Before we begin, a correction...
A little bit of background...

Macros progress report: after a long break

I am going to make no pretense at covering everything here. My goal with this post is simply to bring us largely up-to-date with the current ideas about macros in Perl 6 and possible directions we’re taking. A post about this has been sorely missing for a while now.

In order not to retread old ground, this post assumes that you have read day 23’s post about macros in the 2012 Perl 6 Advent calendar. That post remains a very good high-level summary of all the work so far.
A COMPLETE LANGUAGE TUTORIAL?

AIN'T NOBODY GOT TIME FOR THAT!
Freely borrowing features

007 looks like a mix of Perl, Python, and JavaScript.

<table>
<thead>
<tr>
<th>feature</th>
<th>Perl 6</th>
<th>007</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>braces</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>user-defined operators</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>variable declarations</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>macros</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>implicit typecasts</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>sigils</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>multis</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>implicit returns</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
(live demo)
Everything I've done leads up to this
Secret sauce: the AST format

Homoiconic? I *think* so...

```bash
$ cat examples/hello-world.007
say("Hello, world!");
```

```xml
<!DOCTYPE html>
<html lang="en" data-ember-extension="1">
  <head>…</head>
  …
  <body>
    …
    <a href="https://github.com/masak/007">…</a>
    …
    <h2 id="overview">Overview</h2>
    …
    <p>…</p>
    …
    <table class="table table-bordered">…</table>
    …
    <h2 id="values">Values</h2>
    …
    <p>A small number of values in 007 can be expressed using li</p>
    …
    <pre>…</pre>
    …
    <p>Only double quotes are allowed. Strings don't have any fo</p>
    …
    <h2 id="expressions">Expressions</h2>
    …
    <pre>…</pre>
    …
    Q::CompUnit Q::Block {
      parameterlist: Q::ParameterList [],
      statementlist: Q::StatementList [Q::Statement::Expr Q::Postfix::Call {
        identifier: Q::Identifier "postfix:<>",
        operand: Q::Identifier "say",
        argumentlist: Q::ArgumentList [Q::Literal::Str "Hello, world!"]]}
    }
```
Well-kept secret: this is easy

```perl
rule statement:if {
    if <xblock>
    [ else
    [ | <else=block>
    | <else=statement:if>
    ]
    ]?
}

method statement:if ($/) {
    my %parameters = <$xblock>.ast;
    %parameters<else> = <$else> :exists
    ?? <$else>.ast
    !! Val::None.new;

    make Q::Statement::If.new(|%parameters);
}

class Q::Statement::If does Q::Statement {
    has $.expr;
    has $.block;
    has $.else = Val::None.new;

    method attribute-order { <expr block else> }

    method run($runtime) {
        my $expr = $.expr.eval($runtime);
        if $expr.truthy {
```
Easy: index chains and assignment

```
```
`, and an array is never equal to an int, not even the length of the array.

- The only thing that can be assigned to is variables. Arrays are immutable values, and you can't assign to `ar[3]`, for example.
  - `ar[3] = "hammer";  # error; can't touch this

You can assign to individual variables, like `name`, or long strings of postfix operators, like `employee[n - 1].boss.name`. There's no autovivification like in Perl &mdash; in the previous example, `employee[n - 1].boss` needs to already exist (though its `.name` +property doesn't need to exist).

Operands don't need to be simple values. Arbitrarily large expressions can be built. Parentheses can be used to explicitly show evaluation.
Some less easy things

static
dynamic
Macros: easy

```perl
sub format(fmt, args) {
    sub replaceAll(input, output, transform) {
        my $openBracePos = input.index("{");
        if ($openBracePos == -1) {
            return output ~ input;
        }
        my $closeBracePos = input.suffix($openBracePos).index("}");
        if ($closeBracePos == -1) {
            return output ~ input;
        }
        return replaceAll(
            input.suffix($openBracePos + $closeBracePos + 1),
            output ~ input.prefix($openBracePos) ~ transform(input.substr($openBracePos + 1, $closeBracePos - 1)),
            transform)
        ;
    }
    return replaceAll(fmt, ",", sub transform(arg) { return args[int(arg)]; });
}

say( format("{0}{1}{0}", ["abra", "cad"] ) );  # abracadabra
say( format("foo{0}bar", ["{1}" ] ) );            # foo{1}bar (to demonstrate that \{1\} works in format arguments)
```
if fmt ~ Q::Literal::Str && args ~ Q::Term::Array {
  my highestUsedIndex = findHighestIndex(fmt.value);
  my argCount = args.elements.elems();
  if argCount <= highestUsedIndex {
    die "Highest index was " ~ str(highestUsedIndex)
    ~ " but got only " ~ str(argCount) ~ " arguments.";
  }
}
Closures

(live demo)
Hygiene

```perl
{
    my $program = q:to=./;
    macro foo(expr) {
        my x = "oh noes";
        return quasi {
            say(${expr});
        }
    }

    my x = "yay";
    foo(x);
}

outputs $program, "yay\n", "macro arguments also carry their original environment";
}

This is the case of a variable. Also true for subs, operators, macros...
```
Solution: identifiers with context

class Q::Identifier does Q::Term {
    has Val::Str $.name;
    has $.frame = Val::None.new;

    method attribute-order { <name> }

    method eval($runtime) {
        return $runtime.get-var(
            $.name.value,
            $.frame ~~ Val::None ?? $runtime.current-frame !! $.frame
        );
    }

    method put-value($value, $runtime) {
        $runtime.put-var(self, $value);
    }
}

A variable knows the context in which it was created.
Testing

```
my $program = q:to/.;  
sub A(k, x1, x2, x3, x4, x5) {
    if k <= 0 {
        return x4() + x5();
    } else {
        sub B() {
            k = k - 1;
            return A(k, B, x1, x2, x3, x4);
        }
        return B();
    }
}

sub x1() { return 1 }
sub x2() { return -1 }
sub x3() { return -1 }
sub x4() { return 1 }
sub x5() { return 0 }

say(A(10, x1, x2, x3, x4, x5)) .
```

outputs $program, "-67\n", "007 is a man-compiler";

All tests successful.
Files=37, Tests=418, 106 wallclock secs ( 0.17 usr  0. Result: PASS
Tests I didn't expect

my @lines-ending-with-ws;
for find(".", "/.pm" $/) -> $file {
    for $file.IO.lines.kv -> $i, $line {
        if $line ~~ /\h $/
            { push @lines-ending-with-ws,
                "$file {$i + 1}: " ~
                $line.subst(/\h* $/, - > $/ { chr(0x2620) x $/.chars });
            }
    }
}

is @lines-ending-with-ws.join("\n"). "no whitespace at the end of a line in .pm files":

my @classes = flat
    qx[perl6 -ne 'say ~$0 if /^class \h+ ("Q::: " \S+)/' lib/_007/Q.pm].lines,
    qx[perl6 -ne 'say ~$0 if /^class \h+ ("Val::: " \S+)/' lib/_007/Val.pm].lines;

my @builtins = qx!perl6 -ne 'say ~$0 if /^\h+ ([Val|Q] ":::" <-[,]>+ "", \h* $/' lib/_007/Runtime/Builtins.pm!.lines;

{ my $missing-classes = (@builtins (-) @classes).keys.map({ "- "$ } ).join("\n");
    is $missing-classes, "", "all built-in types are also classes";
}

{ my $missing-builtins = (@classes (-) @builtins).keys.map({ "- "$ } ).join("\n");
    is $missing-builtins, "", "all classes are also built-in types";"}
Make the linter able to reason about macro usages #64

masak commented on Oct 28, 2015

This is a funny one. I had already coded up the linter code for "defined a sub, then didn't use it". I figured I could just copy the same test code into a separate one for macros, and change things around a little, and then implement it for macros.

But no. I can't do that. See, macro calls aren't there when the linter gets to the Qtree.

😂😂😂
Future: syntax macros

Syntax macros

Syntax macros are defined at statement level and essentially introduce a new type of statement. Let pretending keyword, a block form of temp:

```plaintext
class Q::Statement::Pretending is Q {
    has Q::Expr $.expr;
    has Q::Block $.block;
}

macro statement_control:pretending(Q::Expr $expr, Q::Block $block)
    is parsed(rule { <sym> <EXPR> <pblock> }) {
    # ...code to check that $expr is of the form `{{${var}}} = {{{$value}}}`
    elided...
    return quasi {
    temp {{{${var}}} = {{{$value}}};
    {{{${block}}}); # handling of >0 params elided
}
}```
Future: visitor macros

Visitor macros

It's possible we shouldn't call these "macros" at all. But I don't have a better name now.

By way of example, let's say you want to write a macro that makes code such as

The visitor macro might look something like this:

```plaintext
MATCH (Q::If (Q::Infix::NumEq :$expr (Q::Enum :$rhs where *.value eq "Bool::True"))) { 
  die "useless use of `== True`";
}
```

Every single bit of the above is conjectural syntax.
Thank you!

Questions?

https://github.com/masak/007

Remember, pull requests are love <3

https://www.flickr.com/photos/brenda-starr/3727579805