RakuAST: a foundation for Raku macros

Jonathan Worthington
Edument
Two questions that you may already have
What on earth is an AST?
AST

≈

What compiler folk call a Document Object Model for a programming language
Who on earth are you?
I do Raku things...
Rakudo compiler architect
MoarVM founder and architect
Raku concurrency designer
Founder of Cro
I do Raku things...

Rakudo compiler architect
MoarVM founder and architect
Raku concurrency designer
Founder of Cro

And lead a team at Edument
building developer tooling...

Including the Comma IDE for Raku
IntelliJ platform consultancy
Compiler/language design consultancy
The motivation for making an AST form of Raku part of the language

The design of RakuAST, and a compiler based around it

The progress on implementing RakuAST so far

The impact of RakuAST on Raku users
The motivation for making an AST form of Raku part of the language
"It'd be cool to have macros!"
"It'd be cool to have macros!"
"It'd be cool to have macros!"

I mean, it would be, but it's only one motivation for all of this work...
There's more than one way to macro...
Textual macros
(a la C)
Textual macros

(a la C)

Really, they work at the token level
I'm not saying textual macros aren't fun...
#define do       {
#define end       }

int main() do
  printf("Phew, no curlies!\n");
end
I mean, they are fun, until some day they aren't.

#define THING_HEADER_SIZE    16
#define THING_BODY_SIZE      40
#define THING_SIZE \
    THING_HEADER_SIZE + THING_BODY_SIZE
// Allocate memory for things.
Thing *things = malloc(
    num_things * THING_SIZE);

// Allocate memory for things.
Thing *things = malloc(num_things * THING_SIZE);
// Allocate memory for things.
Thing *things = malloc(
    num_things * THING_HEADER_SIZE + THING_BODY_SIZE);
// Allocate memory for things.
Thing *things = malloc(
    num_things * 16 + 40);

#define THING_HEADER_SIZE    16
#define THING_BODY_SIZE      40
#define THING_SIZE \
    THING_HEADER_SIZE + THING_BODY_SIZE
// Allocate memory for things.
Thing *things = malloc(
    num_things * 16 + 40);
AST macros
(a la Lisp)
Macros operate on the *parsed* program, and so are aware of its structure
(define-macro (THING_SIZE) `(+ 40 6))
(* 5 (THING_SIZE))
A function call, but made at compile time

```
(define-macro (THING_SIZE) `(+ 40 6))
(* 5 (THING_SIZE))
```
(define-macro (THING_SIZE) `(+ 40 6))

(* 5 (\(+\ 40\ 6\)))

Correct!
Lisp is conceptually beautiful

It's a language for processing lists

Programs themselves are lists
So what about Raku?
A much greater diversity of syntax

Requires a more complex model
But still...

A Raku macro is a function called at compile time

The arguments that are passed represent the code, not its result

They return value is also a representation of a piece of program
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while-defined @a.shift, -> $val {
    say $val;
}
Macro is called by the compiler after parsing its arguments
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while-defined @a.shift, -> $val {
    say $val;
}
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while-defined @a.shift,
    say $val;
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while-defined @a.shift, -> $val {
    say $val;
}
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while-defined @a.shift, -> $val { say $val; }
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{{ $cond }}}).defined {
            {{{ $body }}}($temp);
        }
    }
}

my @a = False, True, False;
while (my $temp = @a.shift).defined {
    (-> $val { say $val })( $temp );
}
macro while-defined($cond, $body) {
    quasi {
        while (my $temp = {{ $cond }}).defined {
            {{ $body }}($temp);
        }
    }
}

my @a = False, True, False;
while (my $temp = @a.shift).defined {
    (-> $val { say $val })(*$temp);
}
"It'll be cool to have AST macros!"

But wait, there's more...
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 is password;
    has Str $.verify-password is required;

    ...
}

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 is password;
    has Str $.verify-password is required;

    ...
}

Trait handlers run at compile time...
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;

    ...so if they can get the AST, then it's possible to compile the Raku regex into something for the HTML5 pattern attribute!
}
ECMA262Regex
Compiles JavaScript regex syntax into Raku regexes (used by JSON::Schema)

File::Ignore
Compiles .gitignore style patterns into Raku regexes

JSON::Mask, JSON::Path
Have interpreters written in Raku - but could be more efficient if we compiled them into Raku
Today, modules that want to compile into Raku look something like this...
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 $start ~ ']>';
method control-letter($/) {
    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 $start ~ ']'>;
}
method control-letter($/) {
    my $name = %control-char-to-unicode
    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 ~ ']'>;
}
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 ~ ']'>';
}
Could there be an injection attack?

Ewwwwwwwwwwww! Strings?!

How do we know it's well-formed?

Raku compiler has to spend time parsing too! 😞
Instead, they could produce a Raku AST

Either by building up an object graph, or by using quasi quoting
"It'll be cool to have various ways of accessing a Raku AST at compile time, as well as using it as a compilation target!"

But wait, there's still more...
What if we want to build...

A Raku linter?

A fancier Raku type checker?

Domain-specific compile-time checks?
Often, these tools must be built with their own parser and program model. Those can get out of sync with new language features, or end up having their own bugs.
Instead, they could consume a standard Raku AST

By serving as an extra compiler phase
"It'll be cool to be able to produce and consume Raku code in all kinds of scenarios using a standardized Raku AST!"

Surely there isn't more?
What if Rakudo's 10 year old frontend compiler architecture could be improved?

Because surely we've all learned a thing or ten in that time...
A standard Raku AST isn't just something we're going to add to the Rakudo compiler.
RakuAST will be found at the very heart of Rakudo.
"It'll be cool!"

But how will we get there?
The design of RakuAST, and a compiler based around it
The Rakudo compiler frontend today

Raku Source

Parser

Actions

World

QAST Tree
The Rakudo compiler frontend today

Raku Source

Parser

Actions

World

QAST Tree

Relatively low-level AST, good for VM abstraction
The Rakudo compiler frontend today

- **Raku Source**
  - **Parser**
  - **QAST Tree**
  - **Behavior**
    - **Actions**
    - **World**
    - **Declarations**
Large components that know the whole language

- Raku Source: 5,800 LoC
- Parser
- Actions: 11,200 LoC
- World: 5,600 LoC
- QAST Tree: 3,200 LoC
- optimizer too!
Error reporting spread throughout them
Error reporting spread throughout them

Some stuff reported from here isn't just about syntax...
RakuAST node

= 

The expert on a language construct

(Excluding its syntax)
A RakuAST node knows about a language construct's...

Semantics (code-gen)

Declarations and meta-objects

Symbol usage (explicit, implicit)

CHECK time (semantic errors)

Sink context handling

Optimization
The Rakudo compiler frontend with RakuAST

Raku Source

Only reports syntax errors

Parser

Far simpler; just maps parse tree to RakuAST

Raku AST

Actions

Code-gen, checks, declarations, etc. handled in AST nodes
What about the ECMA262Regex module?
What about the ECMA262Regex module?
What about the ECMA262Regex module?

Regex source → Builds RakuAST nodes describing a regex

Parser → Raku grammar for JS regexes

Raku AST → We then compile from the RakuAST - no Raku source is generated!
RakuAST nodes model language constructs

For example, there's a node for a parenthesized expression, even if these are usually free of semantics.
RakuAST nodes should work just like any other Raku object
Must fit within the type system
So we can multi-dispatch over them, destructuring them using signatures, and so forth

Must be introspectable
So we can explore them in the REPL, have auto-complete on them in the IDE, etc.

Must be easy to construct
Just create them with .new. No context objects or compiler state required.
Use types to encode valid syntactic structure

(Also, macros will be able to use RakuAST types on parameters - which can map back into syntax errors.)
Use roles to extract common features and/or interfaces of AST nodes

RakuAST::Statement
RakuAST::Term
RakuAST::LexicalScope
RakuAST::Lookup
RakuAST::Sinkable
The progress on implementing RakuAST so far
I'm currently working on making RakuAST a reality

Supported by a grant from The Perl Foundation

♥
A slight problem:

We want RakuAST nodes to work as if they are implemented in Raku

But we need RakuAST in order to compile Raku code!
Raku standard library

Metamodel

Bootstrap

CORE.setting
Raku standard library

Metamodel

Bootstrap

CORE.setting

Metaclasses implemented in NQP
Raku standard library

Metamodel

Bootstrap

CORE.setting

NQP code using MOP to piece together basic Raku types
Raku standard library

Metamodel

Bootstrap

CORE.setting

Loads of built-ins, implemented in Raku
The bootstrap already exists to put together just enough that we can write the rest in Raku...

...so it makes sense to have the RakuAST nodes pieced together there too.
But it's *really, really* tedious to manually instantiate all of the meta-objects and piece them all together!
But it's *really, really* tedious to manually instantiate all of the meta-objects and piece them all together!

Thankfully, I'm a compiler writer, so I just wrote a little compiler to do that for me! 😊
Current status

✓ Over 100 node types (some abstract) implemented

✓ Around 200 tests covering construction and EVAL from RakuAST nodes

✓ New RakuAST-based compiler frontend, enabled by an environment variable, passes half the sanity tests
Following progress or trying it out

Find the source
In the rakuast branch of the Rakudo repository

Try it out
Using RAKUDO_RAKUAST=1 in the environment

Follow grant reports
On The Perl Foundation blog
The impact
of RakuAST on Raku users
Compatibility goal

The *vast majority* of Raku users won't notice any changes in their program's behavior when they upgrade Rakudo to a version based around RakuAST
How?

Passing the specification tests
Should not show any regressions

Checking its impact with Blin
Runs the tests of all ecosystem modules; only those dabbling in compiler internals should be affected

Ensuring updates are available
For the few widely used modules that depend on compiler internals
Naively, an "extra layer" could be expected to lead to a slowdown.

However, I'm cautiously optimistic we can come out ahead.
Why might it be faster?

More straightforward compilation
Thanks to a better program representation

Potential for better optimization
The static optimizer today has become challenging to extend; RakuAST should offer a cleaner approach

Potential for parallelism
Some AST processing may be able to happen while we parse the rest of the compilation unit
What next?

RakuAST nodes for all the language
Early autumn 2020

RakuAST-based compiler as default
Late autumn 2020

Macros
Christmas (2020 😊)

Language release including RakuAST
Spring 2021
Thank you!

Questions?

@ jonathan@edument.cz
W jnthn.net
🔗 jnthnwrthngtn
🔗 jnthn