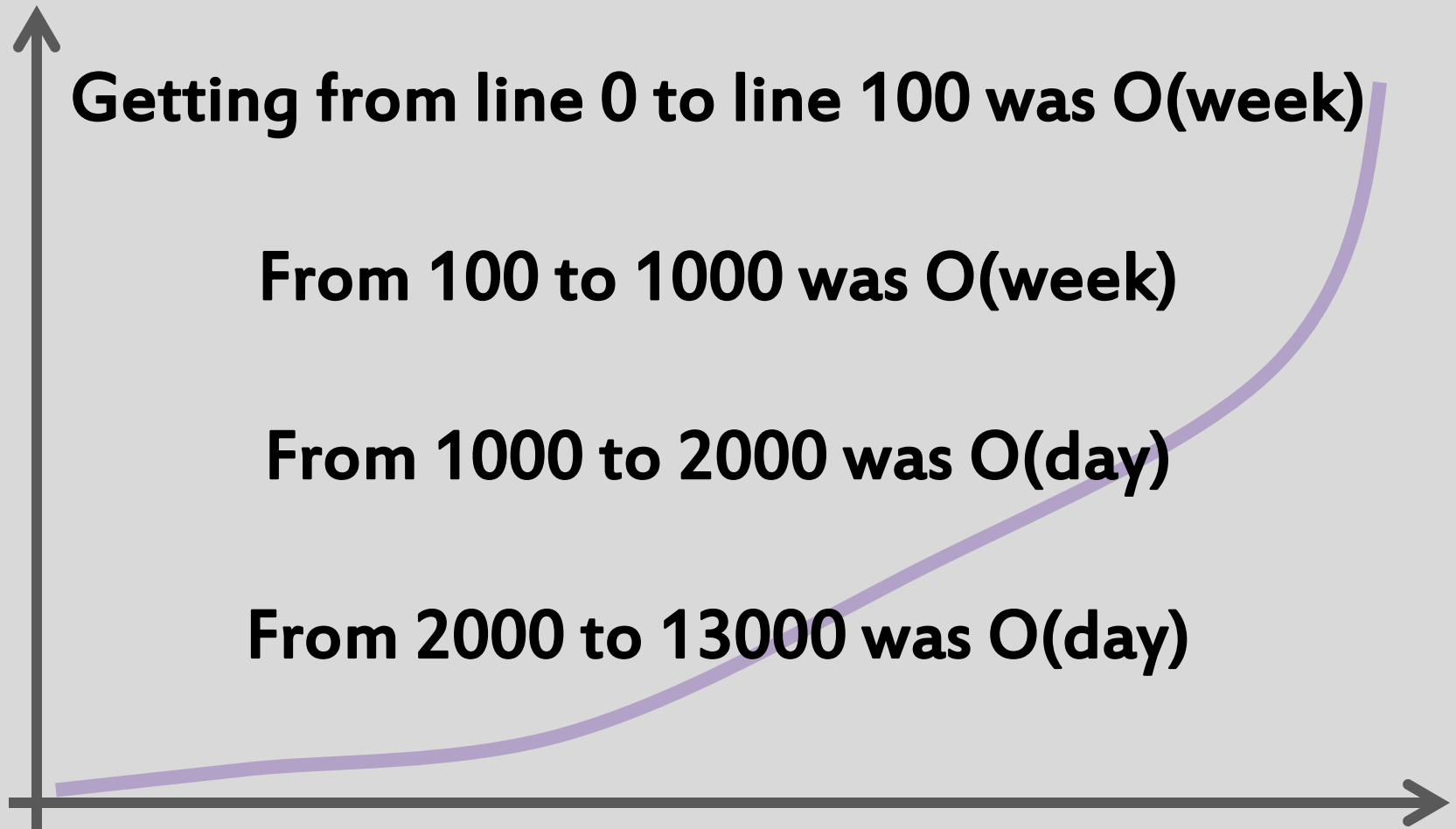
The CORE setting contains the built-in types and functions. It forms the outer scope of your program.

13,250

lines of Perl 6

That's a tough first test. 😊

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



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"

Around a week or two ago...

```
$ 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
→ just "hello world", but not cheating at all 😊**

Of course, still plenty of work to go

So, what next?

Pass the sanity tests
(13 test files)

Pass the specification tests
(740 test files)

Get the ecosystem working with it
(Modules, module installer, etc.)

Performance

Disclaimer: so far, not yet optimized.

"Make it work, then make it fast."

Performance

A few micro-benchmarks that may or may not be indicative; usual caveats apply

Levenshtein Benchmark on NQP

Runs around 15x faster on JVM than on Parrot

Parsing Rakudo CORE setting

Around 3x faster on JVM than on Parrot

`while $i < 1000000 { $i++ }` on Rakudo

Around 5x faster on JVM than Parrot

When?

**The June compiler release of Rakudo will be the first
with some level of JVM support**

**Aim for spectest equivalence with Rakudo on Parrot
in time for the August release**

**Aim for first Rakudo Star based on JVM sometime in
September**

JVM as a Perl 6 backend: the good

The JVM is very mature and well optimized

Serious interest from JVM developers to support dynamic languages, e.g. `invokedynamic`

Ability to handle static and dynamically typed languages well → promising for gradually typed

Solid, battle-hardened threads, so we can focus on nailing down this under-explored area of Perl 6

JVM backend weaknesses

Startup time is currently awful.

JVM backend weaknesses

Startup time is currently awful.

I mean, really, really awful.



JVM backend weaknesses

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

→ can be improved, 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, nowhere near as capable yet

→ "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 targetting multiple backends with a single
implementation, rather than one per VM**

Backend explosion?



We only have so many resources. Expectation: align resources with popularity.

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 the VM specific stuff**

Perl 6 users build awesome stuff, and enjoy doing so

Thank you!

Questions?

Blog: 6guts.wordpress.com

Twitter: [@jnthnwrthngtn](https://twitter.com/jnthnwrthngtn)

Email: jnthn@jnthn.net