Realizing Raku Macros

Jonathan Worthington | Edument

Taking stock

of where Rakudo is today and what macros mean

Why it's time

for us to overhaul the Rakudo compiler frontend

The design of RakuAST

a user-facing AST for the Raku language

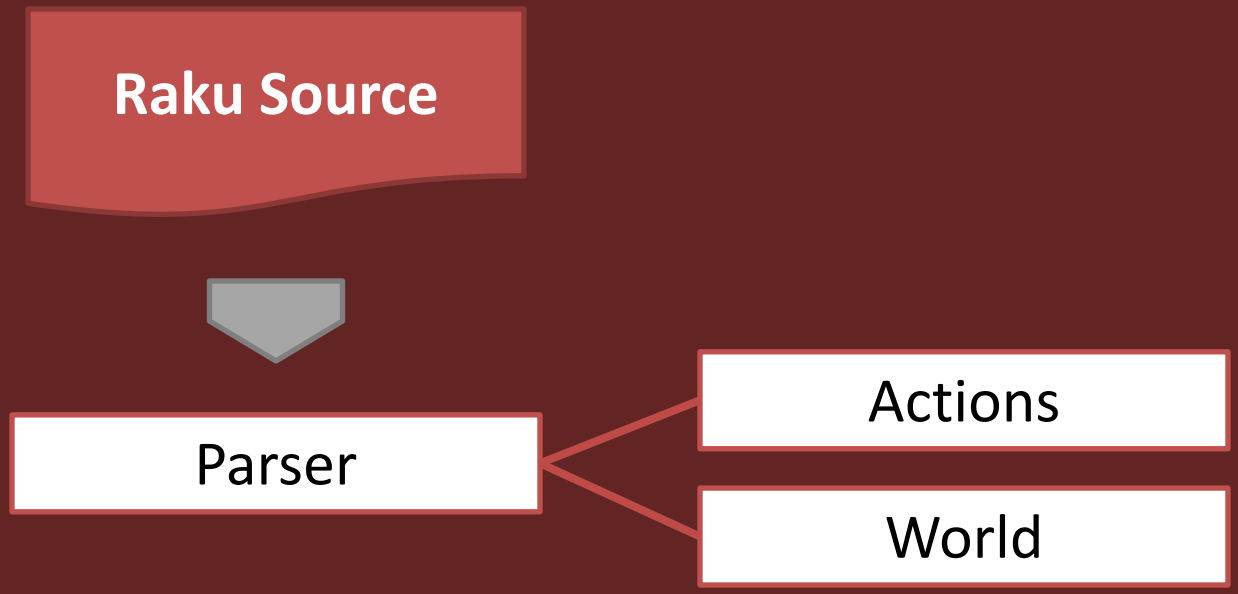
What happens next

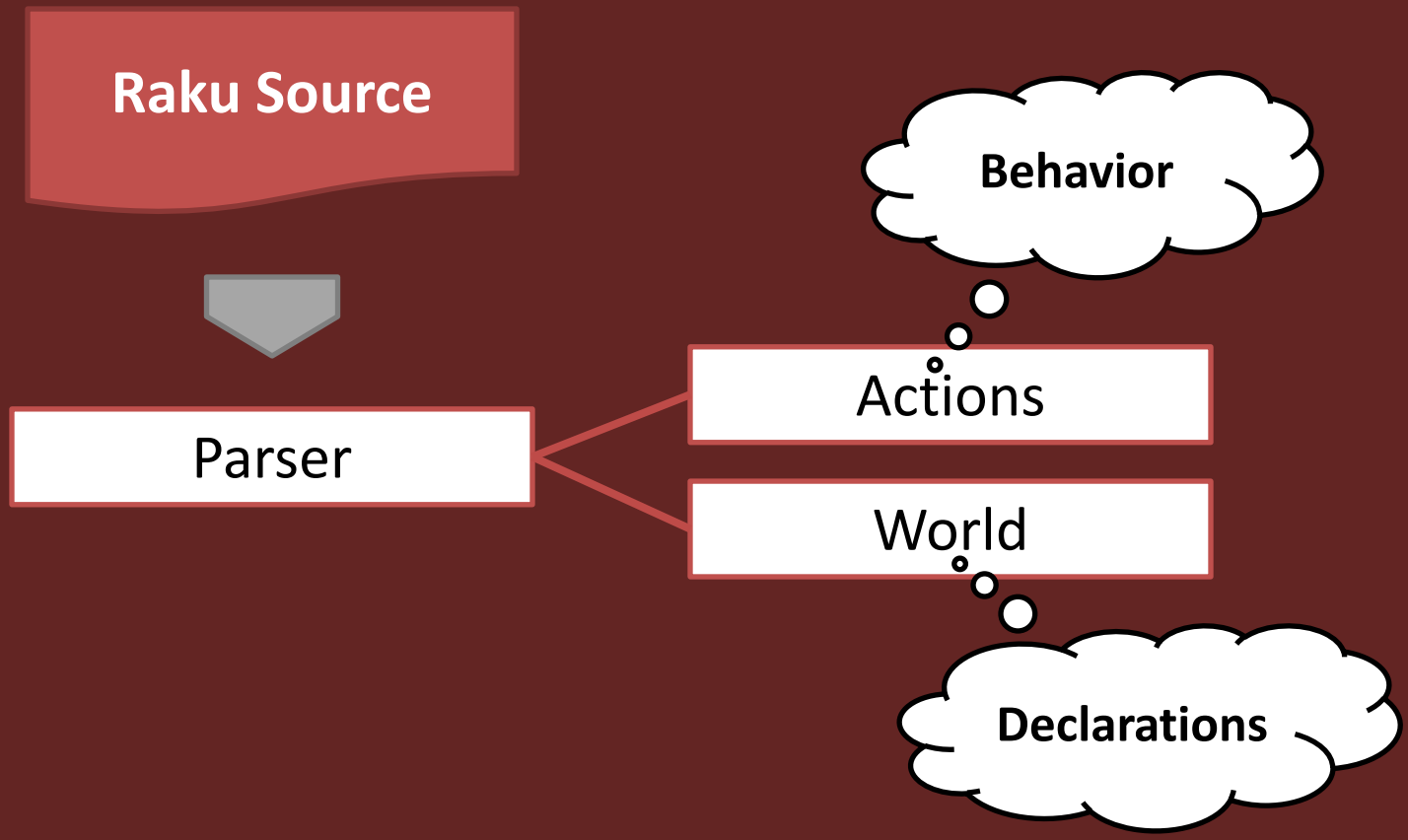
and where this might take us

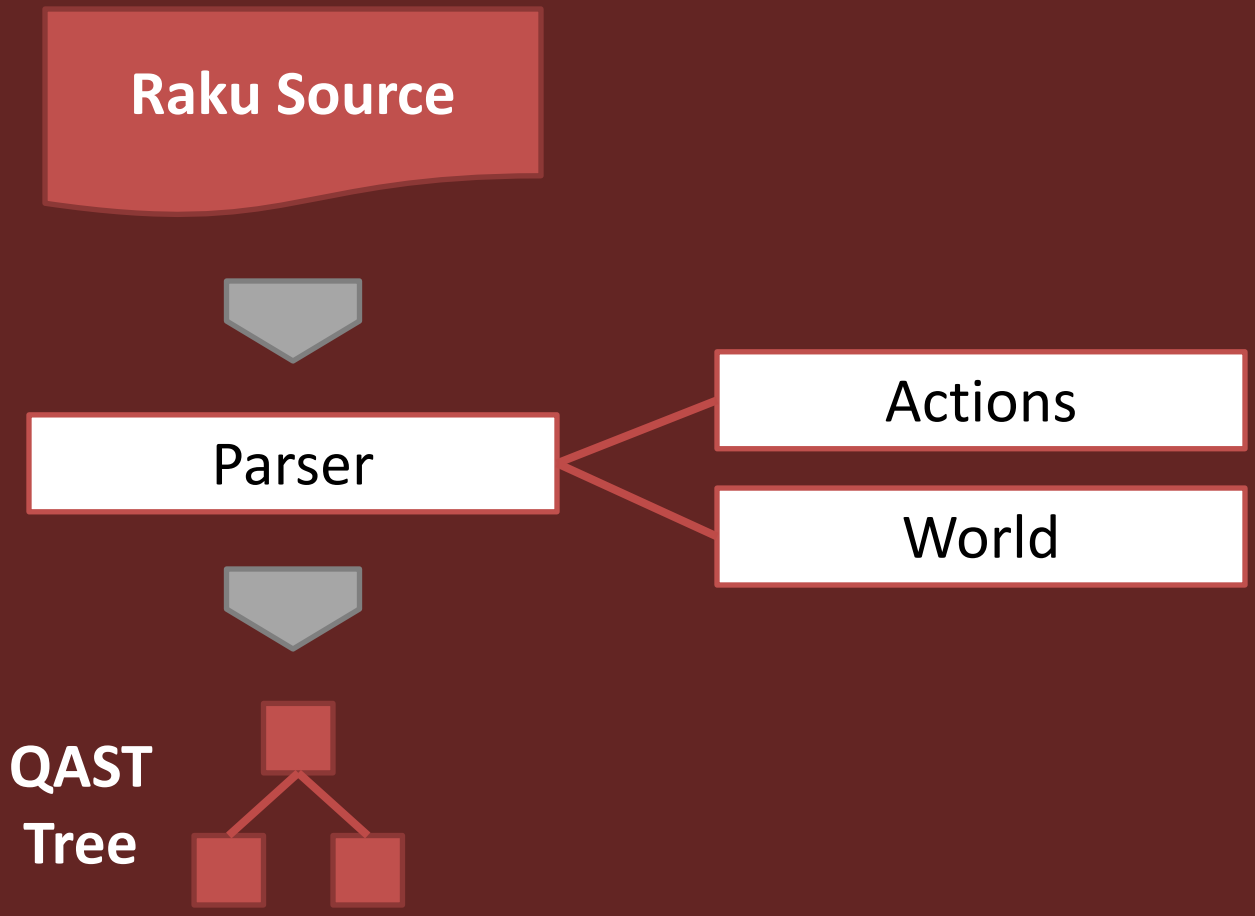
Taking stock

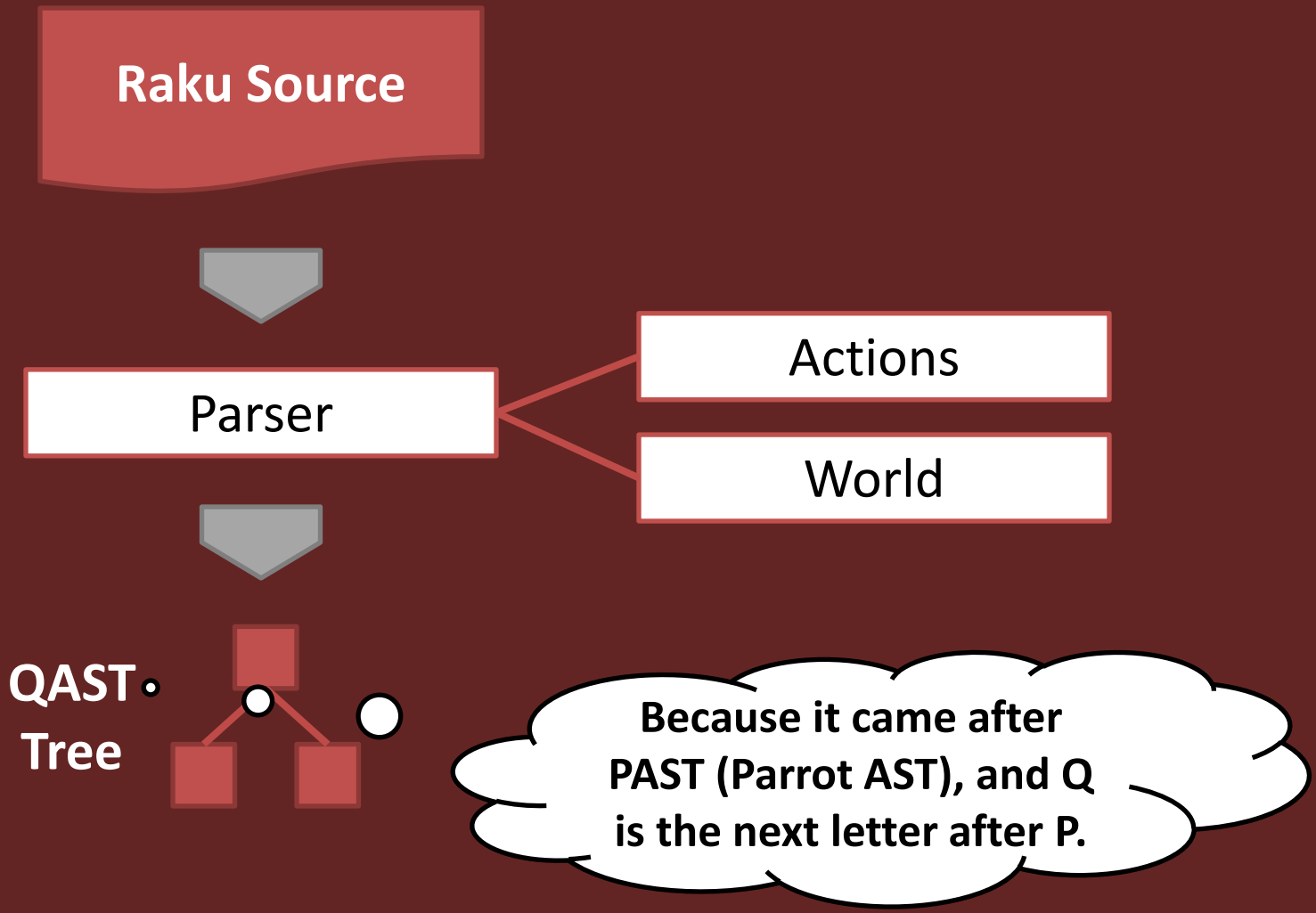
of where Rakudo is today
and what macros mean

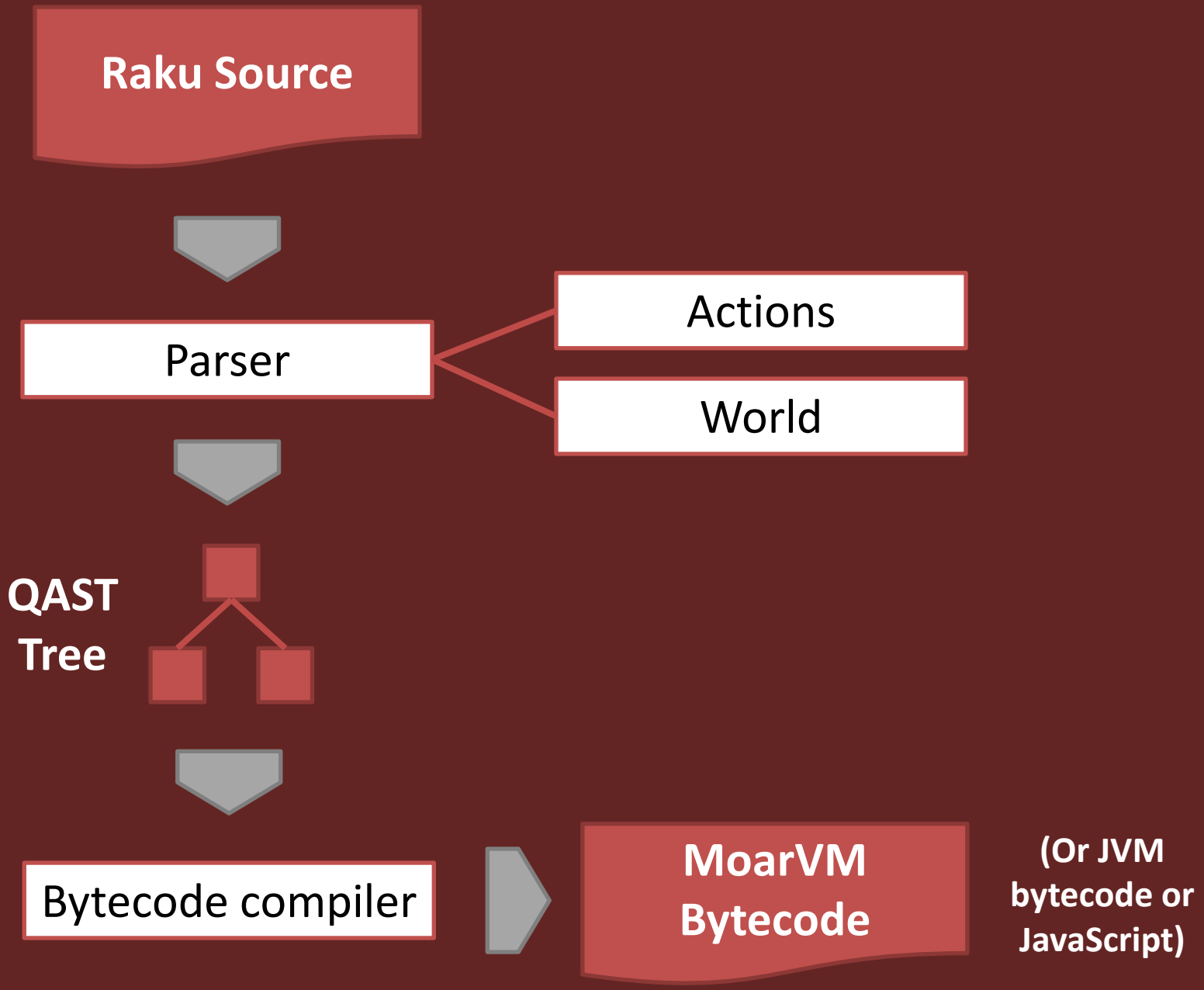
Raku Source

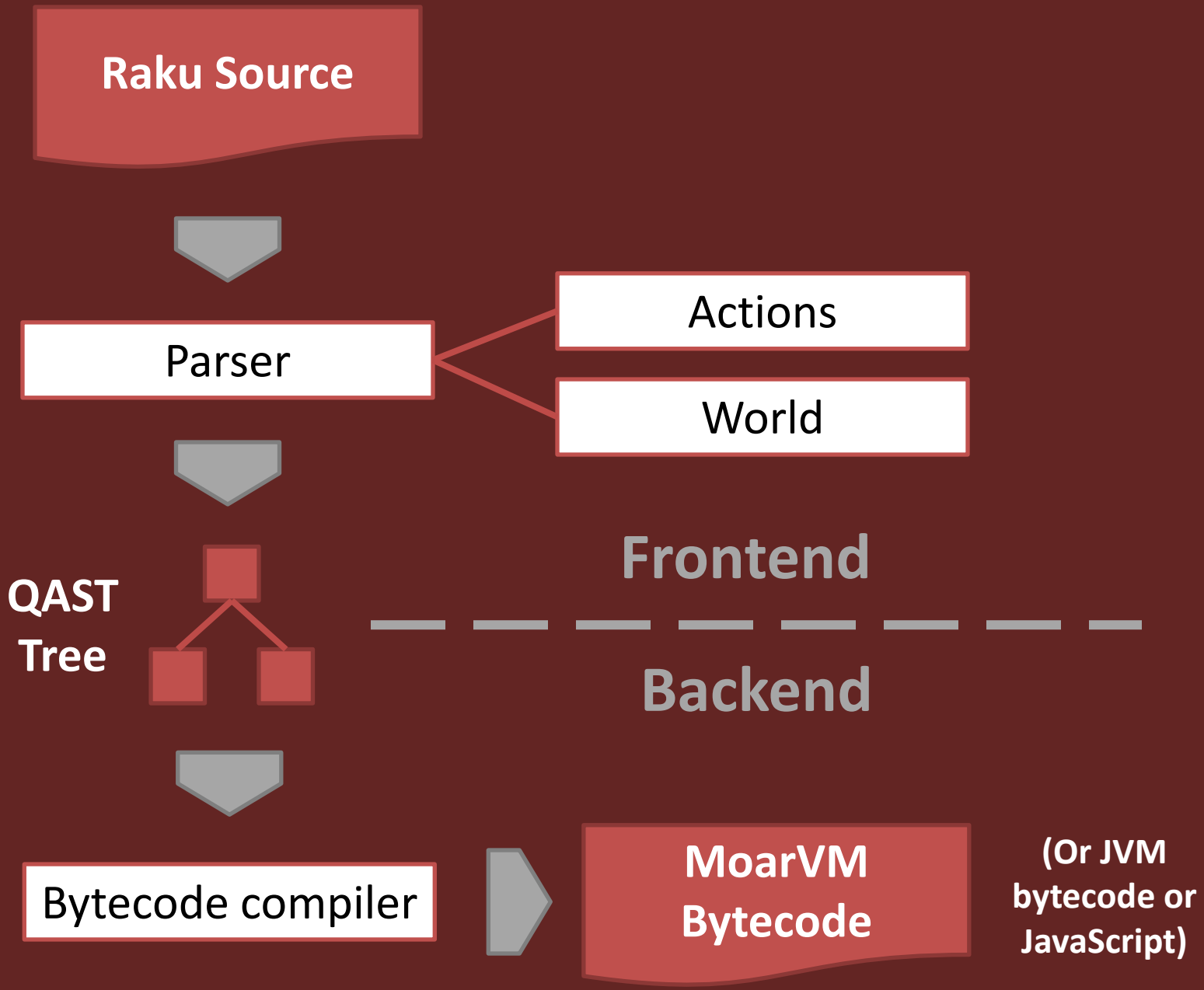

Raku Source
Parser
Actions
World


Raku Source
Parser
Actions
World
Behavior
Declarations


Raku Source
Parser
Actions
World
QAST
Tree


Raku Source
Parser
Actions
World
QAST
Tree
Because it came after
PAST (Parrot AST), and Q
is the next letter after P.


Raku Source
Parser
Actions
World
QAST
Tree
Bytecode compiler
MoarVM
Bytecode
(Or JVM
bytecode or
JavaScript)


Raku Source
Parser
Actions
World
QAST
Tree
Frontend
Backend
Bytecode compiler
MoarVM
Bytecode
(Or JVM
bytecode or
JavaScript)

Architecturally, the frontend hasn't changed much in a decade

The last major overhaul was done by this chap...

(The one on the left, I think...)

It's carried us all the way from

"Is it vaporware?"

to

"It's running in production!"

But now, we're running into the limits of QAST...

QAST: the good

Relatively small and simple compared to the size of the Raku language

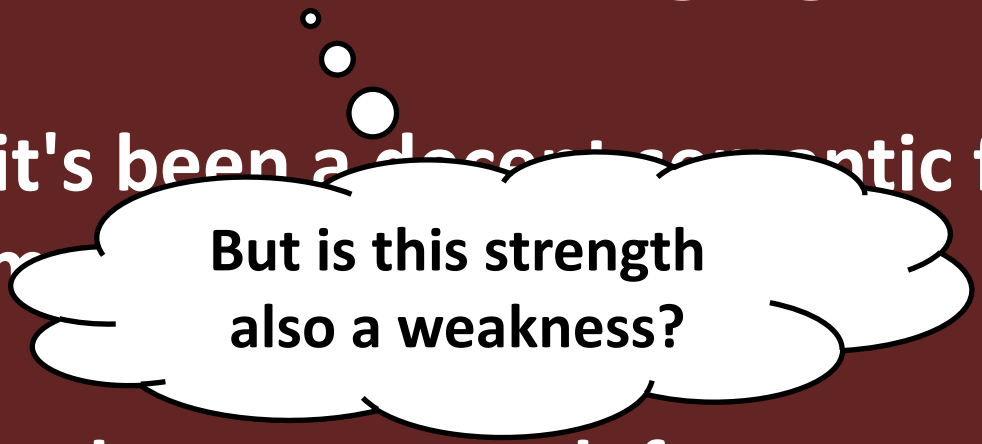
Even so, it's been a decent semantic fit for compiling Raku programs into

Has proven abstract enough for us to target a range of different backends

QAST: the good

Relatively small and simple compared to the size of the Raku language

Even so, it's been a decent semantic fit for com



Has proven abstract enough for us to target a range of different backends

QAST: the limitations

Primarily designed as a compiler-internal representation → not part of the spec

Doesn't fit within the Raku type system

Even internally, it's often a little *too* abstracted → we end up having to go back and figure out what stuff was

So what does this have to do with macros?

Macros receive an AST

```
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while-defined @a.shift, -> $val {
    say $val;
}
```

Macros receive an AST

```
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}
        }
    }
}

my @a = False, True, False;
while-defined @a.shift, -> $val {
    say $val;
}
```

Macro is called by the compiler after parsing its arguments

Macros receive an AST

```
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, Fals
while-defined @a.shift, -> $val {
    say $val;
}
```

quasi lets us write code with "holes"...

Macros receive an AST

```
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = Fals
while-define
    say $val;
}
```

which we fill with ASTs
using {{{ escape }}}
syntax

Macros receive an AST

```
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = Fals
while-define
    say $val;
}
```

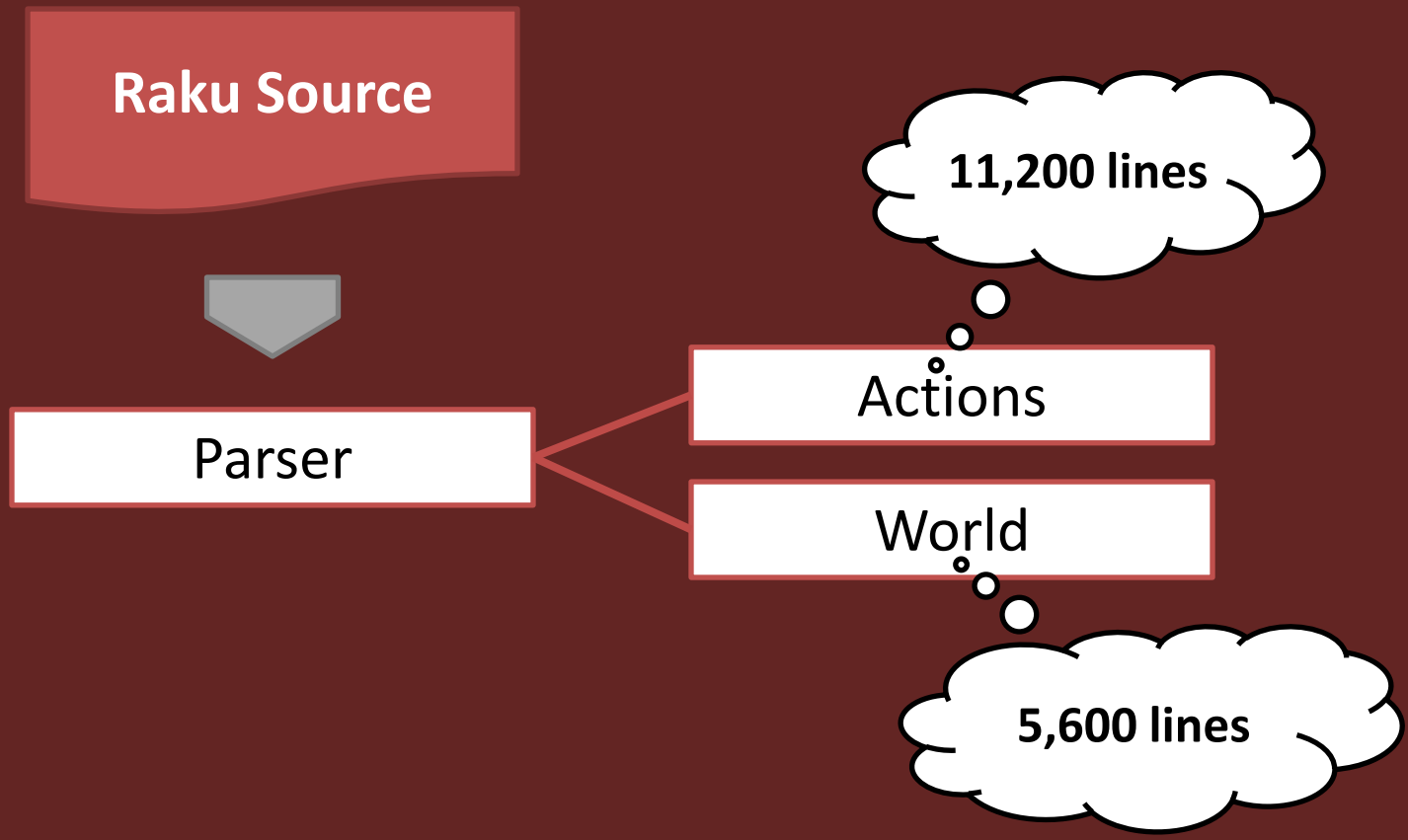
Not well formed if it's
not a block we can call,
but how do we check?

The most powerful uses of macros rely on being able to talk about the AST.

But ours isn't suitable for consumption by the Raku language user!

I've heard it said that

"Rakudo is too complex!"


Raku Source
Parser
Actions
World
11,200 lines
5,600 lines

But really, I think it's

too complicated

complex /ˈkɒmplɛks/

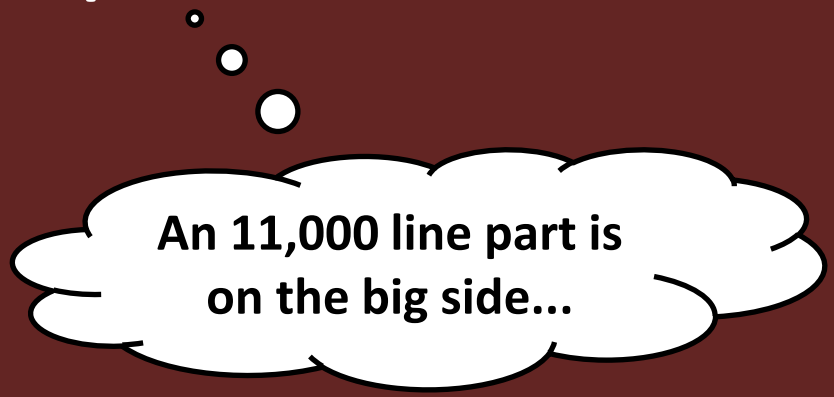
adjective

consisting of many different and connected parts

complex /ˈkɒmplɛks/

adjective

consisting of many different and connected parts



complex /ˈkɒmplɛks/

adjective

consisting of many different and connected parts

Complexity isn't inherently bad.

The challenge is how that complexity is *tackled*.

Why it's time

for us to overhaul the Rakudo compiler frontend

We want macros!

We want

useful

macros!

But not only that...

Cro::WebApp::Form

```
class Signup does Cro::WebApp::Form {
    has Str $.username
        is validated(/^<[A..Za..z0..9]>+$/,
            'Only alphanumerics are allowed');
    has Str $.password is required is password;
    has Str $.verify-password is required;

    ...
}
```

Cro::WebApp::Form

```
class Signup does Cro::WebApp::Form {
    has Str $.username
        is validated(/^<[A..Za..z0..9]>+$/,
            'Only alphanumerics are allowed');
    has Str $.password is required is password;
    has Str $.verify-password is required;

    ...
}
```

Traits handlers are invoked at compile time...

Cro::WebApp::Form

```
class Signup does Cro::WebApp::Form {
    has Str $.username
        is validated(/^<[A..Za..z0..9]>+$/,
            'Only alphanumerics are allowed');
    has Str $.password is required is password;
    has Str $.verify-password is required;

    ...
}
```

...we could take the AST of the regex and compile it into something for the HTML5 pattern attribute!

Cro::WebApp::Form

```
class Signup does Cro::WebApp::Form {
    has Str $.username
        is validated(/^<[A..Za..z0..9]>+$/,
            'Only alphanumerics are allowed');
    has Str $.password is required is password;
    has Str $.verify-password is required;

    ...
}
```

...we could take the AST of the regex and compile it into something for the HTML5 pattern attribute!

No reparsing!

ECMA262Regex

Takes an ECMA262 (JavaScript) regex and compiles it into the Raku regex syntax

Used for implementing JSON::Schema

ECMA262Regex

```
method control-letter($/) {
    my $name = %control-char-to-unicode-name{~$/};
    unless $name.defined {
        die 'Unknown control character escape is present: '
                ~ $/.Str;
    }
    make '"\c[' ~ $name ~ ']"';
}

method character-class($/) {
    my $start = '<';
    $start ~= '-' if $/.Str.starts-with('[^');
    $start ~= '[' ~ $<class-ranges>.made;
    make $start ~ ']>';
}
```

ECMA262Regex


```
method control-letter($/) {
    my $name = %control-char-to-un
    unless $name.defined {
        die 'Unknown control character escape is present: '
                ~ $/.Str;
    }
    make '"\c[' ~ $name ~ ']"'; ∘
}

method character-class($/) {
    my $start = '<';
    $start ~= '-' if $/.Str.starts-with('[^');
    $start ~= '[' ~ $<class-ranges>.made;
    make $start ~ ']>';
}
```

ECMA262Regex

```
method control-letter($/) {
    my $name = %control-char-to-u
    unless $name.defined {
        die 'Unknown control character escape is present: '
                ~ $/.Str;
    }
    make '\c[' ~ $name ~ ']';
}

method class-range($/) {
    my $start = $<class-range-char>[0].made;
    $start ~= '-' if $/.Str.starts-with('[^');
    $start ~= '[' ~ $<class-ranges>.made;
    make $start ~ ']>';
}
```

Ewwwwwwwww! Strings?!

How do we know it's well-formed?

ECMA262Regex

Could there be an injection attack?

Ewwwwwwwwww!
Strings?!

```
...-to-un...
        ...d {
        die 'Unknown control character escape is present: '
                ~ $/.Str;
    }
    make ..."\..." ... e ~ ']"';
...
...($/) {
    my $s... '<';
    $start ~= '-' if $/.Str.starts-with('[^');
    $start ~= '[' ~ $<class-ranges>.made;
    make $start ~ ']>';
}
```

How do we know it's well-formed?

ECMA262Regex

Could there be an injection attack?

Ewwwwwwwwww! Strings?!

```
                -to-u
                d {
        die 'Unknown control character escape is present: '
                ~ $/.Str;
    }
    make "[" ~ $/.Str ~ "]";
}

                (+/) {
    $start ~= '-' if $/.Str.starts-w
    $start ~= '[' ~ $<class-ranges>.made;
    make $start ~ ']>';
}
```

How do we know it's well-formed?

Raku compiler has to spend time parsing too! ☹

ECMA262Regex

Wouldn't it be nice if we could instead produce a tree representing the Raku regex?

Well formed by construction!
No data/syntax confusion!
No time wasted parsing again!

File::Ignore

Compiles Git-style ignore file patterns into Raku regexes

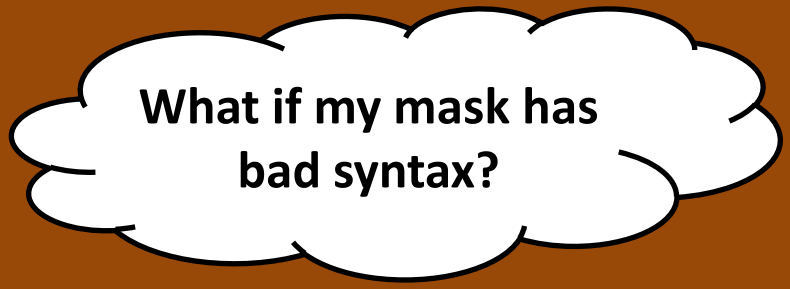
Same story as ECMA262Regex

JSON::Mask

```
my $mask = compile-mask('a,b,c');
mask($mask, %data1);
mask($mask, %data2);
mask($mask, %data3);
```

JSON::Mask

```
my $mask = compile-mask('a,b,c');
mask($mask, %data1);
mask($mask, %data2);
mask($mask, %data3);
```



JSON::Mask

```
my $mask = compile-mask('a,b,c');
mask($mask, %data1);
mask($mask, %data2);
mask($mask, %data3);
```

What if my mask has bad syntax?

Have to parse it every program run

JSON::Mask

```
my $mask = BEGIN compile-mask('a,b,c');
mask($mask, %data1);
mask($mask, %data2);
mask($mask, %data3);
```

Actually, BEGIN gives us a solution for this today - but with a macro we'd not need to write that!

JSON::Mask

```
my $mask = BEGIN compile-mask('a,b,c');
mask($mask, %data1);
mask($mask, %data2);
mask($mask, %data3);
```

(Also, JSON::Mask parses this once, but walks a tree to evaluate the mask. But with a nice Raku AST, we could more easily compile it.)

Cro::HTTP::Router

Internal DSL a la dynamic variables

```
my $app = route {
    get -> 'shop', $category {
        template 'category.crotmp', {
            products => $db.summaries($category)
        };
    }

    get -> 'catalogue', $category, $product {
        template 'product.crotmp', {
            product => $db.product($product)
        };
    }
}
```

Cro::HTTP::Router

Internal DSL a la dynamic variables

```
my $app = route {
    get -> 'shop', $category {
        template 'category.crotmp', {
            products => $db.summaries($category)
        };

        gory, $product {
        .crotmp', {
            product => $db.product($product)
        };
    }
}
```

Have to run the route block and build the matcher every startup

Cro::HTTP::Router

Internal DSL a la dynamic variables

```
my $app = route {
    get -> 'shop', $category {
        template 'category.crotmp', {
            products => $db.summaries($category)
        };
    }

    get -> 'catalogue'
        template 'product.
            product => $db.product($product)
        };
    }
}
```

Does this signature work with the router? (Yes, but nice to know at compile time if it would not!)

Cro::HTTP::Router

Internal DSL a la dynamic variables

```
my $app = route {
    get -> 'shop', $category {
        template 'category.crotmp', {
            products => $db.summaries($category)
        };
    }

    get -> 'shop', $category, $product {
        template 'product.crotmp', {
            product => $db.product($product)
        };
    }
}
```

Did I typo the template name? Wish it'd tell me when I compile, so I don't find out in production!

Cro::HTTP::Router

Internal DSL a la dynamic variables

```
my $app = route {
    get -> 'shop', $category {
        template 'category.crotmp', {
            products => $db.summaries($category)
        };
    }

                                    egory, $product {
                                    tmp', {
                                    duct($product)
}
```

What about a compile-time unused warning if the template never uses the data I give it?

Cro::HTTP::Router

And yes, the route table is compiled into....you guessed, a Raku regex!

Which is then EVAL'd!

Bet you can't guess what I'd prefer? ;-)

So much of the goodness we can get will only be achieved if we have a user-facing AST for Raku

And also...

We can make Rakudo better on the inside

Better collect responsibilities

Less figuring out "what was that" - especially in the optimizer

More accessible to language users

The design of RakuAST

a user-facing AST for the Raku language

Use cases

Constructed by Rakudo as it parses source code

Passed into macros, where it can be traversed (and maybe manipulated)

Constructed by Raku programs instead of producing code strings and calling EVAL

Use cases

Constructe

source co

Must be made of Raku objects that fit within the Raku type system...

Passed into macros, where it can be traversed (and maybe manipulated)

Constructed by Raku programs instead of producing code strings and calling EVAL

Use cases

Constructed by Rakudo as it parses source code

Passed into
traversed

...but can't compile it using Rakudo because Rakudo needs it to function!

Constructed by Raku programs instead of producing code strings and calling EVAL

So what can we do?

Piece the AST nodes together using the Meta-Object Protocol

Give them real Raku signature objects, so they introspect like other objects

Bodies of the methods are in NQP (our self-hosting Raku subset)


GOOD FATS FROM
PLANT-BASED OILS
CONTAINS OMEGA-3 ALA*
SIMPLE
INGREDIENTS
I can't believe
it's not
Butter!
60
CALORIES
PER 1 TBSP
the ORIGINAL
NET WT 15 OZ (425g) 45% VEGETABLE OIL SPREAD
Raku!

But...it's *so* tedious to write out all the MOP calls to do that!

So I wrote a compiler...

Subset of Raku classes, methods, and signatures with NQP bodies in

NQP code that pieces things together using the Raku MOP out

But what then?

My first idea

Start implementing RakuAST nodes

Gradually transition the action methods to producing them instead

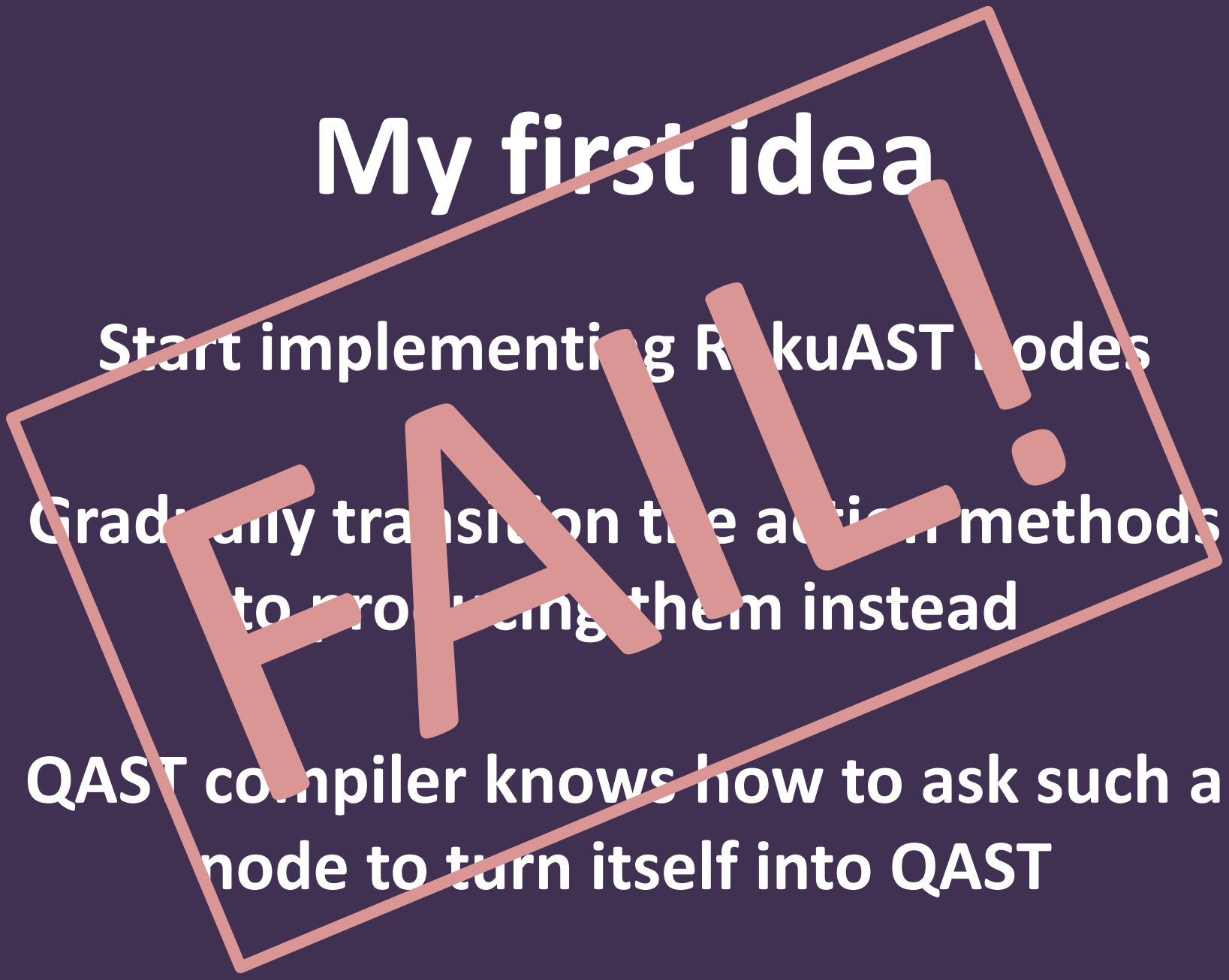
QAST compiler knows how to ask such a node to turn itself into QAST

My first idea

Start implementing RakuAST nodes

Gradually transition the action methods to producing them instead

QAST compiler knows how to ask such a node to turn itself into QAST



I did...

`RakuAST::IntLiteral`
`RakuAST::NumLiteral`
`RakuAST::RatLiteral`
`RakuAST::VersionLiteral`

But what next?

Pick almost *anything* else

Look at what it depends on

You'll end up in a recursion that sucks in most of the language

Infix operators? They use terms.

Subs (for example) are terms.

Subs have statements.

Statements have expressions.

Expressions have infix operators.

Variables?

Need a declaration model.

Declarations live in lexical scopes.

Thus blocks. Thus statements.
Thus expressions. Thus terms.
Thus variables.

Plan B

Implement EVAL of RakuASTs

Start with literals

Gradually build up from there

When most things work, adapt actions

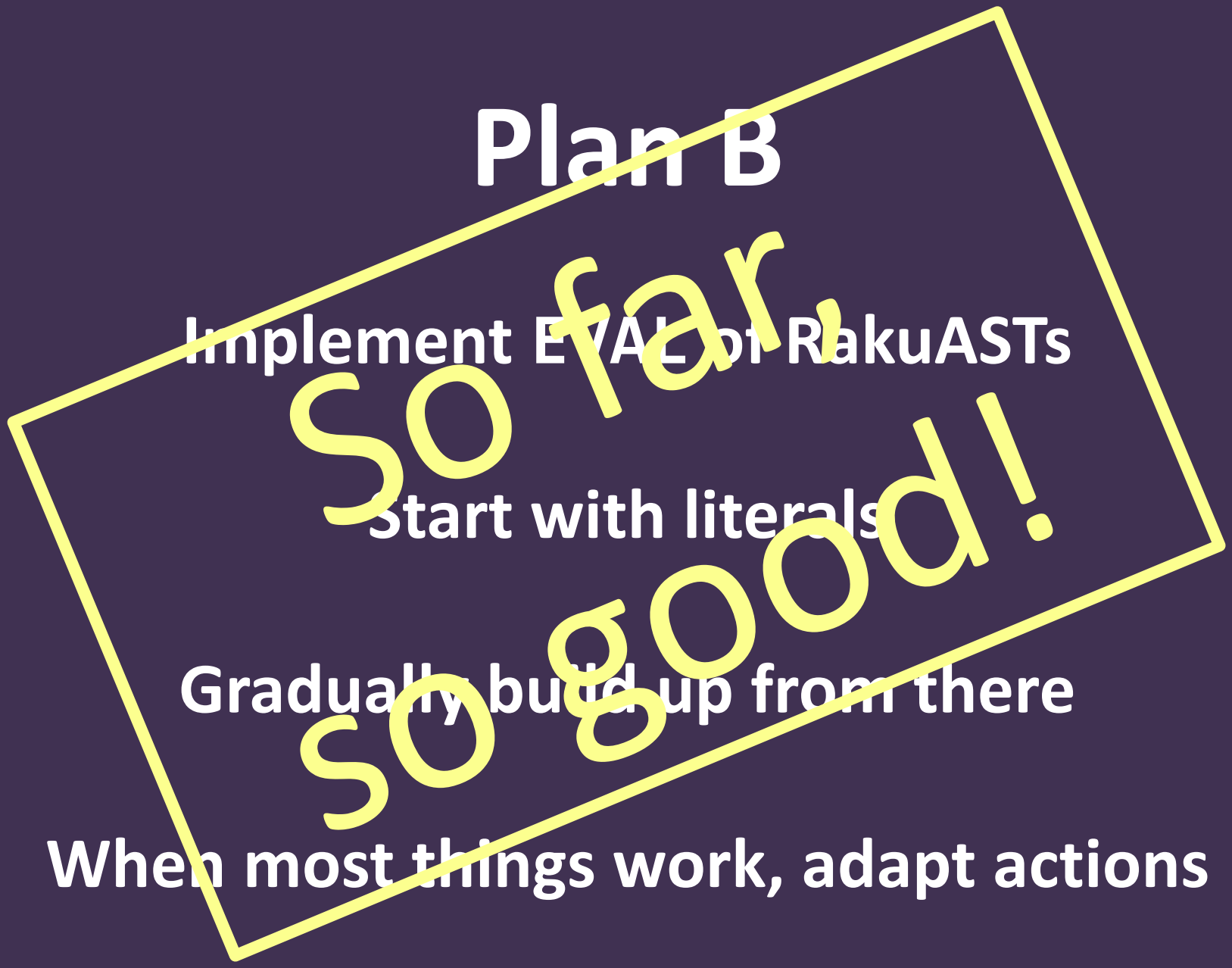
Plan B

Implement EVAL of RakuASTs

Start with literals

Gradually build up from there

When most things work, adapt actions



Plan B

Though I've only go so far...

Implement EVAL of RakuASTs

Start with literals

Gradually build up from there

When most things work, adapt actions

So far, so good!

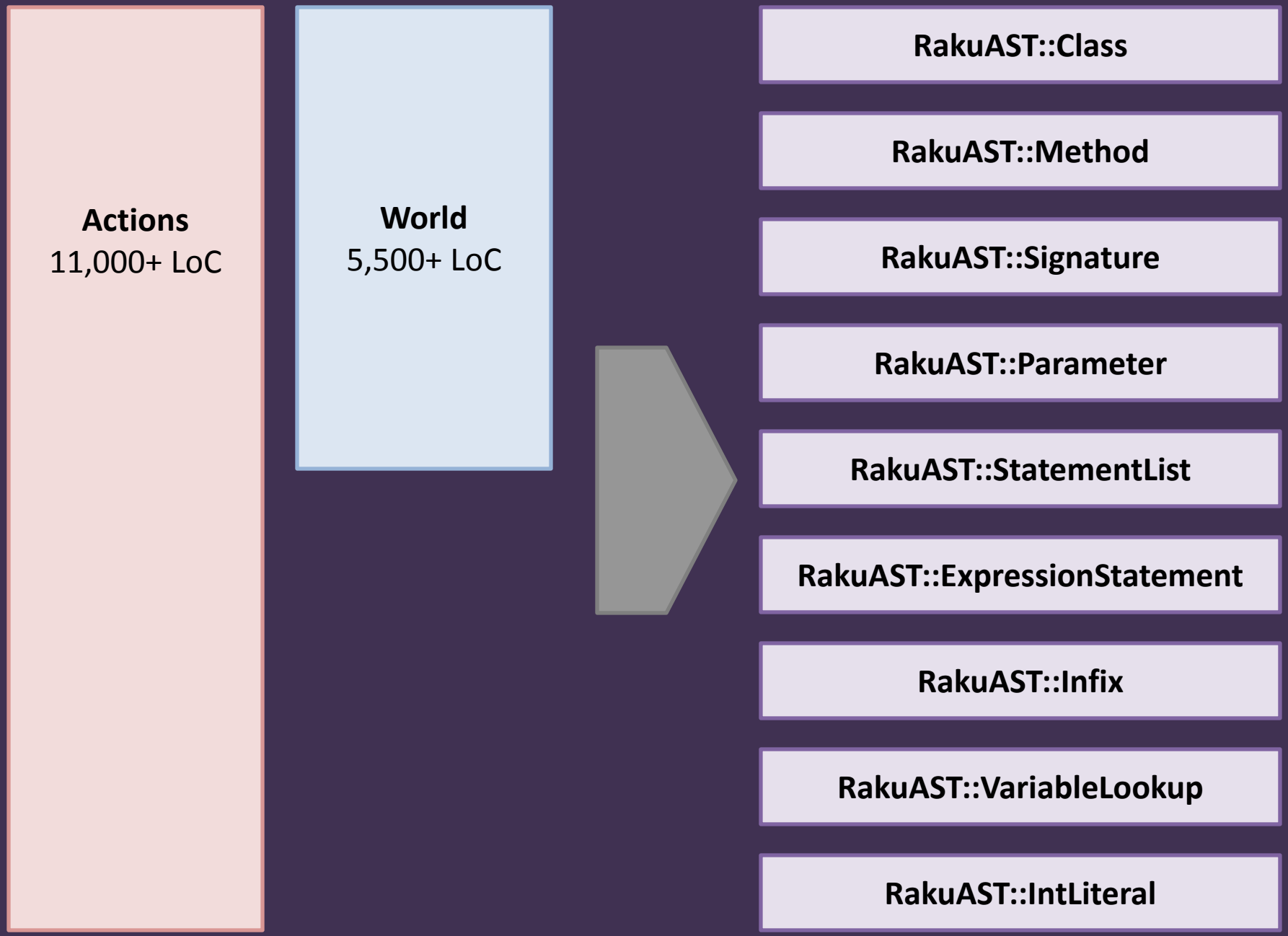
<live demo>

Some interesting design issues so far...

Actions *and* World get sucked in and chopped up

A class declaration implies both *meta-objects* and *runtime semantics*

RakuAST *must* be involved in both: we produce meta-objects once per quasi instantiation


Actions
11,000+ LoC
World
5,500+ LoC
RakuAST::Class
RakuAST::Method
RakuAST::Signature
RakuAST::Parameter
RakuAST::StatementList
RakuAST::ExpressionStatement
RakuAST::Infix
RakuAST::VariableLookup
RakuAST::IntLiteral

Meta-objects have a construction lifecycle

Producing the class meta-object needs the attribute meta-objects

But attribute meta-objects should refer to the class they're in

Need "stubs"/partial meta-objects

Lazier meta-objects too...

Now we make them very eagerly

Opening line of a class declaration creates the meta-object

In a quasi this must be deferred until interpolation time - but `BEGIN` time out of a quasi needs them up to date

What happens next

and where this might take us

I think by the summer, we can be most of the way to a RakuAST design, and have Rakudo using it

My aim is that we include RakuAST in the next Raku language release, in around a year's time

In scope:

The AST itself

Macros using the AST

Quasi quotes and splicing

Traits accessing the AST

Synthetic AST construction

Out of scope:

Non-expression macro arguments and quasi splicing

Non-operator syntax additions

Slangs

User-defined compiler passes

The things that are in scope are sufficient for dealing with all of the examples I gave

The things out of scope will be considered in future Raku language versions

And what might be in store for Rakudo's internal architecture?

(Disclaimer: this is all really speculative!)

A tree-walking evaluator for RakuAST

Cheapen simple, short-running `BEGIN` and EVAL by not doing the whole compilation thing

Which can be taken further...

Always start off tree-walking, and only compile what's hot?

More language-aware specializations at AST level? Feels like use-as-r-value style optimizations are easier at this level.

LessVM?

New interface to MoarVM

Same garbage collector, object model, Unicode support, JIT, etc.

Targeted at interpreter cooperation rather than being given an entire compilation unit as bytecode

A closing thought

Developer experience matters.

With Raku macros, we give module developers the power to build safer, richer, development experiences.

Let's do it.

Thank you!

@ jonathan@edument.cz

W jnthn.net

 jnthnwrthngtn

 jnthn