A Raku API to Raku programs

The journey so far

Jonathan Worthington

Edument

Rakudo is currently going through two

huge

transformations

RakuAST

Create a document object model for the Raku language, and rewrite the compiler frontend to use it

(This talk)

Generalized dispatch

A huge internals overhaul and (mostly) simplification, that will help us optimize some currently awfully slow things

(My other talk)

Somehow, I've ended up leading both of them

How do I do it?

Really, it just needs

two things

Denial

Denial

about how much work it's going to be, in order to even begin

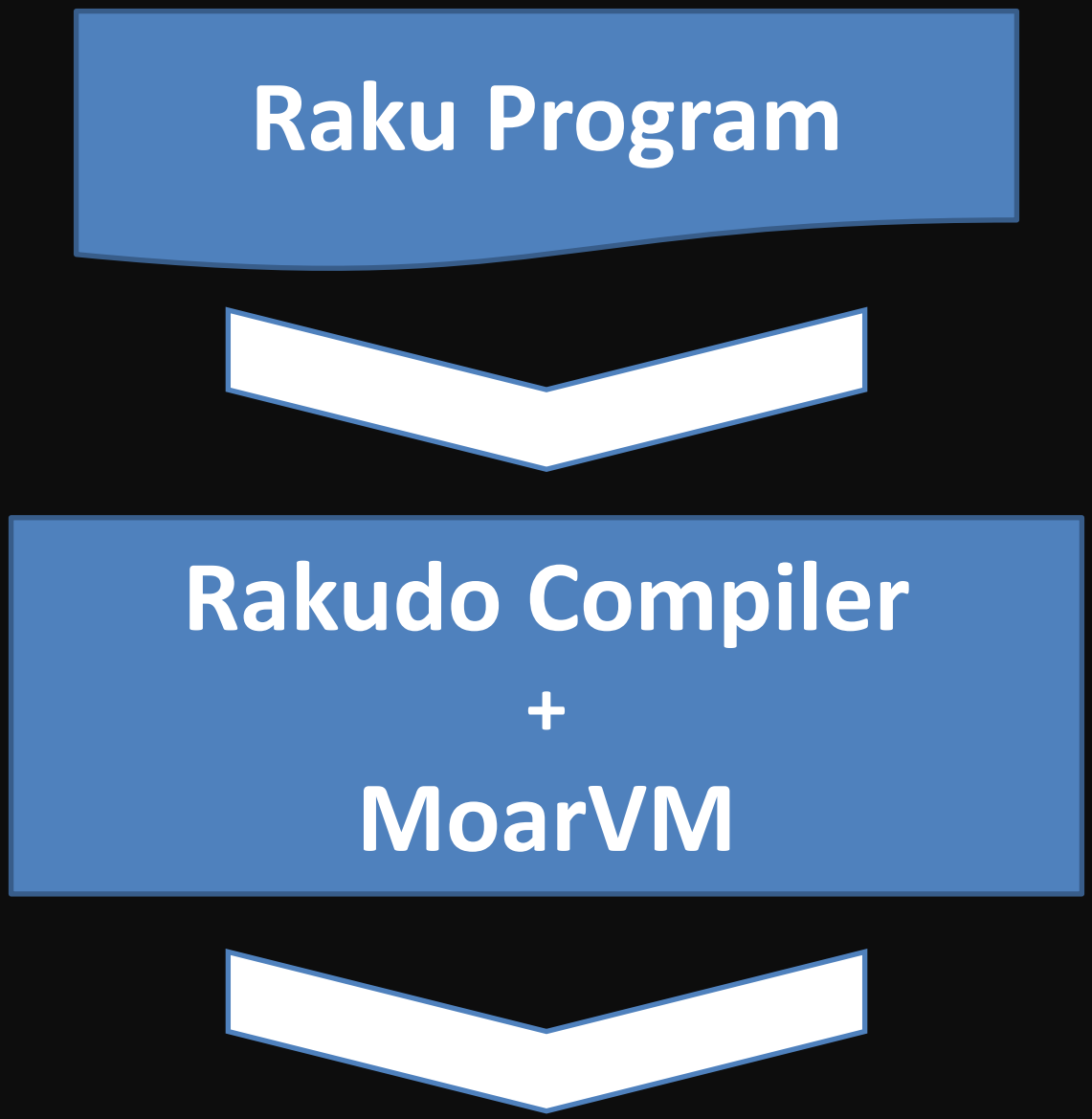
Stubbornness

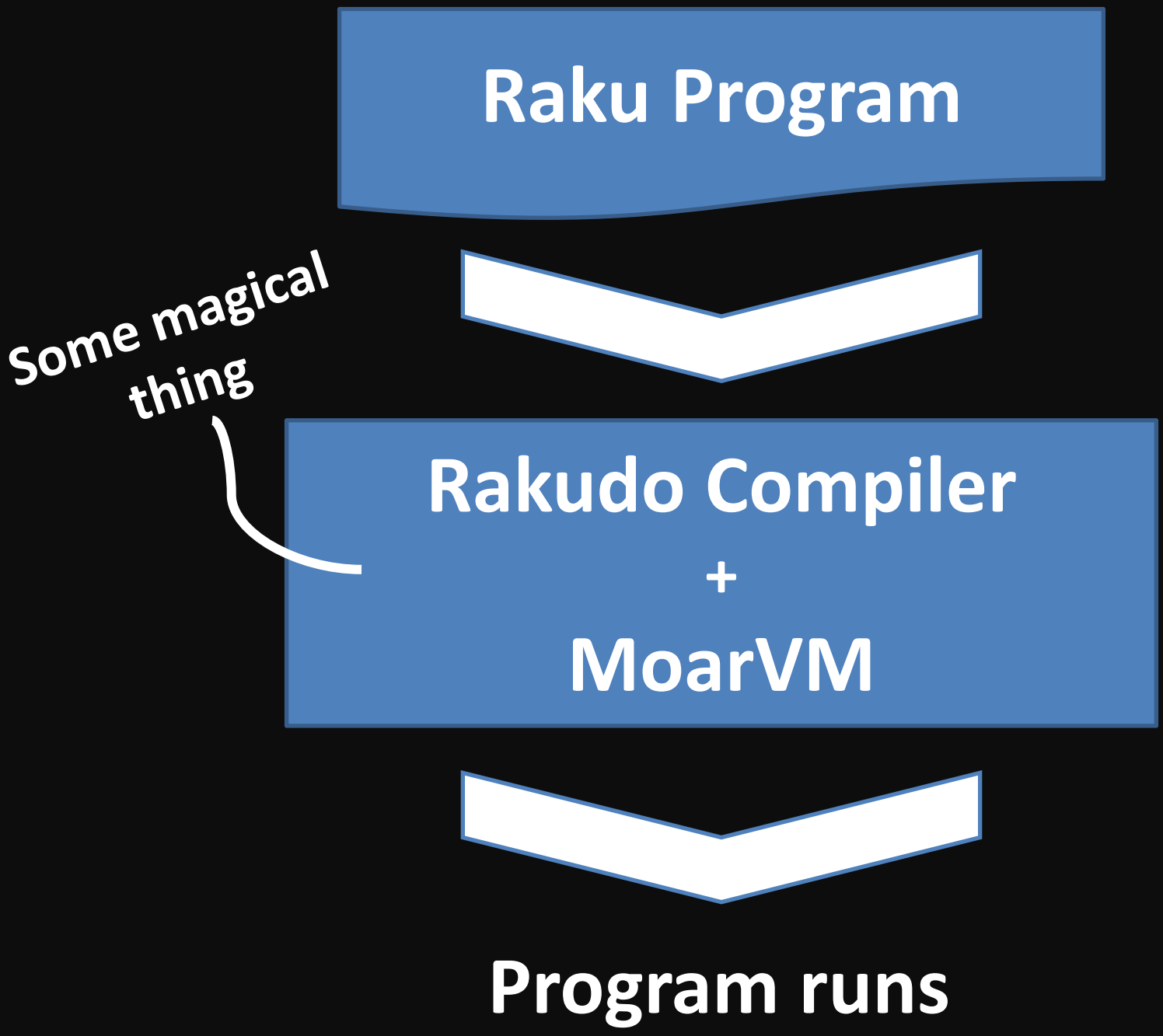
Stubbornness

to keep going even after realizing just how much work it is

First, a recap...

What is RakuAST?
Why might you care?


Raku Program
Rakudo Compiler
+
MoarVM
Program runs


Raku Program
Some magical thing
Rakudo Compiler
+
MoarVM
Program runs

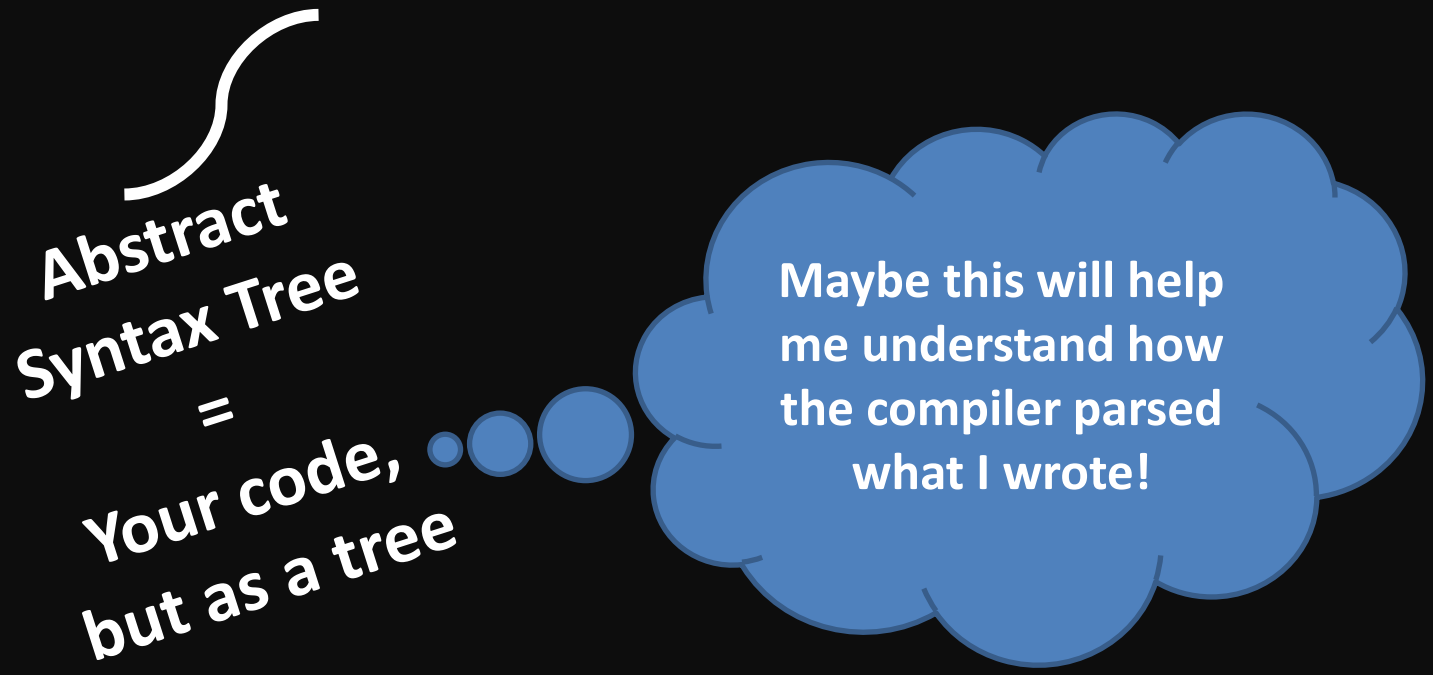
```
raku --target=ast -e 'say [*] 1..10'
```

```
raku --target=ast -e 'say [*] 1..10'
```

```
raku --target=ast -e 'say [*] 1..10'
```

Abstract
Syntax Tree
=
Your code,
but as a tree

```
raku --target=ast -e 'say [*] 1..10'
```



raku --target=ast -e 'say [*] 1..10'

```
- QAST::CompUnit  :W<?> :UNIT<?> :CAN_LOWER_TOPIC<?>
  [pre_deserialize]
    - QAST::Stmt
      - QAST::Stmt
        - QAST::Op(loadbytecode)
          - QAST::VM
            [moar]
              - QAST::SVal(ModuleLoader.moarvm)
            [jvm]
              - QAST::SVal(ModuleLoader.class)
            [js]
              - QAST::SVal(ModuleLoader)
        - QAST::Op(callmethod load_module)
          - QAST::Op(gethllsym)
            - QAST::SVal(nqp)
            - QAST::SVal(ModuleLoader)
          - QAST::SVal(Perl6::ModuleLoader)
      - QAST::Op(forceouterctx)
        - QAST::BVal(2)
        - QAST::Op(callmethod load_setting)
          - QAST::Op(getcurhllsym)
            - QAST::SVal(ModuleLoader)
          - QAST::SVal(CORE.d)
  [post_deserialize]
    - QAST::Stmts
      - QAST::Op(bind)
        - QAST::Var(attribute $!do)
          - QAST::WVal(Block)
          - QAST::WVal(Code)
        - QAST::BVal(1)
    - QAST::Op(bindcurhllsym)
      - QAST::SVal(GLOBAL)
      - QAST::WVal(GLOBAL)
  [load]
    - QAST::Op(call)
      - QAST::BVal(2)
  [children]
    - QAST::Block(:name(<unit-outer>) :cuid(2))  :in_stmt_mod<?> say [*] 1..10
    | - QAST::Var(local __args__ :decl(param))
    | - QAST::Stmts  say [*] 1..10
    | - QAST::Op(call)
    |   - QAST::Block(:name(<unit>) :cuid(1) :blocktype(declaration_static))
:IN_DECL<mainline>
    |   | - QAST::Stmts  say [*] 1..10
    |   |   - QAST::Var(lexical $¢ :decl(contvar))
    |   |   - QAST::Var(lexical $! :decl(contvar))
    |   |   - QAST::Var(lexical $/ :decl(contvar))
    |   |   - QAST::Var(lexical $_ :decl(contvar))
    |   |   - QAST::Var(lexical GLOBALish :decl(static))
    |   |   - QAST::Var(lexical EXPORT :decl(static))
    |   |   - QAST::Var(lexical $?PACKAGE :decl(static))
    |   |   - QAST::Var(lexical ::?PACKAGE :decl(static))
    |   |   - QAST::Var(lexical $=finish :decl(static))
    |   |   - QAST::Var(lexical $=pod :decl(static))
    |   |     [value]
    |   |       -
    |   |   - QAST::Var(lexical !UNIT_MARKER :decl(static))
    |   | - QAST::Stmts
    |   |   - QAST::Op(bind)
    |   |     - QAST::Var(local ctxsave :decl(var))
    |   |     - QAST::Var(contextual $*CTXSAVE)
    |   |   - QAST::Op(unless)
    |   |     - QAST::Op(isnull)
    |   |       - QAST::Var(local ctxsave)
    |   |     - QAST::Op(if)
    |   |       - QAST::Op(can)
    |   |         - QAST::Var(local ctxsave)
    |   |         - QAST::SVal(ctxsave)
    |   |       - QAST::Op(callmethod ctxsave)
    |   |         - QAST::Var(local ctxsave)
    |   | - QAST::Stmts
    |   |   - QAST::WVal(Array)
    |   |   - QAST::Stmts <sunk> say [*] 1..10
    |   |     - QAST::Stmt <sunk final> say [*] 1..10
    |   |       - QAST::Want <sunk>
    |   |         - QAST::Op(call &say) <sunk> :statement_id<1> say [*] 1..10
    |   |           - QAST::Op(call) <wanted> [*] 1..10
    |   |             - QAST::Op(call &METAOP_REDUCE_LEFT) <wanted>
    |   |               - QAST::Var(lexical &infix:<*>) <wanted>
    |   |             - QAST::Op(call &infix:<..>) <wanted> ..
    |   |               - QAST::Want <wanted> 1
    |   |                 - QAST::WVal(Int)
    |   |                 - Ii
    |   |                 - QAST::IVal(1)  1
    |   |               - QAST::Want <wanted> 10
    |   |                 - QAST::WVal(Int)
    |   |                 - Ii
    |   |                 - QAST::IVal(10)  10
    |   |           - v
    |   |           - QAST::Op(p6sink)
    |   |             - QAST::Op(call &say) <sunk> :statement_id<1> say [*] 1..10
    |   |               - QAST::Op(call) <wanted> [*] 1..10
    |   |                 - QAST::Op(call &METAOP_REDUCE_LEFT) <wanted>
    |   |                   - QAST::Var(lexical &infix:<*>) <wanted>
    |   |                 - QAST::Op(call &infix:<..>) <wanted> ..
    |   |                   - QAST::Want <wanted> 1
    |   |                     - QAST::WVal(Int)
    |   |                     - Ii
    |   |                     - QAST::IVal(1)  1
    |   |                   - QAST::Want <wanted> 10
    |   |                     - QAST::WVal(Int)
    |   |                     - Ii
    |   |                     - QAST::IVal(10)  10
    |   |   - QAST::WVal(Nil)
```

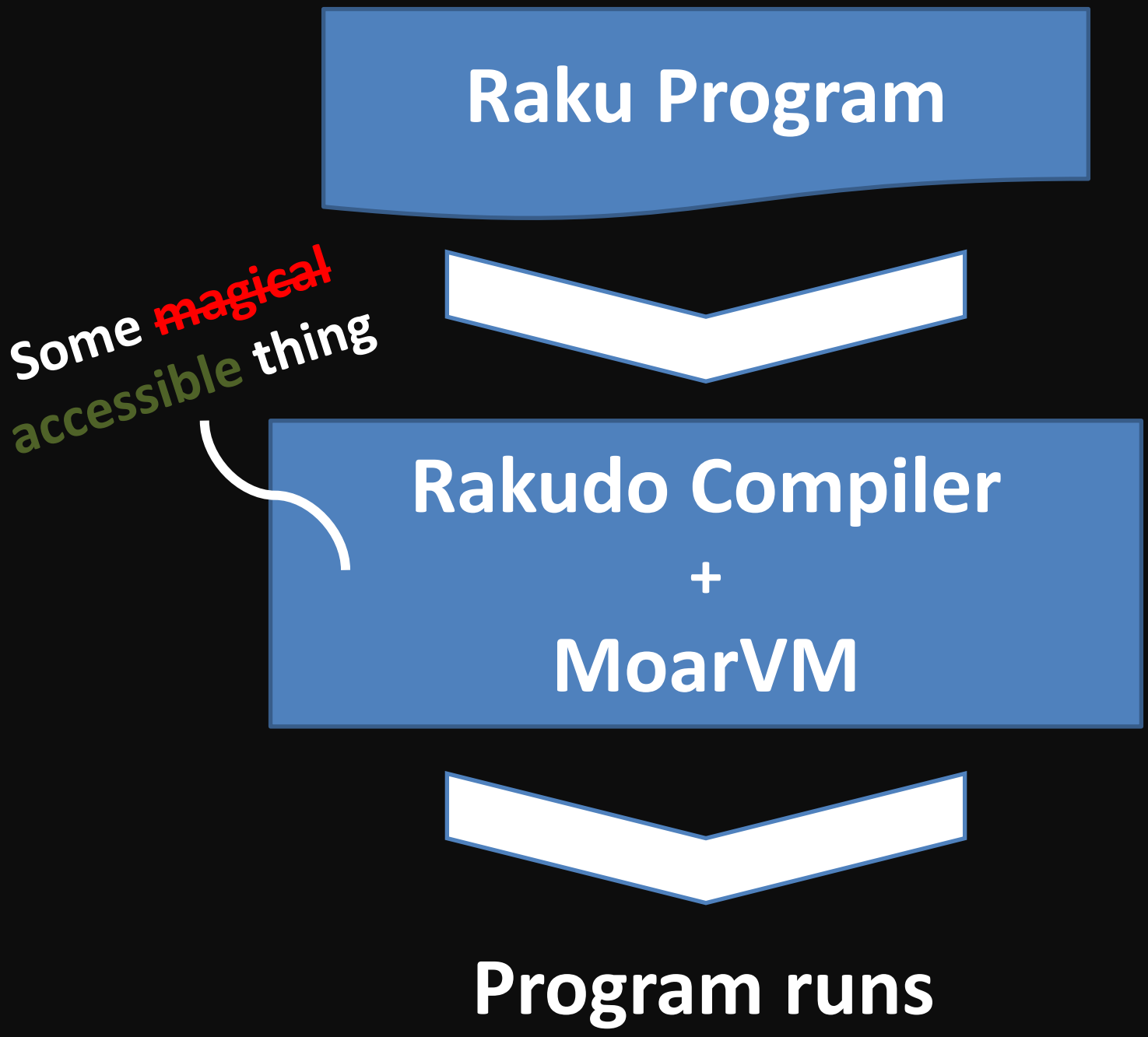
raku --target=ast -e 'say [*] 1..10'

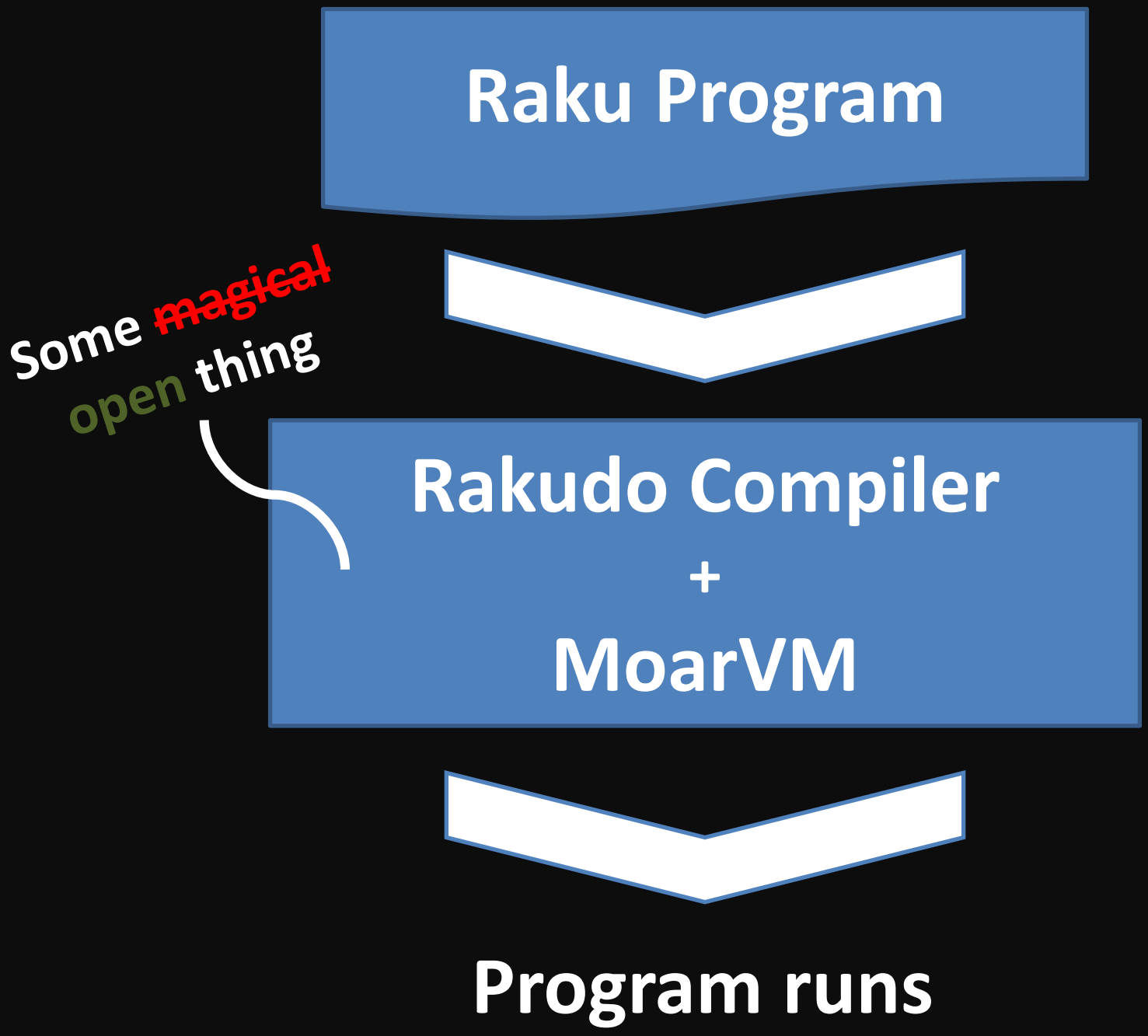
```
- QAST::CompUnit  :W<?> :UNIT<?> :CAN_LOWER_TOPIC<?>
  [pre_deserialize]
    - QAST::Stmt
      - QAST::Stmt
        - QAST::Op(loadbytecode)
          - QAST::VM
            [moar]
              - QAST::SVal(ModuleLoader.moarvm)
            [jvm]
              - QAST::SVal(ModuleLoader.class)
            [js]
              - QAST::SVal(ModuleLoader)
        - QAST::Op(callmethod load_module)
          - QAST::Op(gethllsym)
            - QAST::SVal(nqp)
            - QAST::SVal(ModuleLoader)
          - QAST::SVal(Perl6::ModuleLoader)
      - QAST::Op(forceouterctx)
        - QAST::BVal(2)
        - QAST::Op(callmethod load_setting)
          - QAST::Op(getcurhllsym)
            - QAST::SVal(ModuleLoader)
          - QAST::SVal(CORE.d)
  [post_deserialize]
    - QAST::Stmts
      - QAST::Op(bind)
        - QAST::Var(attribute $!do)
          - QAST::WVal(Block)
          - QAST::WVal(Code)
        - QAST::BVal(1)
    - QAST::Op(bindcurhllsym)
      - QAST::SVal(GLOBAL)
      - QAST::WVal(GLOBAL)
  [load]
    - QAST::Op(call)
      - QAST::BVal(2)
  [children]
    - QAST::Block(:name(<unit-outer>) :cuid(2))  :in_stmt_mod<?> say [*] 1..10
      | - QAST::Var(local __args__ :decl(param))
      | - QAST::Stmts  say [*] 1..10
      | - QAST::Op(call)
      |   - QAST::Block(:name(<unit>) :cuid(1) :blocktype(declaration_static))
:IN_DECL<mainline>
      |   | - QAST::Stmts  say [*] 1..10
      |   |   - QAST::Var(lexical $¢ :decl(contvar))
      |   |   - QAST::Var(lexical $! :decl(contvar))
      |   |   - QAST::Var(lexical $/ :decl(contvar))
      |   |   - QAST::Var(lexical $_ :decl(contvar))
      |   |   - QAST::Var(lexical GLOBALish :decl(static))
      |   |   - QAST::Var(lexical EXPORT :decl(static))
      |   |   - QAST::Var(lexical $?PACKAGE :decl(static))
      |   |   - QAST::Var(lexical ::?PACKAGE :decl(static))
      |   |   - QAST::Var(lexical $=finish :decl(static))
      |   |   - QAST::Var(lexical $=pod :decl(static))
      |   |     [value]
      |   |       -
      |   |   - QAST::Var(lexical !UNIT_MARKER :decl(static))
      |   | - QAST::Stmts
      |   |   - QAST::Op(bind)
      |   |     - QAST::Var(local ctxsave :decl(var))
      |   |     - QAST::Var(contextual $*CTXSAVE)
      |   |   - QAST::Op(unless)
      |   |     - QAST::Op(isnull)
      |   |       - QAST::Var(local ctxsave)
      |   |     - QAST::Op(if)
      |   |       - QAST::Op(can)
      |   |         - QAST::Var(local ctxsave)
      |   |         - QAST::SVal(ctxsave)
      |   |       - QAST::Op(callmethod ctxsave)
      |   |         - QAST::Var(local ctxsave)
      |   | - QAST::Stmts
      |   |   - QAST::WVal(Array)
      |   |   - QAST::Stmts <sunk> say [*] 1..10
      |   |     - QAST::Stmt <sunk final> say [*] 1..10
      |   |       - QAST::Want <sunk>
      |   |         - QAST::Op(call &say) <sunk> :statement_id<1> say [*] 1..10
      |   |           - QAST::Op(call) <wanted> [*] 1..10
      |   |             - QAST::Op(call &METAOP_REDUCE_LEFT) <wanted>
      |   |               - QAST::Var(lexical &infix:<*>) <wanted>
      |   |             - QAST::Op(call &infix:<..>) <wanted> ..
      |   |               - QAST::Want <wanted> 1
      |   |                 - QAST::WVal(Int)
      |   |                 - Ii
      |   |                 - QAST::IVal(1)  1
      |   |               - QAST::Want <wanted> 10
      |   |                 - QAST::WVal(Int)
      |   |                 - Ii
      |   |                 - QAST::IVal(10)  10
      |   |           - v
      |   |           - QAST::Op(p6sink)
      |   |             - QAST::Op(call &say) <sunk> :statement_id<1> say [*] 1..10
      |   |               - QAST::Op(call) <wanted> [*] 1..10
      |   |                 - QAST::Op(call &METAOP_REDUCE_LEFT) <wanted>
      |   |                   - QAST::Var(lexical &infix:<*>) <wanted>
      |   |                 - QAST::Op(call &infix:<..>) <wanted> ..
      |   |                   - QAST::Want <wanted> 1
      |   |                     - QAST::WVal(Int)
      |   |                     - Ii
      |   |                     - QAST::IVal(1)  1
      |   |                   - QAST::Want <wanted> 10
      |   |                     - QAST::WVal(Int)
      |   |                     - Ii
      |   |                     - QAST::IVal(10)  10
      |   |   - QAST::WVal(Nil)
```

This is QAST

And it's actually a fairly alright representation of a Raku program...

...if your primary interest is compiling it into bytecode


Raku Program
Some ~~magical~~ accessible thing
Rakudo Compiler
+
MoarVM
Program runs


Raku Program
Some ~~magical~~ open thing
Rakudo Compiler
+
MoarVM
Program runs

Rakudo knows Raku *really* well

But it's largely a walled garden

(and if you break in, the compiler team aren't responsible for the consequences)

You give it Raku source code, have it run...and that's about it

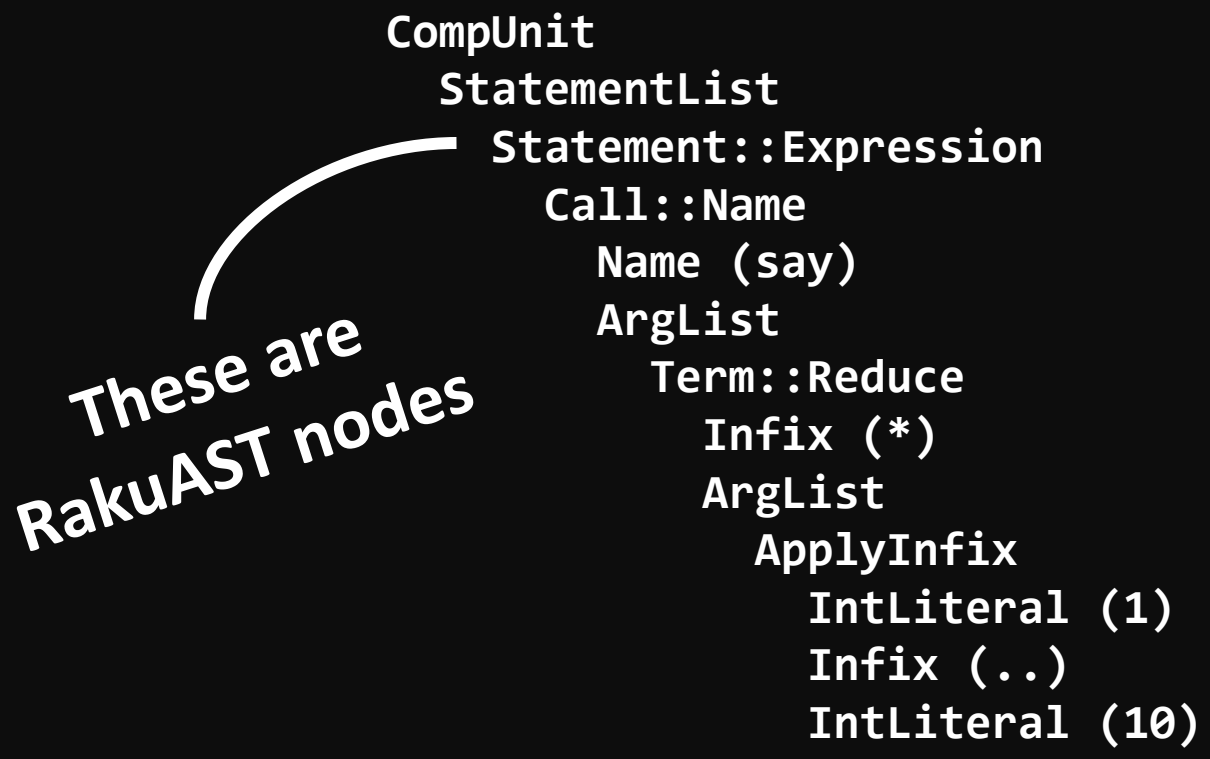
On the rakuast branch…

```
RAKUDO_RAKUAST=1 \
raku --target=ast -e 'say [*] 1..10'
```

```
CompUnit
  StatementList
    Statement::Expression
      Call::Name
        Name (say)
        ArgList
          Term::Reduce
            Infix (*)
            ArgList
              ApplyInfix
                IntLiteral (1)
                Infix (..)
                IntLiteral (10)
```

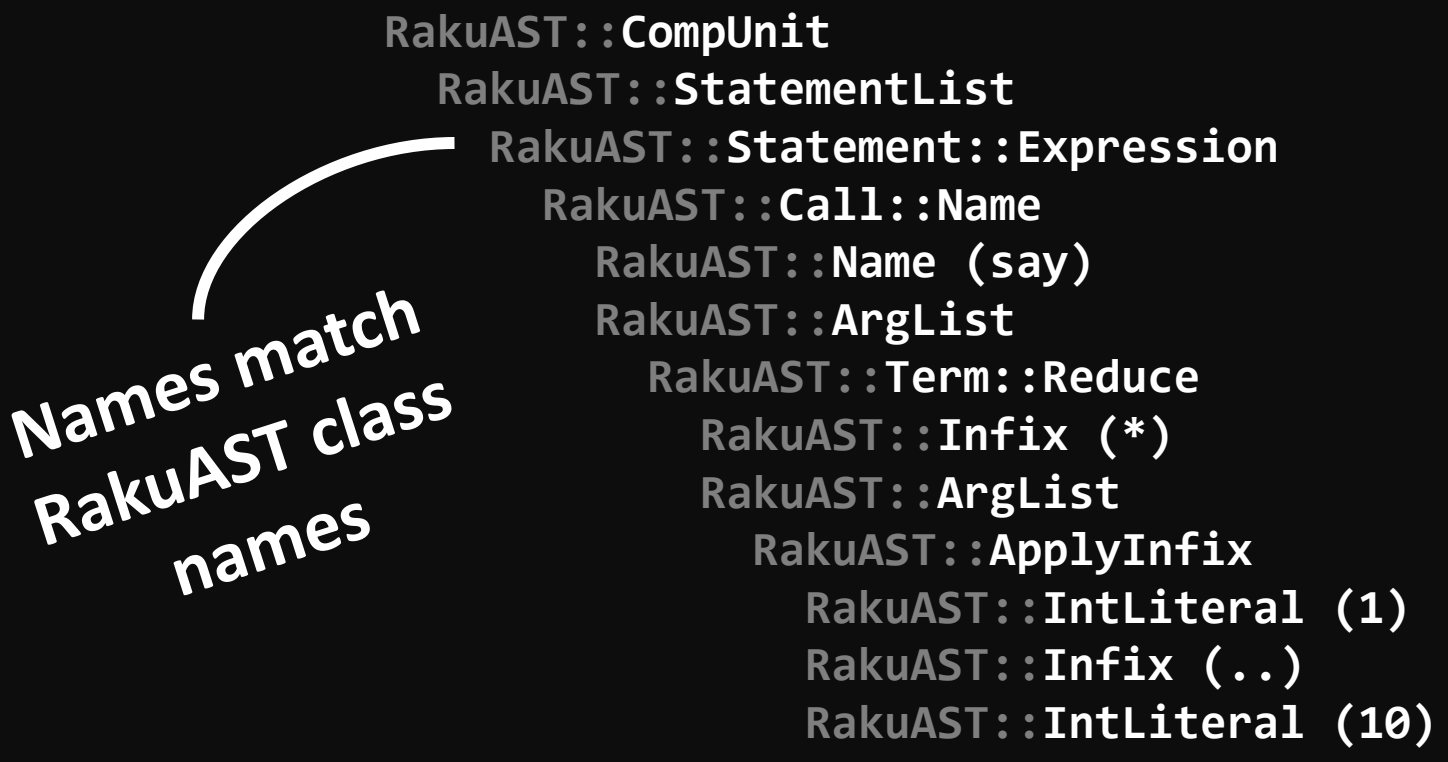
On the rakuast branch...

```
RAKUDO_RAKUAST=1 \
raku --target=ast -e 'say [*] 1..10'
```



On the rakuast branch…

```
RAKUDO_RAKUAST=1 \
raku --target=ast -e 'say [*] 1..10'
```



RakuAST is a proposed addition to the Raku language

So unlike QAST, it'll also be a supported feature with a stable API

Synthetic ASTs

RakuAST can be constructed and EVAL'd - a faster and safer option for modules that generate code

```
say EVAL RakuAST::IntLiteral.new(42);
```

The Rakudo compiler itself will construct RakuAST

Setting RAKUDO_RAKUAST=1 uses a new compiler frontend that constructs RakuAST, not QAST

Still a work in progress; passes 333 spectest files in full (25%)

The Rakudo compiler itself will construct RakuAST

Se 1 uses a
chat
ot QAST

Many spectests files make incidental use of language features, or test interactions of features, so this is an significant underestimate of how much of the language is covered by RakuAST

Still a work in progress; passes 333 spectest files in full (25%)

Far neater actions

Rakudo today parses using a Raku grammar and the action methods build up a QAST tree

```
method statement_prefix:sym<gather>($/) {
    my $past := unwanted($<blorst>.ast,'gather');
    $past.ann('past_block').push(QAST::WVal.new(
        :value($*W.find_single_symbol('Nil'))
    ));
    make QAST::Op.new( :op('call'), :name('&GATHER'), $past );
}
```

Far neater actions

The RakuAST-based frontend instead has the simpler job of constructing RakuAST

```
method statement_prefix:sym<gather>($/) {
    self.attach: $/,
        self.r('StatementPrefix', 'Gather').new($<blorst>.ast);
}
```

(Likely) far shorter actions

11,461 lines on master

2,300 lines on rakuast

(it's incomplete, but easily > 50% of what is needed)

Not just smaller, but simpler

In various places, Rakudo builds QAST trees, then later has to figure out what Raku constructs they originated from

Sink context handling is an especially painful example

A base for new features

A standardized AST, used by the compiler itself, is a prerequisite for a number of long-promised features

Macros

A macro is like a sub, but called at compile time and passed the ASTs of the arguments

Current macros are marked as experimental and *very* limited, as the AST arguments are opaque

Macros with RakuAST

The AST arguments will be RakuAST

Standardized part of the language, so they can be traversed, analyzed, manipulated, etc.

A great deal more useful

Slangs

Extend the Raku language syntax by providing extra grammar rules

But what about the semantics?
No good answers today.

Slangs with RakuAST

The semantics can be provided by producing RakuAST nodes

Since the compiler works in terms of those anyway, the pieces should all just fit neatly together

Custom compiler passes

Modules could provide extra compiler passes that perform analysis on the full RakuAST of the scope they are imported into

Linters
Type checkers
Domain-specific analyses

Proposed feature summary

(Very much subject to refinement)

	Macro	Slang	Compiler Pass
Phase	BEGIN	BEGIN	CHECK
Where it can be implemented	Current file *or* Module	Module	Module
Scope	Lexical	Lexical	Lexical
Syntax changes	Operators and terms only	Anything (I'm already scared)	N/A
Semantic changes	To AST arguments only	To anything in the scope it is used	To anything in the scope it is used
Planned for language version	E or F	F (Needs grammar standardization)	E

A glimpse into the future

At a conference a year ago, I was complaining about my `ECMA262Regex` module

Translates JavaScript regexes into Raku regexes

(Motivation: JavaScript regexes show up in standards, such as JSON schemas)

Parse with a Raku grammar ☺

```
token quantifier {
    <quantifier-prefix> '?'?
}
token quantifier-prefix {
    | '+'
    | '*'
    | '?'
    | '{' <decimal-digits> [ ',' <decimal-digits>? ]? '}'
}
```

Smash together strings ☹

```
method quantifier-prefix($/) {
    if not $/.Str.starts-with('{') {
        make ~$/;
    } else {
        # {n}
        if not $/.Str.contains(',') {
            make ' ** ' ~ ~$<decimal-digits>;
        } else {
            if $<decimal-digits>.elems == 1 {
                make ' ** ' ~
                      $<decimal-digits>[0].Str ~  '..* ';
            } else {
                make ' ** ' ~
                    $<decimal-digits>.map({ ~$_ }).join('..');
            }
        }
    }
}
```

This year, there's enough of RakuAST implemented to rework the module! *

* In a branch, of course. Also, all that follows uses the `rakuast` branch of Rakudo.

Challenge 1:
Just enough for simple literals

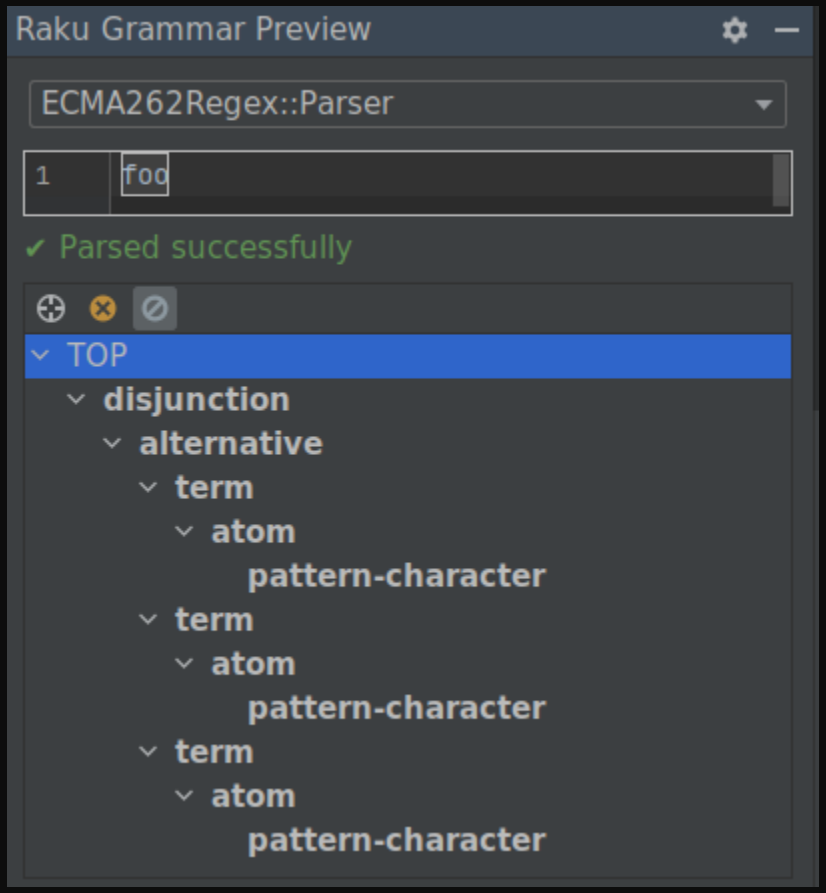
```
"tasty food" ~~ ECMA262Regex.compile("foo")
```

We need to rewrite some action methods...

...but how to know which?

Look at the parse tree!

(In Comma or use Grammar::Tracer)



```
token TOP {
    <disjunction>
}
token disjunction {
    <alternative>* % '|'
}
token alternative {
    <term>*
}
token term {
    <!before $>
    [
        | <assertion>
        | <atom> <quantifier>?
    ]
}
token atom {
    | <pattern-character>
    # ...other stuff we'll do later
}
token pattern-character {
    <-[^$\\.*+?()[\]{}|]>
}
```

```
token TOP {
    <disjunction>
}
token disjunction {
    <alternative>* % '|'
}
token alternative {
    <term>*
}
token term {
    <!before $>
    [
        | <assertion>
        | <atom> <quantifier>?
    ]
}
token atom {
    | <pattern-character>
    # ...other stuff we'll do later
}
token pattern-character {
    <-[^$\\.*+?()[\]{}|]>
}
```

```
token TOP {
    <disjunction>
}
token disjunction {
    <alternative>* % '|'
}
token alternative {
    <term>*
}
token term {
    <!before $>
    [
        | <assertion>
        | <atom> <quantifier>?
    ]
}
token atom {
    | <pattern-character>
    # ...other stuff we'll do later
}
token pattern-character {
    <-[^$\\.*+?()[\]{}|]>
}
```

Alternation (| |)

Concatenation

Literals

We need to know which RakuAST nodes to use...

...but how to find which?

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
```

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Regex::SequentialAlternation
          Regex::Sequence
            Regex::Literal (a)
            Regex::Literal (b)
          Regex::Sequence
            Regex::Literal (c)
            Regex::Literal (d)
```

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Reg  :SequentialAlternation
               equence
```

```
method TOP($/) {
    make RakuAST::QuotedRegex.new(body => $<disjunction>.made);
}
```

```
            Regex::Sequence
              Regex::Literal (c)
              Regex::Literal (d)
```

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Regex::SequentialAlternation
          Regex::Sequence
            Regex::Literal (a)
```

```
method disjunction($/) {
    my @branches = $<alternative>>>.made;
    make @branches == 1
            ?? @branches[0]
            !! RakuAST::Regex::SequentialAlternation.new:
                    |@branches;
}
```

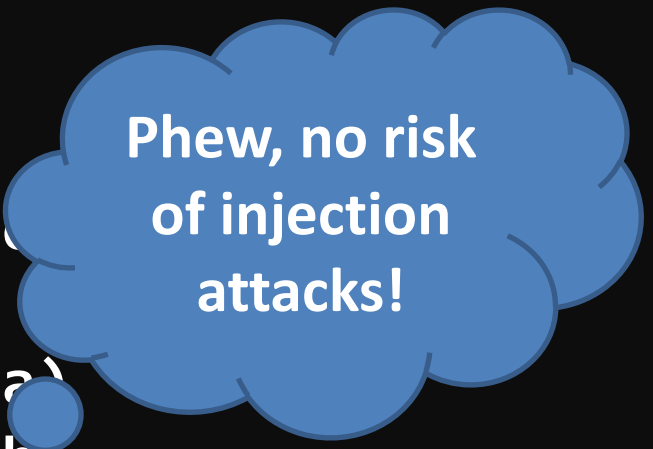
```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Regex::SequentialAlternation
          Regex::Sequence
            Reg  ::Literal (a)
                 iteral (b)
```

```
method alternative($/) {
    my @terms = $<term>>>.made;
    make @terms == 1
        ?? @terms[0]
        !! RakuAST::Regex::Sequence.new(|@terms);
}
```

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Regex::SequentialAlternation
          Regex::Sequence
            Regex::Literal (a)
            Re    ::Literal (b)
```

```
method pattern-character($/) {
    make RakuAST::Regex::Literal.new(~$/);
}
```

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/ab||cd/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Regex::SequentialAlt
          Regex::Sequence
            Regex::Literal (a)
            Re    ::Literal (b)
```




```
method pattern-character($/) {
    make RakuAST::Regex::Literal.new(~$/);
}
```

```
method term($/) {
    with $<atom> {
        my $atom = $<atom>.made;
        with $<quantifier> {
            !!! "nyi"
        }
        else {
            make $atom;
        }
    } else {
        !!! "nyi"
    }
}

method atom($/) {
    with $<pattern-character> {
        make $<pattern-character>.made;
    }
    else {
        !!! "nyi"
    }
}
```

Finally, plug it in...

...and it works!

```
method as-ast($str) {
    my $regex = ECMA262Regex::Parser.parse: $str,
        actions => ECMA262Regex::ToRakuAST;
    without $regex {
        die 'Regex is not valid!';
    }
    $regex.made
}

method compile($regex) {
    EVAL self.as-ast($regex)
}
```

Challenge 2:
Quantifiers

```
say "tasty food" ~~ ECMA262Regex.compile('o+')
「oo」
say "tasty food" ~~ ECMA262Regex.compile('o{2}')
「oo」
say "tasty food" ~~ ECMA262Regex.compile('o{1}')
「o」
say "tasty food" ~~ ECMA262Regex.compile('o{1,3}')
「oo」
say "tasty food" ~~ ECMA262Regex.compile('o{1,3}?')
「o」
```

The grammar rules

```
token quantifier {
    <quantifier-prefix> [$<frugal>='?']?
}
token quantifier-prefix {
    | '+'
    | '*'
    | '?'
    | '{'
      <from=decimal-digits>
      [ $<upto>=',' <to=decimal-digits>? ]?
      '}'
}
```

Find out about RakuAST

```
$ RAKUDO_RAKUAST=1 raku --target=ast -e '/o+|a**3..5/'
CompUnit
  StatementList
    Statement::Expression
      QuotedRegex
        Regex::Alternation
          Regex::QuantifiedAtom
            Regex::Literal (o)
            Regex::Quantifier::OneOrMore
          Regex::QuantifiedAtom
            Regex::Literal (a)
            Regex::Quantifier::Range
```

Create quantifier AST

```
method quantifier($/) {
    my $backtrack = $<frugal>
            ?? RakuAST::Regex::Backtrack::Frugal
            !! RakuAST::Regex::Backtrack::Greedy;
    given $<quantifier-prefix> {
        when .<from> && .<upto> {
            make RakuAST::Regex::Quantifier::Range.new:
                    :min(+.<from>), :max(+.<to> // Int), :$backtrack;
        }
        when .<from> {
            make RakuAST::Regex::Quantifier::Range.new:
                    :min(+.<from>), :max(+.<from>), :$backtrack;
        }
        when '+' {
            make RakuAST::Regex::Quantifier::OneOrMore.new: :$backtrack;
        }
        ...
    }
}
```

Apply it to the atom...

...and it works!

```
method term($/) {
    with $<atom> {
        my $atom = $<atom>.made;
        with $<quantifier> {
            make RakuAST::Regex::QuantifiedAtom.new:
                    :$atom, :quantifier(.made);
        }
        else {
            make $atom;
        }
    }
    else {
        !!! "nyi"
    }
}
```

I did a bunch more

It's pretty mechanical

See `rakuast` branch of the module if curious

One more glimpse into the future

A custom compiler pass

Simple example: add a CHECK-time compiler pass that looks for spelling mistakes in routine and package names

Disclaimer: the API you're about to see for adding compiler passes is provisional and almost certainly will change.

Example usage

```
use CodePolicy::SpellCheckedNames;

sub this-is-fine() {}
sub this-is-wierd() {}
sub spelink-is-hard() {}

class EnglishIsJustOdd {}
class INeedADikshunary {}
```

Some spell-checking bits

```
my %dict := set '/usr/share/dict/words'.IO.lines.map(*.lc);

sub check-kebab-case(Str $name, %wrong) {
    for $name.split('-') -> $word {
        unless %dict{lc $word} {
            %wrong{$word}++;
        }
    }
}

sub check-camel-case(Str $name, %wrong) {
    for $name.split(/<?:Lu>/, :skip-empty) -> $word {
        unless %dict{lc $word} {
            %wrong{$word}++;
        }
    }
}
```

An exception type

```
my class X::MisSpelled is X::Comp {
    has @.words;
    method message() {
        "Program has misspelled words in names:\n" ~
            @!words.join("\n").indent(4)
    }
}
```

Check a single node

```
sub spell-check(RakuAST::Node $node, %wrong) {
    given $node {
        when RakuAST::Routine {
            with .name -> $name {
                check-kebab-case($name.canonicalize, %wrong);
            }
        }
        when RakuAST::Package {
            with .name -> $name {
                check-camel-case($name.canonicalize, %wrong);
            }
        }
    }
}
```

The compiler pass itself

```
sub EXPORT-POST-CHECK-PASS(RakuAST::CompUnit $compunit,
        RakuAST::Resolver $resolver) {
    my %wrong;
    $compunit.visit: -> $node {
        spell-check($node, %wrong);
        True # visit childern
    }
    if %wrong {
        $resolver.add-worry: X::MisSpelled.new:
            words => %wrong.keys;
    }
}
```

And the result

```
$ RAKUDO_RAKUAST=1 raku -Ilib example.raku
Potential difficulties:
    Program has misspelled words in names:
        spelink
        wierd
        Dikshunary
```

A final thought

RakuAST represents a

change in the relationship

between Raku compiler and Raku developer

Thank you!

@ jonathan@edument.cz

W jnthn.net

 jnthnwrthngtn

 jnthn