Inside A Compiler

Jonathan Worthington

Compiler Hacker

Rakudo Perl 6 core developer

Compiler toolkit core developer

Focus on the object model, type system, multiple dispatch and backends

Traveller

I love to travel. Especially to open source events. ☺

Beer Drinker

Beer is tasty.

I drink it.

\☺/

Perl 6

Mutable grammar means we really need to parse Perl 6 using a Perl 6 grammar → grammar engine

Pluggable object model → need meta-object programming support

Runs everywhere → need to generate code for many backends

...

What we need in Perl 6 is pretty much what you need to write compilers anyway

From the start of Rakudo, those parts have been factored out

Thus, we have an actively developed, growing compiler toolkit for others to use, whether they care about Perl 6 or not ☺

What Compilers Do

Take input in one language…

…and produce output in another, lower level language.

(Hopefully, the output has some semantic relationship with the input ☺)

Trees

Working with code as text is slow and difficult, so compilers tend to prefer to work with trees

Source Code

Parsing

Parse Tree

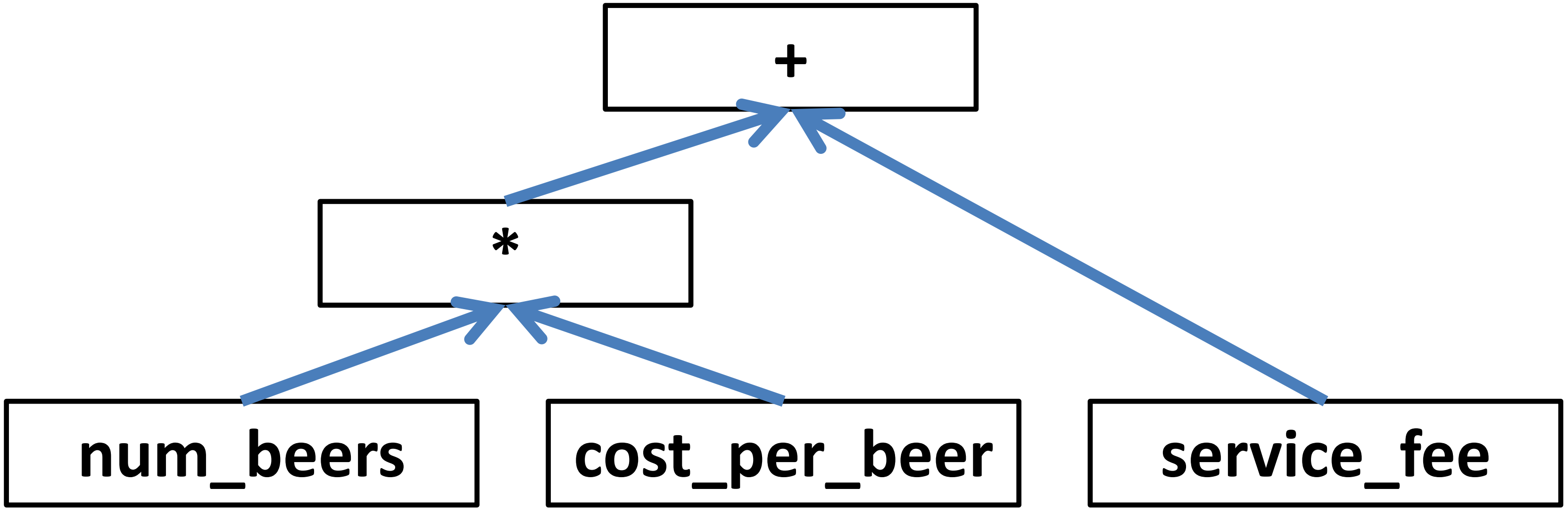
(more trees built here)

Code Generation

Code In Target Language

Tree Example

```
num_beers * cost_per_beer + service_fee
```



Parse Tree

A tree representation of the source language; closely related to it

Abstract Syntax Tree

Represents the semantics of the program; a step away from the actual syntax

Backend Tree

Tree representation that is close to the target language; final step before doing code generation

Traditional View

Parse tree

→

AST

→

Backend tree

Dynamic Language Reality

Need to do bits of runtime during compile time

(Parrot) Compiler Toolkit

Created as part of the Rakudo Perl 6 compiler project, but not at all Perl 6 specific

Grammar engine
Set of AST nodes (PAST)
AST → Intermediate Language
Compiler Infrastructure

NQP

“Not Quite Perl (6)”

Very small subset of the Perl 6 language that’s ideal for writing compilers, especially parse tree to AST mapping

NQP compiler is implemented in NQP (bootstrapped)

To create a language...

Write a grammar for it

Write “action methods” to map parse tree to AST

Write built-in types/functions

Optionally, write meta-objects to implement OO features

TinyLang

We’re going to build a very little language with...

String literals
A writeline built-in
String concatenation
Variables and binding
Classes with methods

TinyLang

Aim: run this little program

```
var dog_noise = "woof";

a Dog {
    can bark {
        writeline(dog_noise ~ " " ~ dog_noise)
    }
};

var fido = new(Dog);
fido.bark;

var puppy = new(Dog);
dog_noise = "yap";
puppy.bark;
```

Getting Started

Stub in grammar, actions and compiler classes, inheriting from HLL base classes

```
use HLL;

class TinyLang::Grammar is HLL::Grammar {
}

class TinyLang::Actions is HLL::Actions {
}

class TinyLang::Compiler is HLL::Compiler {
}
```

Getting Started

Create a MAIN sub
(the compiler entry point)

```
sub MAIN(*@ARGS) {
    # Create and configure compiler object.
    my $tlcomp := TinyLang::Compiler.new();
    $tlcomp.language('tinylang');
    $tlcomp.parsegrammar(TinyLang::Grammar);
    $tlcomp.parseactions(TinyLang::Actions);

    # Enter the compiler.
    $tlcomp.command_line(@ARGS,
                         :encoding('utf8'));
}
```

TOP

The grammar rule named TOP is the one that is called first when starting to parse

For now, it'll parse a list of terms separated by ";"

```
rule TOP { <term> ** ';' }
```

Syntactic Categories

Programs often contain many items of the same “type”, e.g.

Infix operators
Terms
Literal values

We make these explicit in our grammar using “proto regexes”

Terms

For now, our terms will either be values or a call to a built-in

```
proto token term { <...> }
token term:sym<call> {
    <ident> '(' <arglist> ')'
}
token term:sym<value> { <value> }
```

Argument list is terms separated by commas

```
rule arglist { <term> ** ',' }
```

Values

Most languages have many types of literal value (strings, integers, floating point numbers)

For TinyLang, we just do strings

```
proto token value { <...> }
token value:sym<string> { <?["]> <quote_EXPR> }
```

(quote_EXPR, inherited from HLL::Grammar, is an extensible quote parser)

Parsing Works!

Run the compiler with --target=parse

```
> "foo"
"parse" => PMC 'Regex;Match' => "\"foo\"\n" @ 0 {
  <term> => ResizablePMCArray (size:1) [
    PMC 'Regex;Match' => "\"foo\"" @ 0 {
      <value> => PMC 'Regex;Match' => "\"foo\"" @ 0 {
        <quote_EXPR> => PMC 'Regex;Match' => "\"foo\"" @ 0 {
          ...
        }
      }
    }
  ]
}
```

AST

Every time a parsing rule completes, we run an action method from the actions class

Method produces an AST node and associates it with the grammar match object

Parse top down, build AST bottom up

Values

The action method is easy – quote_EXPR's action method in HLL::Actions does the work for us, so just use whatever it made

```
method value:sym<string>($/) {
    make $<quote_EXPR>.ast;
}
```

Argument List

Loop over all of the terms and get their AST, and push each one into a container PAST::Op node

```
method arglist($/) {
    my $args := PAST::Op.new();
    for $<term> {
        $args.push($_.ast);
    }
    make $args;
}
```

Terms

Value – use value’s AST

```
method term:sym<value>($/) {
    make $<value>.ast;
}
```

Built-in call: twiddle the PAST::Op node made by arglist

```
method term:sym<call>($/) {
    my $call := $<arglist>.ast;
    $call.pasttype('call');
    $call.name(~$<ident>);
    make $call;
}
```

TOP

Need to run each term in order

Put the AST for each of them into a PAST Statements node

```
method TOP($/) {
    my $program := PAST::Stmts.new();
    for $<term> {
        $program.push($_.ast);
    }
    make $program;
}
```

Built-in

We just use the NQP say function to implement our writeline built-in

```
our sub writeline($message) {
    say($message);
}
```

It works!

The compiler toolkit knows how to map PAST nodes to intermediate code for the VM

We now have a working compiler

```
> writeline("hello"); writeline("Taipei");
hello
Taipei
```

Operators

So far our parsing has been roughly recursive descent

Not good for parsing operators at various precedence levels

HLL::Grammar provides EXPR, a configurable operator parser

Concatenation

Need to configure the OPP...

```
INIT {
    # Precedence levels (just one so far).
    TinyLang::Grammar.O(':prec<z=>,
        :assoc<left>',  '%concatenation');
}
```

...and add a rule for parsing ~

```
token infix:sym<~> {
    <sym> <O('%concatenation')>
}
```

Concatenation

For the action method, we make a PAST::Op node that will call the VM's concat op

```
method infix:sym<~>($/) {
    make PAST::Op.new( :pirop('concat Sss') );
}
```

term → EXPR

Finally, anywhere we used to parse a term, we now switch to parsing an expression (EXPR will call term for us as needed)

```
rule TOP { <EXPR> ** ';' }
```

```
rule arglist { <EXPR> ** ',' }
```

(We update action methods to match.)

It works!

Now have working concatenation

```
> writeline("Taipei " ~ "101”)
Taipei 101
```

Can view VM’s intermediate code with --target=pir

```
concat $S296, "Taipei ", "101"
"writeline"($S296)
```

Binding

Update operator precedence parser to also know about =

```
INIT {
    # Precedence levels.
    TinyLang::Grammar.O(':prec<z=>,
        :assoc<left>',  '%concatenation');
    TinyLang::Grammar.O(':prec<y=>,
        :assoc<right>', '%assignment');
}
token infix:sym<~> { <sym> <O('%concatenation')> }
token infix:sym<=> { <sym> <O('%assignment')> }
```

Binding

Add an action method; PAST compiler knows how to compile a bind operation, so just use that

```
method infix:sym<=>($/) {
    make PAST::Op.new( :pasttype('bind') );
}
```

Symbol Tables

We tend to have variables with different scopes (lexical, package, attribute, etc.)

Need to have a symbol table to map names to scopes

PAST::Block provides this capability – but we need to use it!

Block Refactor

Action methods need a block stack to know the current block

```
my @BLOCK;
```

And a method to create a PAST::Block node when we are starting a new block (e.g. scope)

```
method newblock($/) {
    @BLOCK.unshift(PAST::Block.new());
}
```

Block Refactor

Update grammar and actions so that we wrap the program in a block, not a statements node

```
rule TOP { <.newblock> <EXPR> ** ';' }
token newblock { <?> }
```

```
method TOP($/) {
    my $program := @BLOCK.shift;
    for $<EXPR> {
        $program.push($_.ast);
    }
    make $program;
}
```

Variable Declarations

Add variable_declaration grammar rule and action method

```
rule variable_declaration { 'var' <ident> }
```

```
method variable_declaration($/) {
    my $name := ~$<ident>;
    @BLOCK[0].symbol($name, :scope('lexical'));
    make PAST:Var.new( :name($name), :isdecl(1) );
}
```

Variable References

Add variable grammar rule and action method

```
token variable { <ident> }
```

```
method variable($/) {
    make PAST::Var.new( :name(~$<ident>) );
}
```

Terms Update

Need to add variable declarations and references as terms

```
token term:sym<var> { <variable> <![(]> }
token term:sym<decl> { <variable_declaration> }
```

```
method term:sym<var>($/) {
    make $<variable>.ast;
}
method term:sym<decl>($/) {
    make $<variable_declaration>.ast;
}
```

It works!

Note how we can now start to naturally mix the language features that we have implemented ☺

```
> var city = "Taipei";
  var emo = "love";
  writeline("I " ~ emo ~ " " ~ city ~ "!”);
I love Taipei!
```

Our Progress

We now implemented all of the non-OO features of TinyLang

Excluding comments and blank lines, we have only 88 lines of code!

\☺/

6model

Previous generations of the compiler toolkit left OO to the compiler writer and VM

Latest NQP includes "6model" OO core, which offers:

Meta-object Programming
Gradual Typing Support
Representation Polymorphism

Meta-objects

Meta-objects have methods that respond to various "events" that occur as we compile a package

Implementing classes

=

Writing methods

Example

```
a Dog {
    can bark {
        writeline("woof")
    }
};
```

Example

```
a Dog {
     can bark {
          writeline("woof")
     }
};
```

new_type

Example

```
a Dog {
    can bark {
        writeline("woof")
    }
};
```

add_method

Example

```
a Dog {
    can bark {
        writeline("woof")
    }
};
```

compose

TinyLang Class

new_type, compose and name

```
class TinyLangClass {
    has $!name;
    method new_type(:$name = '<anon>') {
        my $metaclass := self.new(:name($name));
        nqp::repr_type_object_for($metaclass,
            'HashAttrStore');
    }
    method compose($obj) {
        return $obj;
    }
    method name($obj) {
        return $!name;
    }
}
```

Statements

We'll do a little refactor to distinguish terms and statements

```
rule TOP { <.newblock> <statement> ** ';' }
token newblock { <?> }

proto token statement { <...> }
token statement:sym<EXPR> { <EXPR> }
```

(Action methods updated to match the changes.)

Class Parsing

For now, we'll just parse "a", the name of the class and the block

```
rule statement:sym<class> {
    'a' <ident>
    '{'
    '}'
}
```

Class Actions

We generate code that makes the appropriate method calls on our meta-object.

```
...
# Does TinyLangClass.new_type(:name($name))
PAST::Op.new(
     :pasttype('callmethod'), :name('new_type'),
     PAST::Var.new( :name('TinyLangClass'),
                    :scope('package') ),
     PAST::Val.new( :value(~$name),
                    :named('name') )
)
...
```

Built-ins

We'll add "new" and "typeof" built-ins

```
our sub new($type) {
    return nqp::repr_instance_of ($type)
}
our sub typeof($obj) {
    return $obj.HOW.name($obj)
}
```

(.HOW is an NQP macro that gets an object's meta-object.)

It works!

We can now declare a class, create an instance of it and get its name using typeof

```
> a Beer { };
  var budweiser = new(Beer);
  writeline(typeof(budweiser));
Beer
```

Methods

First, we update our meta-object to support methods

```
has %!methods;

method add_method($obj, $name, $code) {
    %!methods{$name} := $code;
}

method find_method($obj, $name) {
    return %!methods{$name}
}
```

Method Parsing

Note that we call <.newblock> so that variables declared inside the method are local to it

```
proto token class_body_item { <...> }
rule class_body_item:sym<method> {
    'can' <ident>
    <.newblock>
    '{' <statement> ** ';' '}'
}
```

Action method calls .add_method on the meta-object.

Example

```
a Cow { can moo { writeline("moo") } }
```

Compiles to:

```
get_global $P802, "TinyLangClass”
$P803 = $P802."new_type"("Cow" :named("name"))
.lex "Cow", $P803
find_lex $P804, "Cow”
get_how $P805, $P804
.const 'Sub' $P808 = "213_1301185815.657”
capture_lex $P808
$P805."add_method"($P804, "moo", $P808)
find_lex $P810, "Cow”
get_how $P811, $P810
$P811."compose"($P810)
```

And we're done!

Our original example runs:

```
var dog_noise = "woof‟;

a Dog {
    can bark {
        writeline(dog_noise ~ " " ~ dog_noise)
    }
};

var fido = new(Dog);
fido.bark;

var puppy = new(Dog);
dog_noise = "yap‟;
puppy.bark;
```

```
woof woof
yap yap
```

181 Lines

For grammar, actions, meta-object and setup code (excluding comments and blank lines)

Grammar: 40 lines
Actions: 102 lines
Meta-object: 20 lines

And plenty to scope to expand without significant refactors

NQP

Just like our compiler, the NQP compiler has...

Grammar class
Actions class
Meta-objects
Glue

Written in NQP; can compile itself

Current Focus

Getting the gradual typing related features of 6model accessible to compiler authors (we need it for Rakudo Perl 6)

Updating documentation (possible to write compilers totally in NQP now, and need to document 6model much more)

The Future

Make the compiler toolkit able to generate code for extra VMs

Currently, there is some early work to support 6model and PAST compilation on both the CLR and the JVM

Terrifying Demo of the Future

We can't run TinyLang on the CLR yet – that will need a pretty full port of the NQP language.

However, we can – with a few minor tweaks – cross-compile to the .Net CLR. ☺

My Goals

Make exploring language ideas or making DSLs quick and easy

We'll be able to write one compiler that "just works" on many VM backends

Create an awesome Perl 6 compiler ☺

Thank You

Questions?