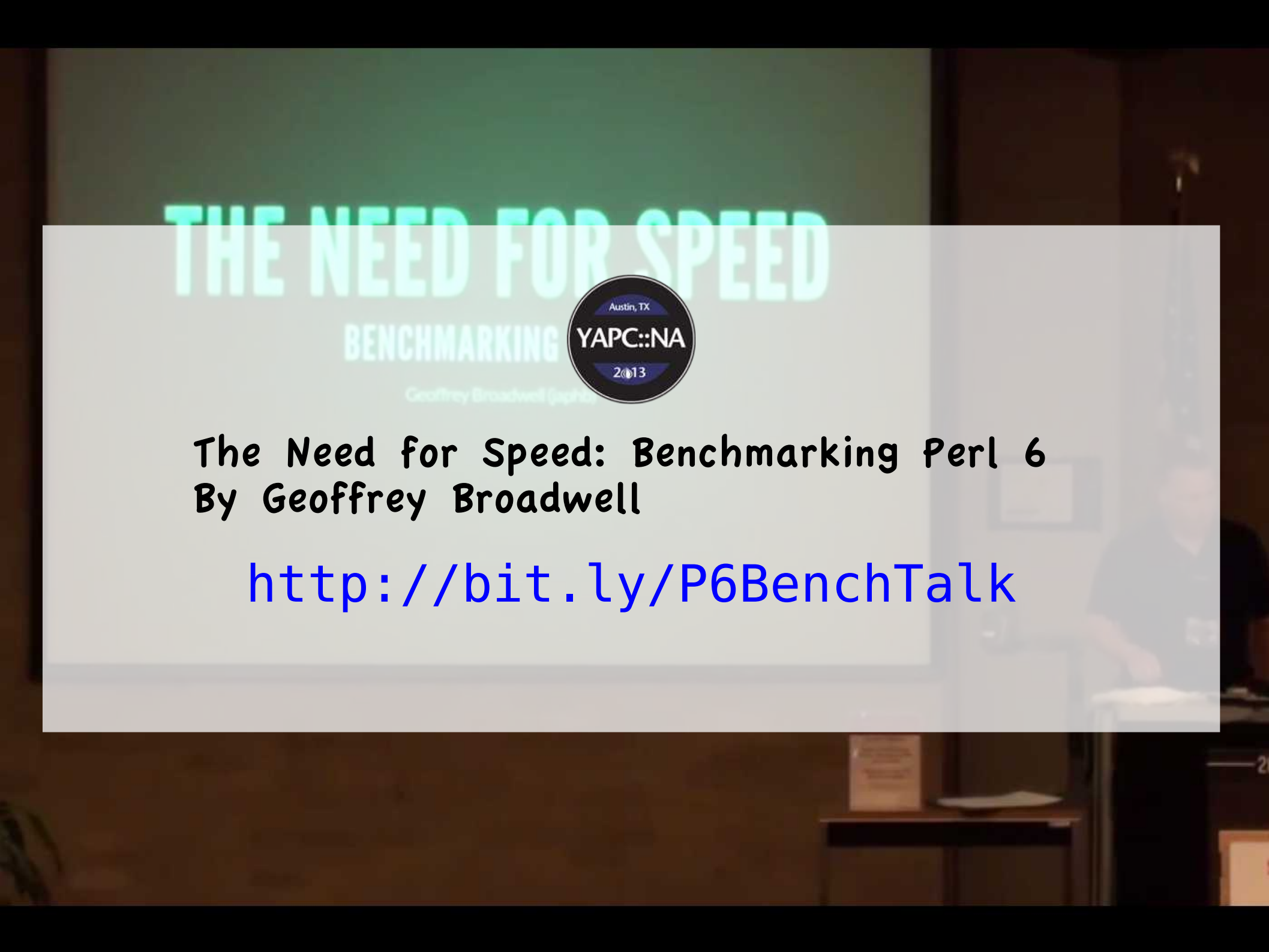
Benchmarking Perl 6

How Ready for Prime Time Is It?



The Need for Speed: Benchmarking Perl 6

By Geoffrey Broadwell

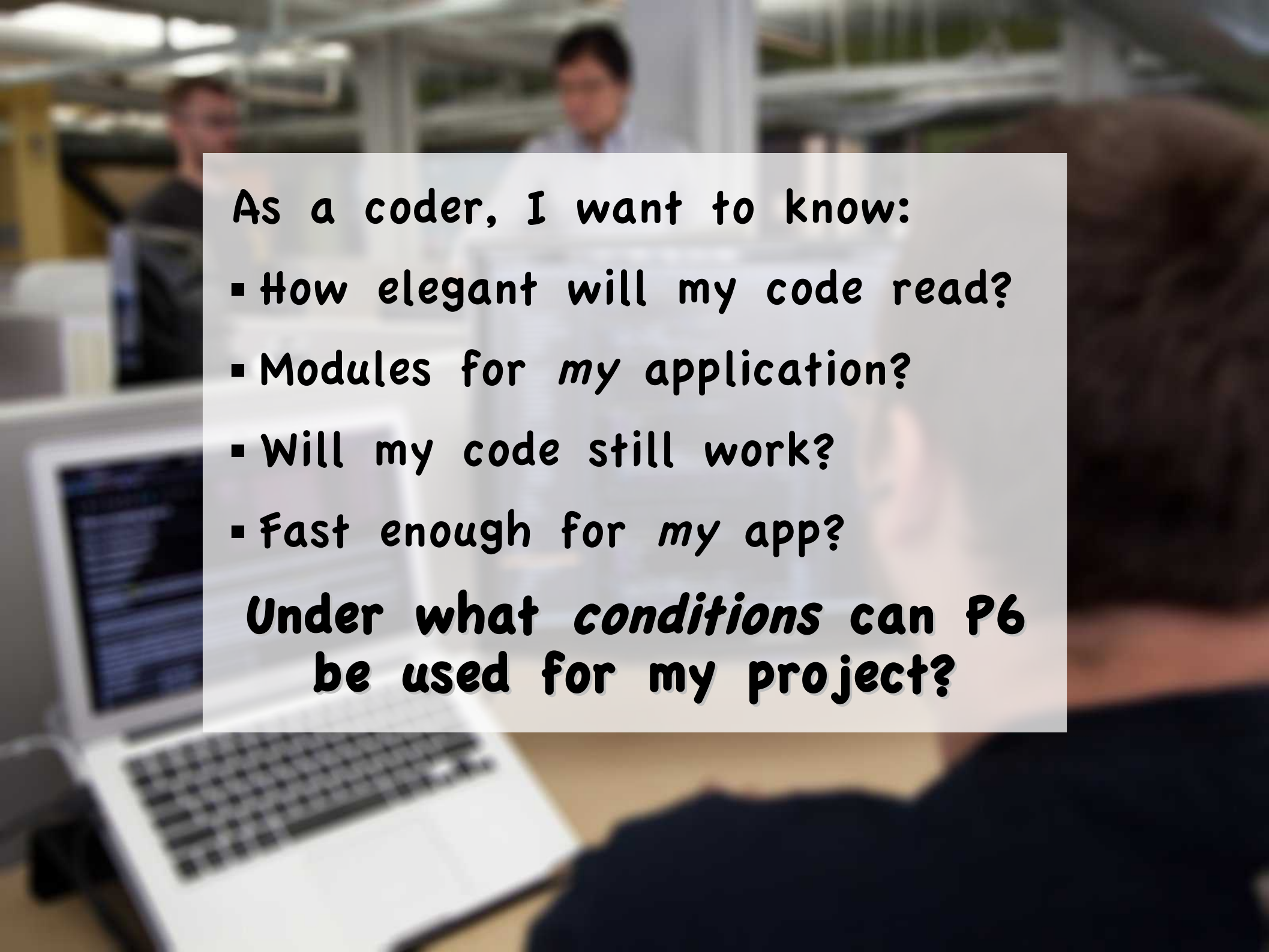
http://bit.ly/P6BenchTalk

What remains?

- Features
- Concurrency
- CPAN
- Speed

“Production” ~~ Quality

No one really knows what it is.



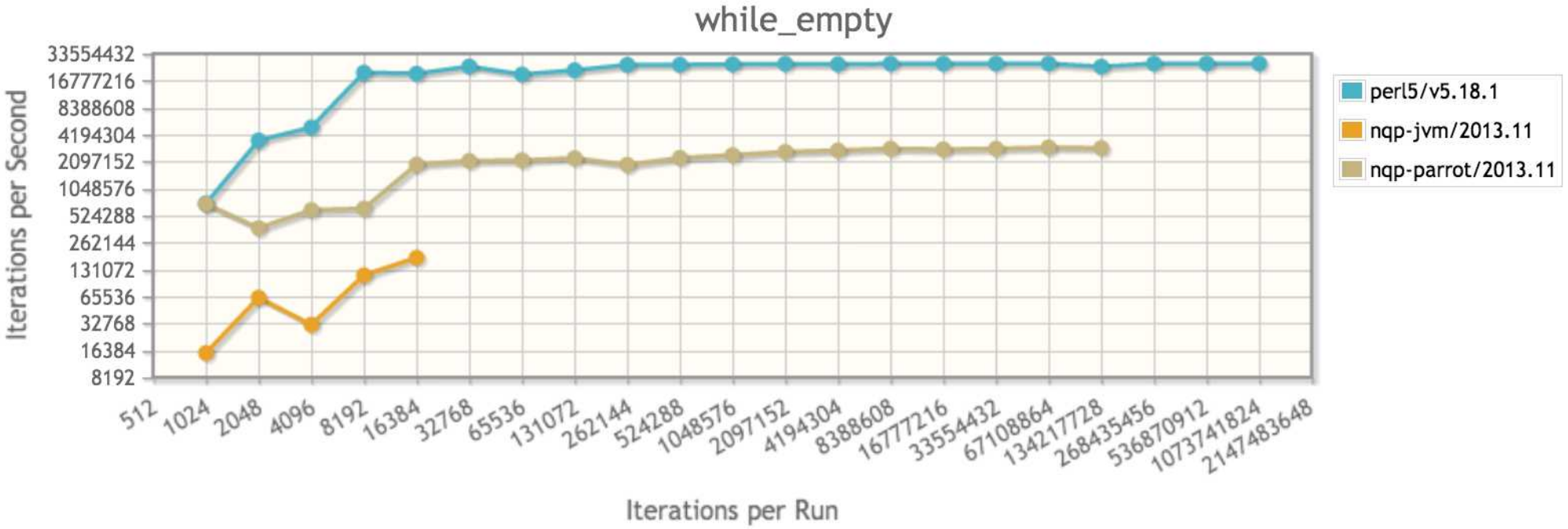
As a coder, I want to know:

- How elegant will my code read?
- Modules for *my* application?
- Will my code still work?
- Fast enough for *my* app?

Under what *conditions* can P6 be used for my project?

perl6-bench

```
$ ./bench build \
  perl5/v5.18.1 rakudo-parrot/2013.11

$ ./bench --tests-tagged=while test

$ ./bench --format=html_plot compare
```

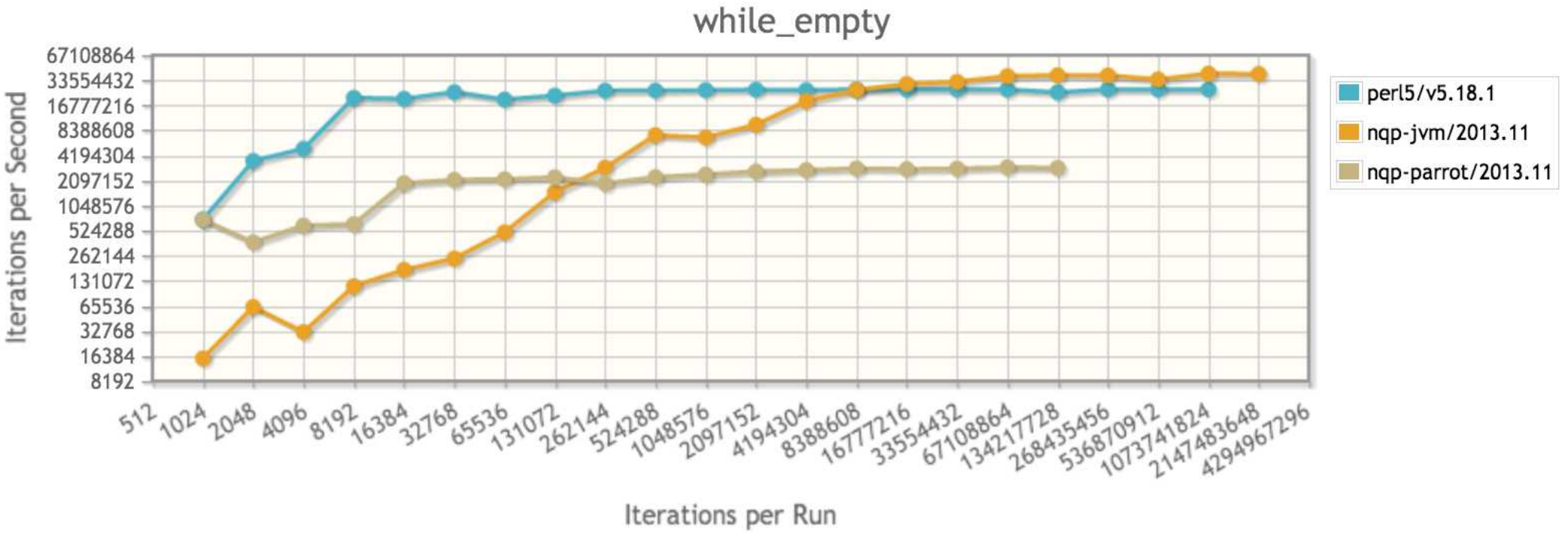


Perl 5

```
my $i = 0; while (++$i <= SCALE) { }
```

NQP

```
my $i := 0; while ($i := $i + 1) <= SCALE { }
```

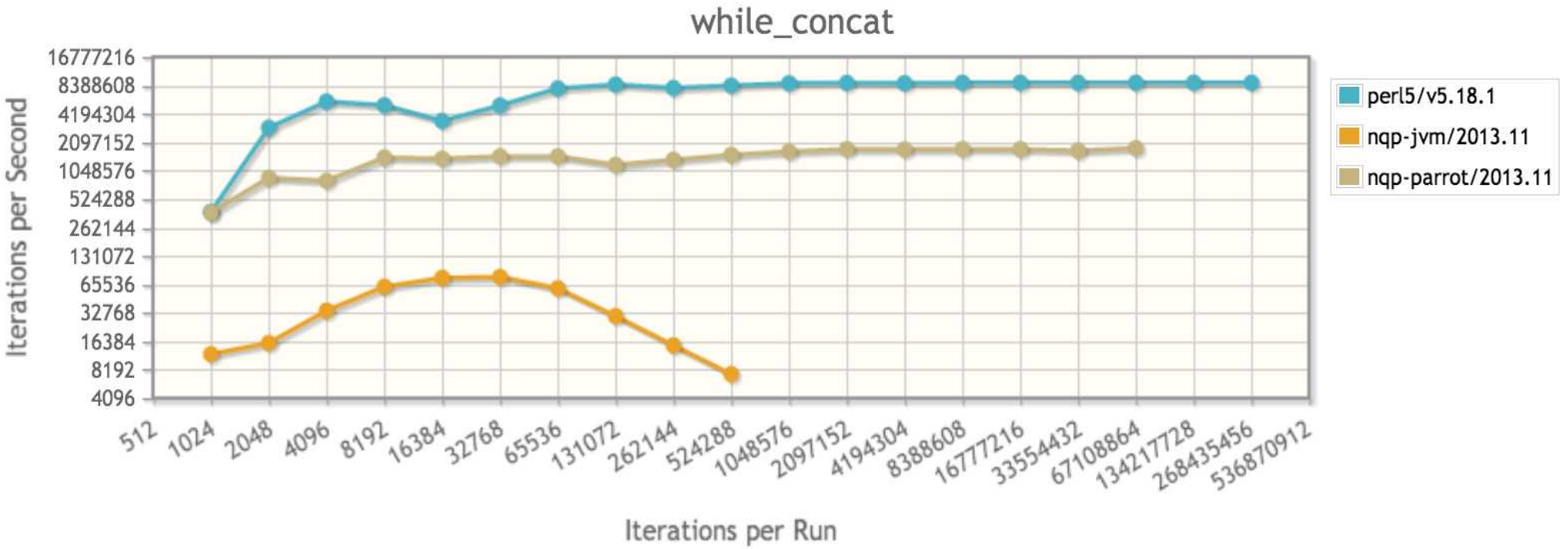


Perl 5

```
my $i = 0; while (++$i <= SCALE) { }
```

NQP

```
my $i := 0; while ($i := $i + 1) <= SCALE { }
```

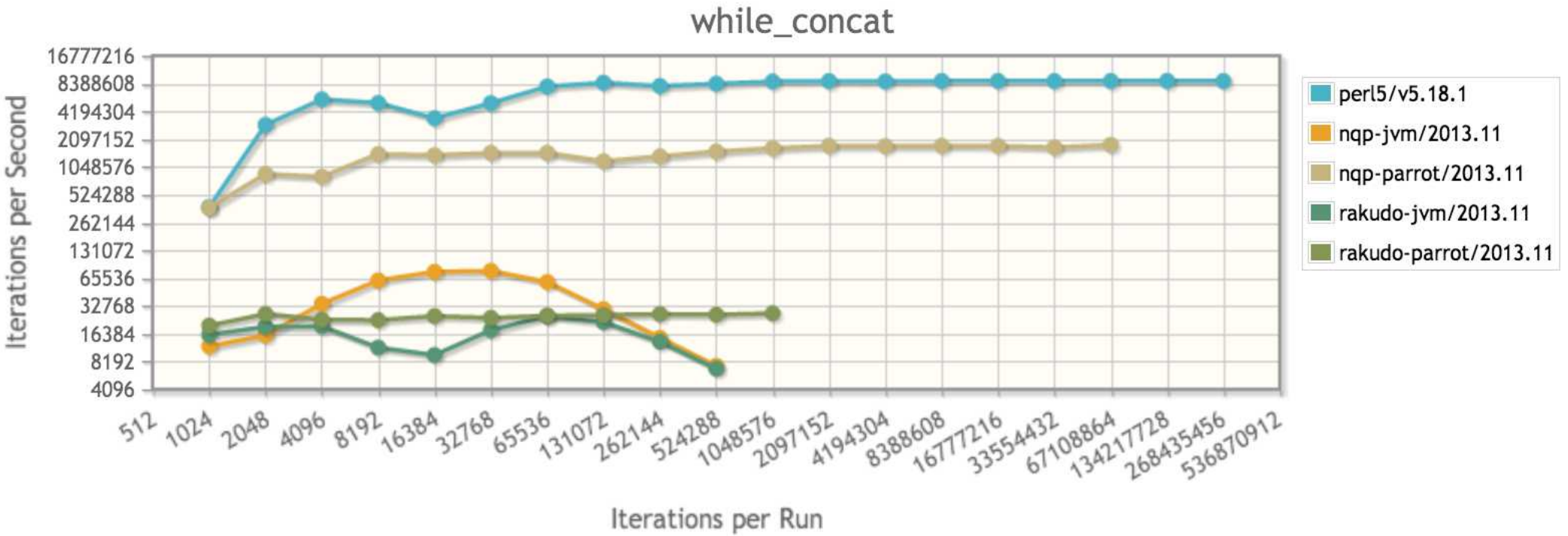


Perl 5

```
my $s = ""; my $i = 0; while (++$i <= SCALE) { $s .= "x" }
```

NQP

```
my $s := ""; my $i := 0;
while ($i := $i + 1) <= SCALE { $s := $s ~ "x" }
```



Perl 5

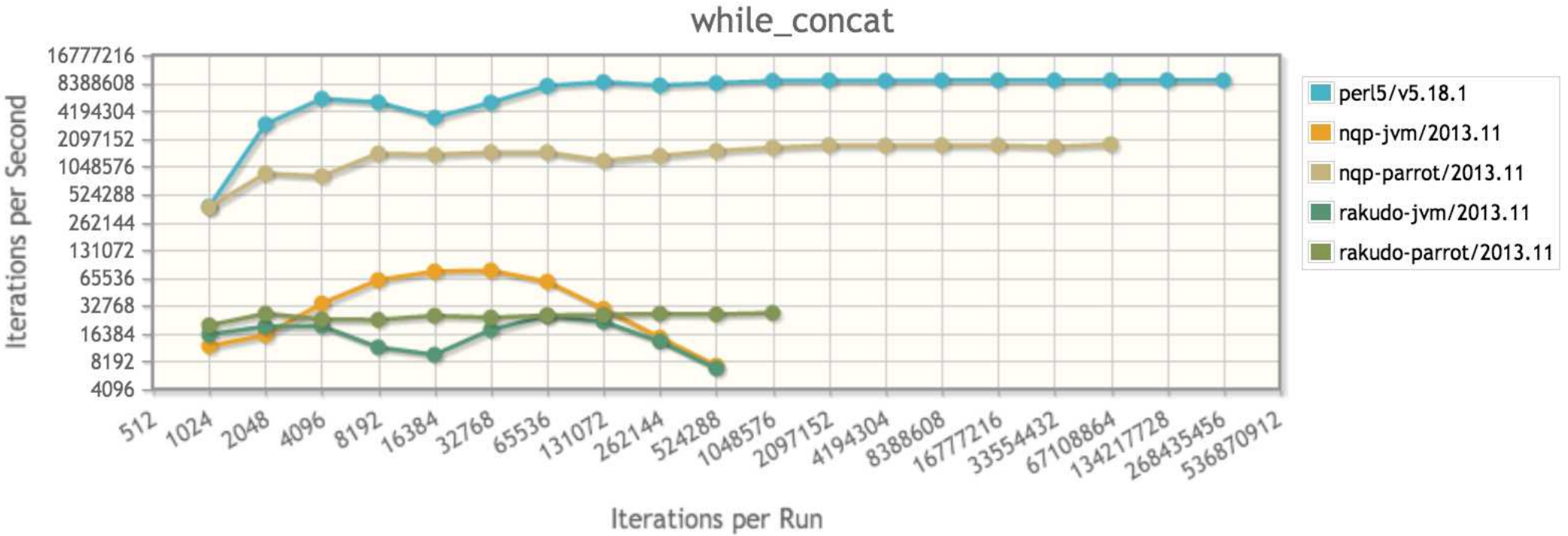
```
my $s = ""; my $i = 0; while (++$i <= SCALE) { $s .= "x" }
```

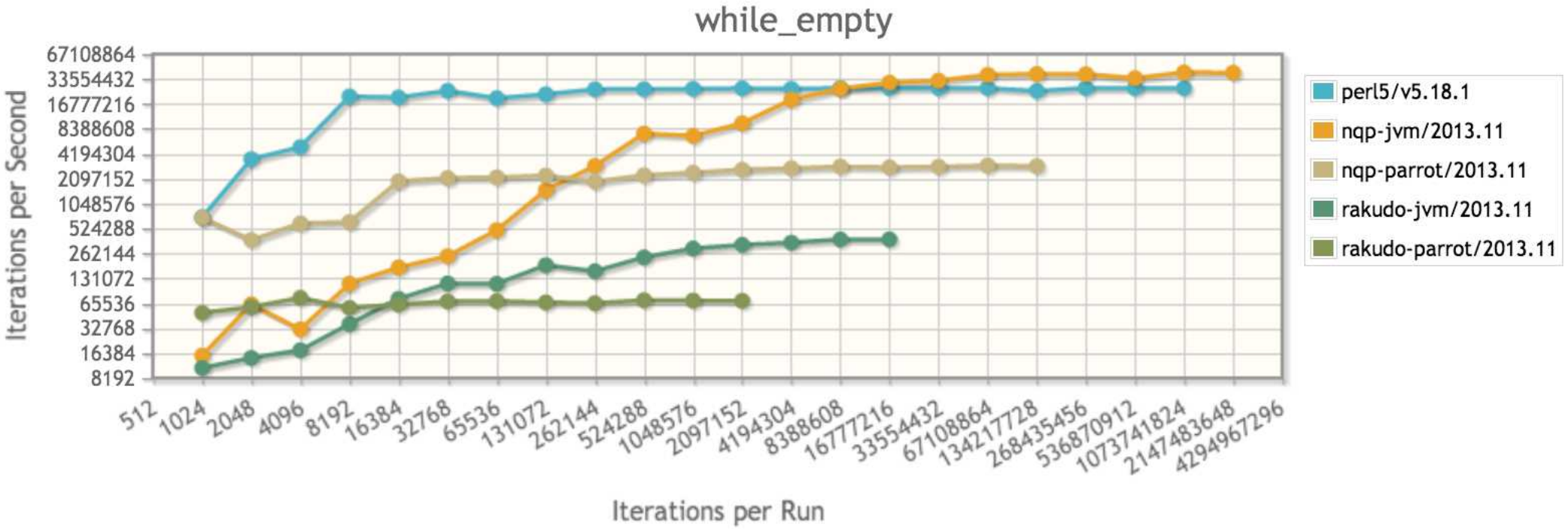
NQP

```
my $s := ""; my $i := 0;
while ($i := $i + 1) <= SCALE { $s := $s ~ "x" }
```

Perl 6

```
my $s := ""; my $i := 0;
while ($i := $i + 1) <= SCALE { $s := $s ~ "x" }
```

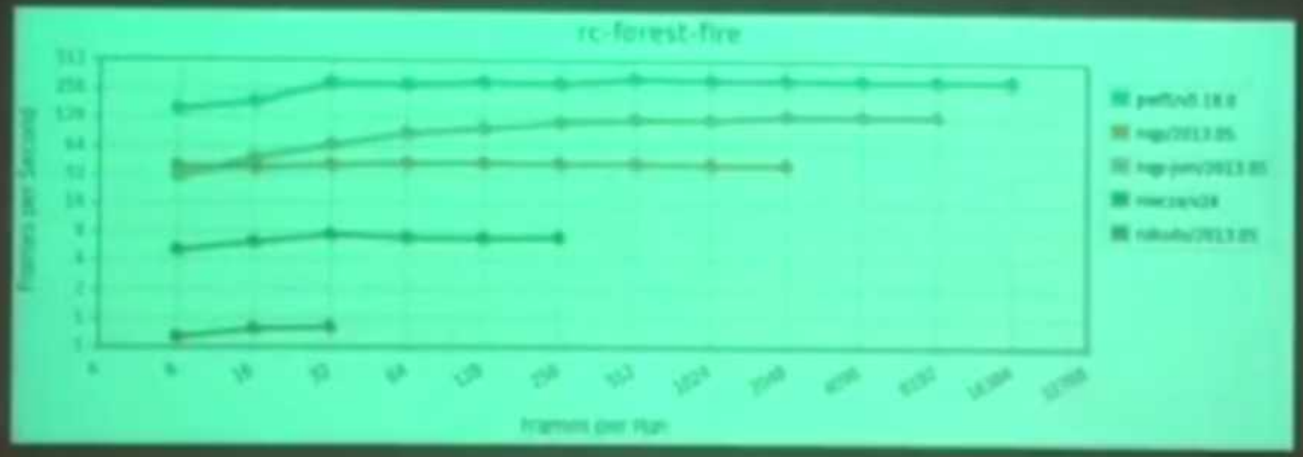

while_concat
Iterations per Second
16777216
8388608
4194304
2097152
1048576
524288
262144
131072
65536
32768
16384
8192
4096
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728
268435456
536870912
Iterations per Run
perl5/v5.18.1
nqp-jvm/2013.11
nqp-parrot/2013.11
rakudo-jvm/2013.11
rakudo-parrot/2013.11


while_empty
Iterations per Second
67108864
33554432
16777216
8388608
4194304
2097152
1048576
524288
262144
131072
65536
32768
16384
8192
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728
268435456
536870912
1073741824
2147483648
4294967296
Iterations per Run
perl5/v5.18.1
nqp-jvm/2013.11
nqp-parrot/2013.11
rakudo-jvm/2013.11
rakudo-parrot/2013.11

Caveats

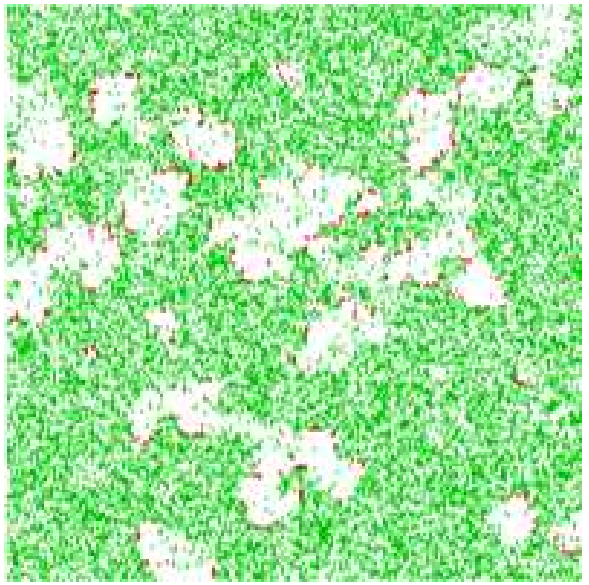
- NOT looking at startup time or memory usage.
- I'm still a newbie with P6.
- We don't yet know how to optimize P6.

The differences are telling:

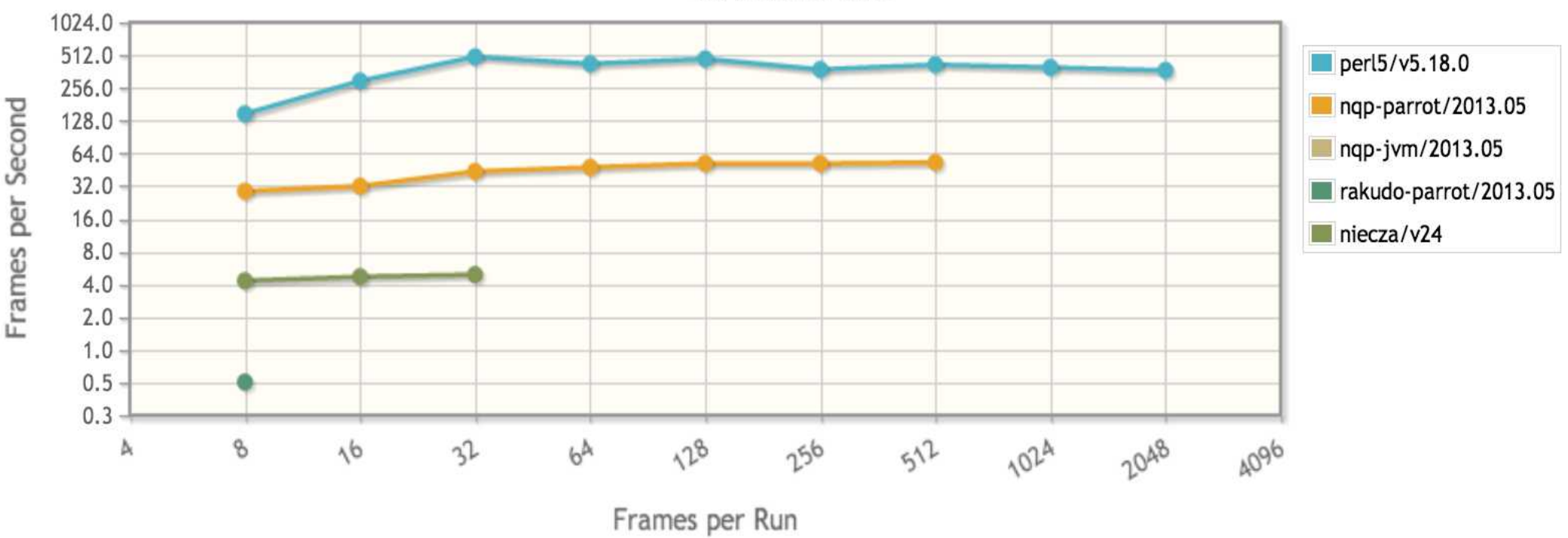


rc-forest-fire

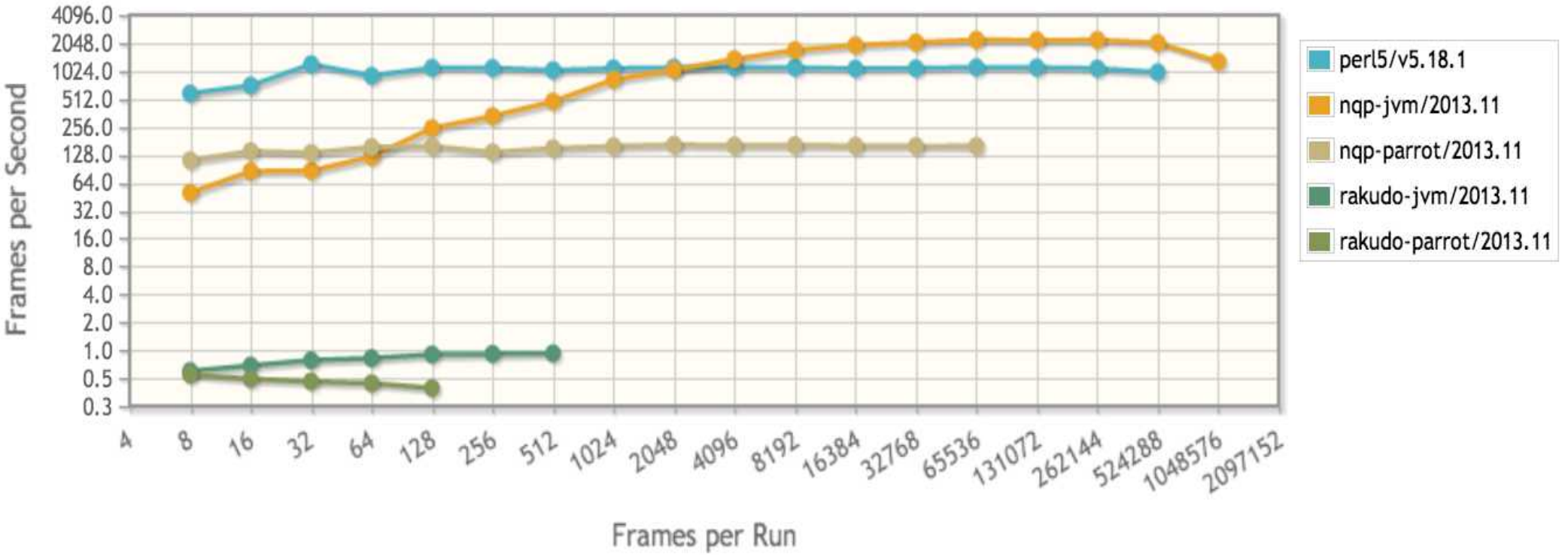
a simple 2D cellular automaton
that simulates a growing forest
with occasional fires



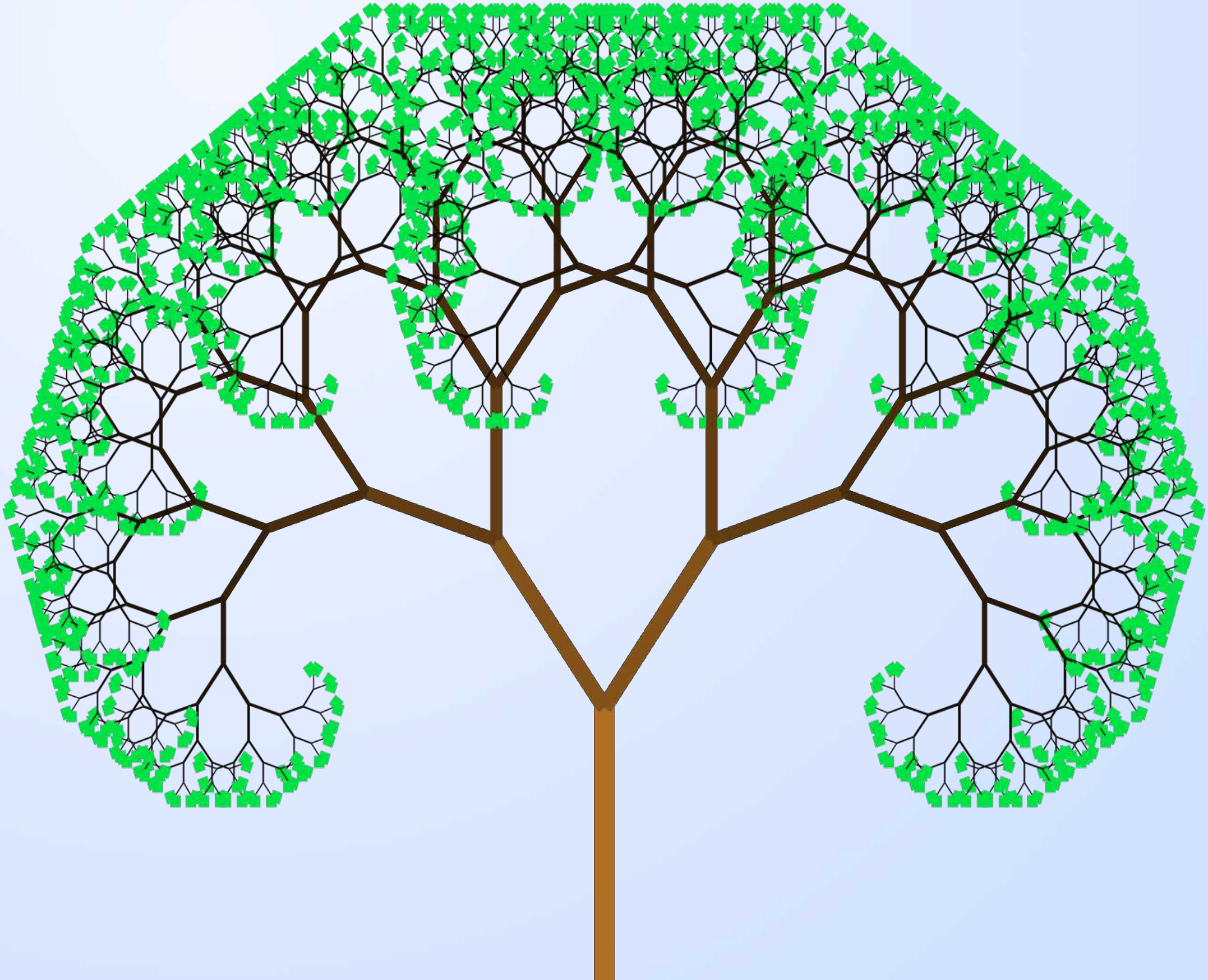
rc-forest-fire



rc-forest-fire

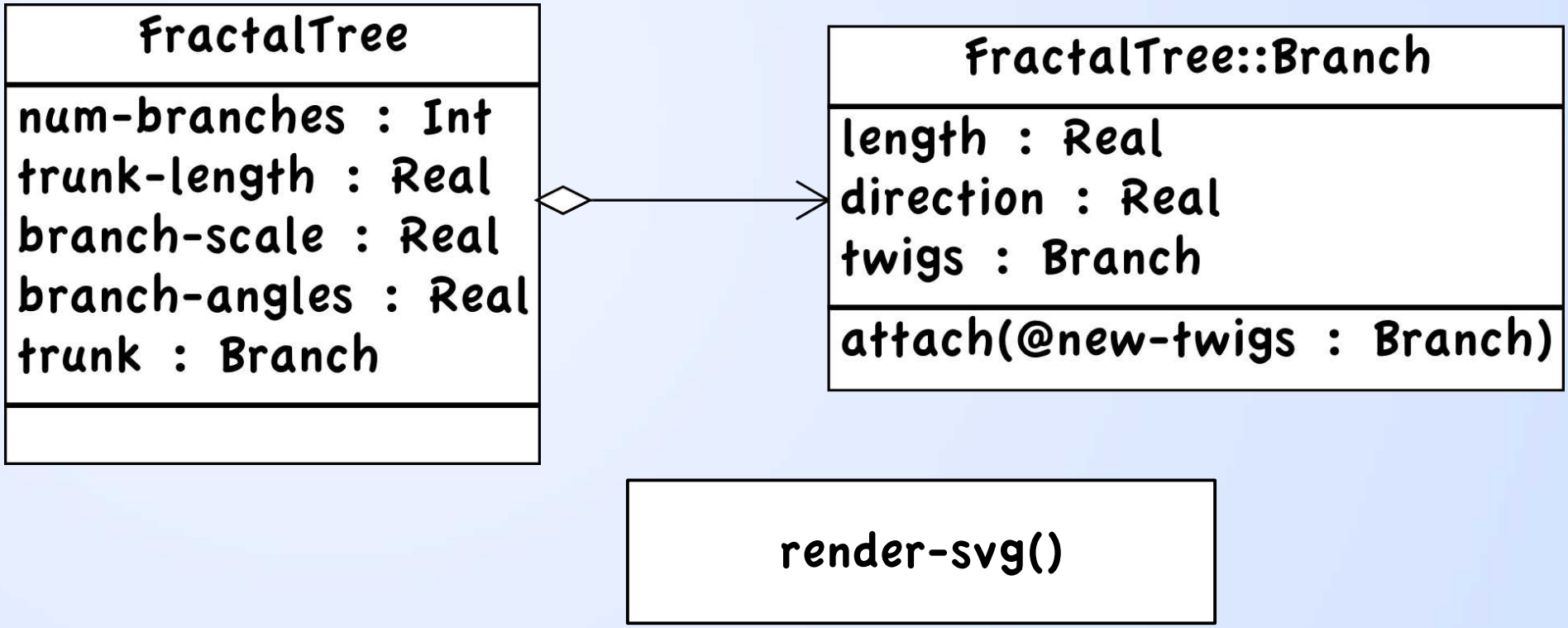



ZENTANGLE
BASICS
acrylic
solutions
exploring mixed media layer by layer
20 WAYS to DRAW A TREE
AND 44 OTHER NIFTY THINGS FROM NATURE
ELOISE RENOUF
A Sketchbook for Artists, Designers, and Doodlers
QUARRY
A TOP ARTISTS' AGENT TELLS ALL
I JUST LIKE TO MAKE THINGS

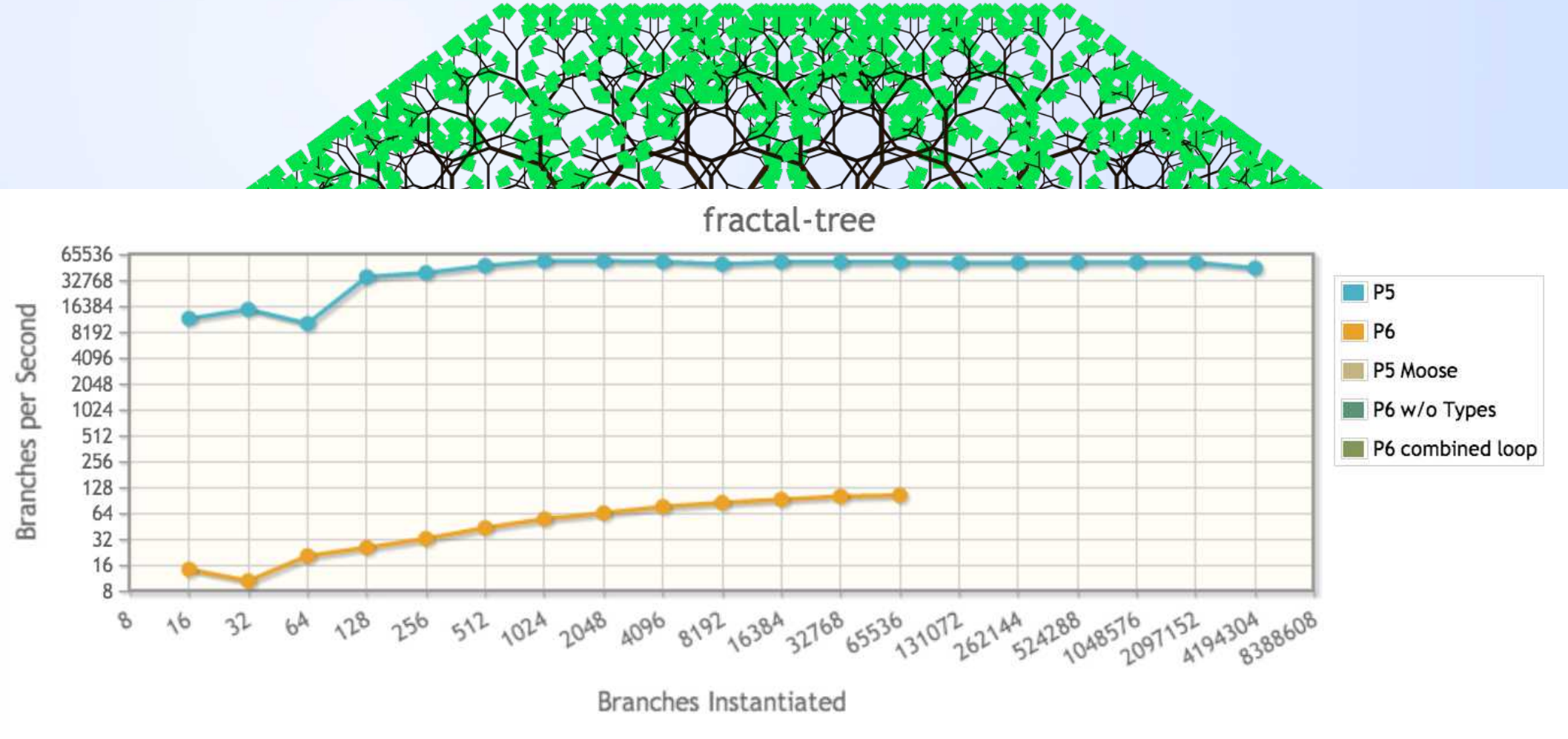


```
say $SVG-HEADER;
tree($width/2, $height, $length, 3*pi/2);
say $SVG-FOOTER;

multi tree($, $, $length where { $length < 1}, $) {}
multi tree($x, $y, $length, $angle)
{
    my ($x2, $y2) = ( $x + $length * $angle.cos,
                      $y + $length * $angle.sin );
    say "<line x1='$x' y1='$y' x2='$x2' y2='$y2'"
        ~ " style='stroke:rgb(0,0,0);stroke-width:1'/>";
    tree($x2, $y2, $length*$scale, $angle + pi/5);
    tree($x2, $y2, $length*$scale, $angle - pi/5);
}
```

Adapted from http://rosettacode.org/wiki/Fractal_tree#Perl_6




```
my $tree = FractalTree.new(:$num-branches, :$trunk-length,
                           :$branch-scale, :$branch-angles);
print render-svg(
    $tree,
    scale => $pixels-high,
    rotate => -pi/2,
    translate => Array[Real].new($pixels-wide/2, $pixels-high),
    trunk-stroke-width => ($pixels-high/50),
    :$branch-scale,
);
```


fractal-tree
Branches per Second
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
Branches Instantiated
P5
P6
P5 Moose
P6 w/o Types
P6 combined loop

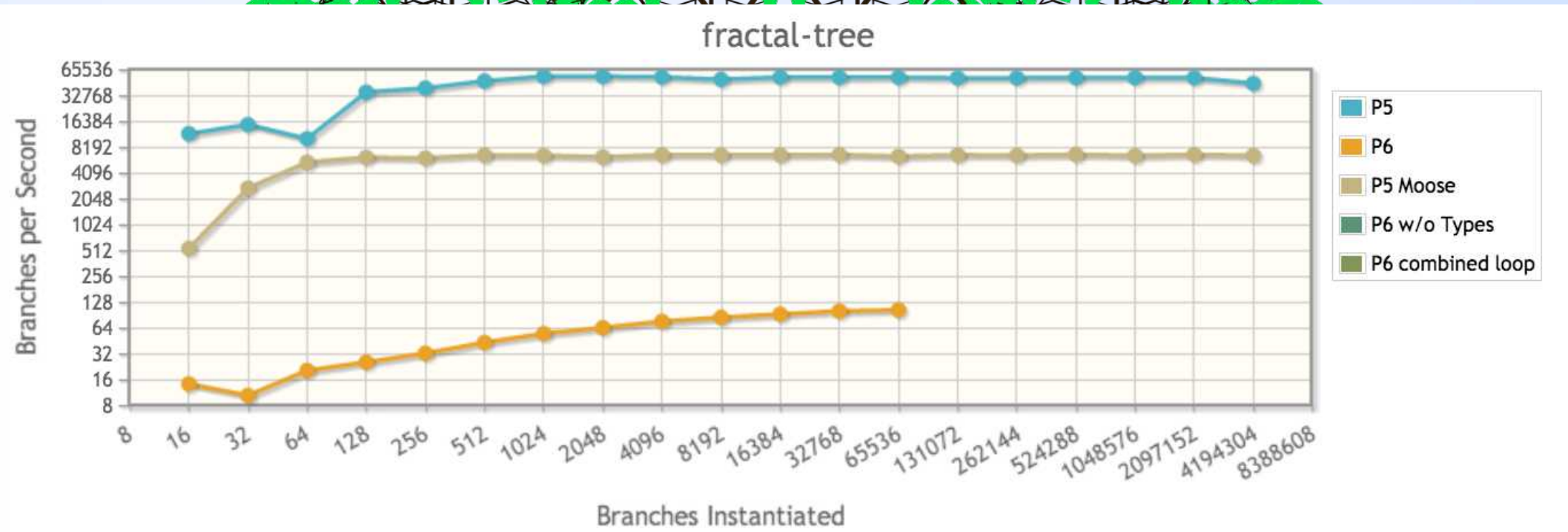
```
my $sum = Point.new([+] @points.x, [+] @points.y)
```

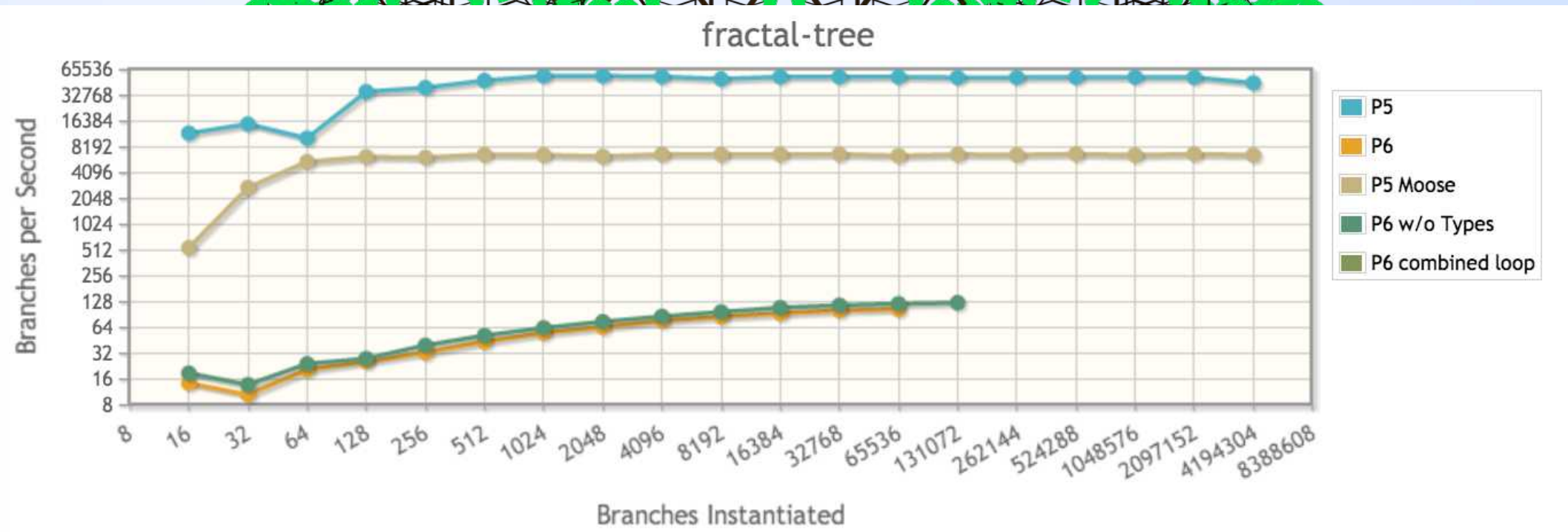
```
my $sum = Point.new([+] @points.x, [+] @points.y)
```

```
Default constructor for 'Point' only takes named arguments
```

```
my $sum = Point.new([+] (@points.x, [+] @points.y))
```

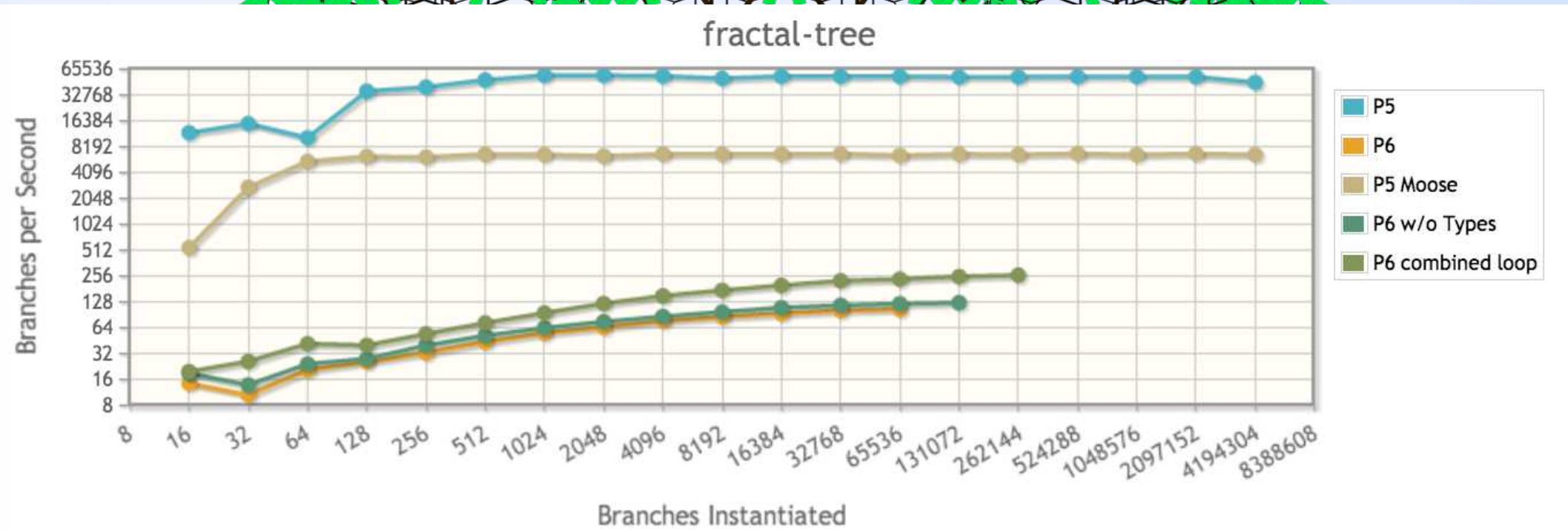
```
Default constructor for 'Point' only takes named arguments
```


fractal-tree
Branches per Second
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
Branches Instantiated
P5
P6
P5 Moose
P6 w/o Types
P6 combined loop

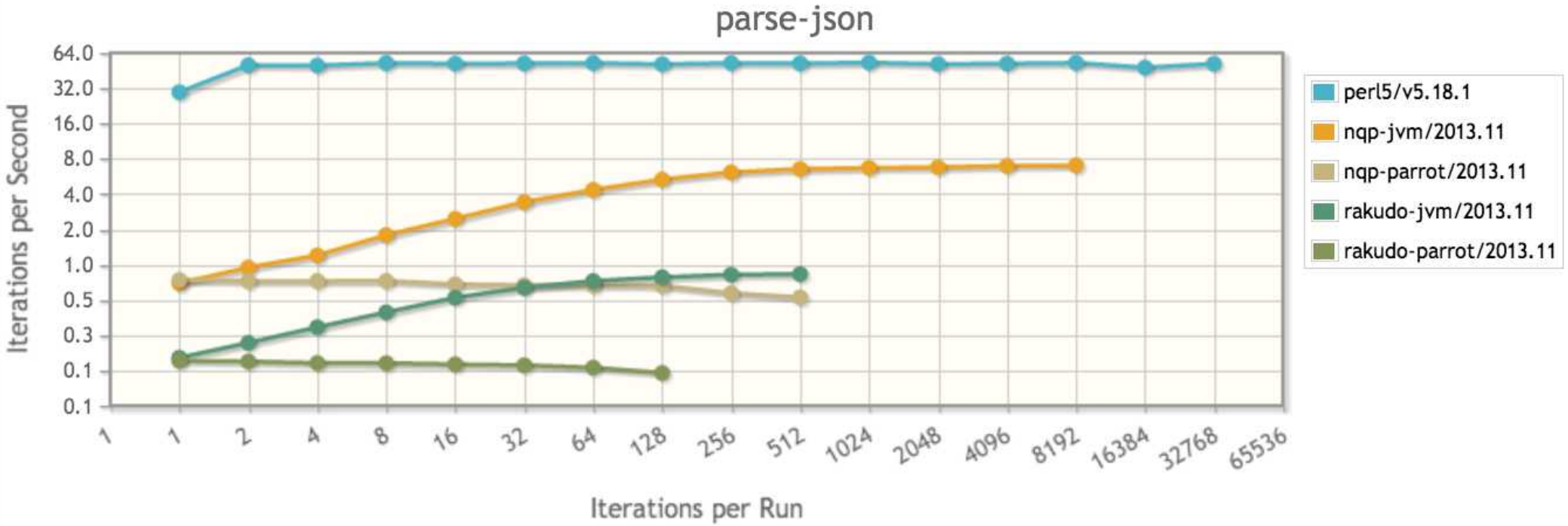

fractal-tree
Branches per Second
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
Branches Instantiated
P5
P6
P5 Moose
P6 w/o Types
P6 combined loop

```
my $num-twigs = @!branch-angles.elems min $branches-remaining;

my @twig-directions
    = $branch.direction <<+<< @!branch-angles[^$num-twigs];

my Branch @twigs = ( Branch.new(
    length => $twig-length,
    direction => $_,
) for @twig-directions );

$branch.attach(@twigs);

$branches-remaining -= @twigs.elems;
@branches-to-twig.push(@twigs);
```

```
my @branch-angles = @!branch-angles;
my $num-twigs = @branch-angles.elems min $branches-remaining;

for ^$num-twigs {
    my $angle = @branch-angles.shift;
    my $twig-direction = $branch.direction + $angle;
    my $twig = Branch.new(
        length => $twig-length,
        direction => $twig-direction,
    );

    $branch.attach($twig);

    $branches-remaining -= 1;
    @branches-to-twig.push($twig);
}
```


fractal-tree
Branches per Second
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
Branches Instantiated
P5
P6
P5 Moose
P6 w/o Types
P6 combined loop

JSON::Tiny vs JSON::Tiny

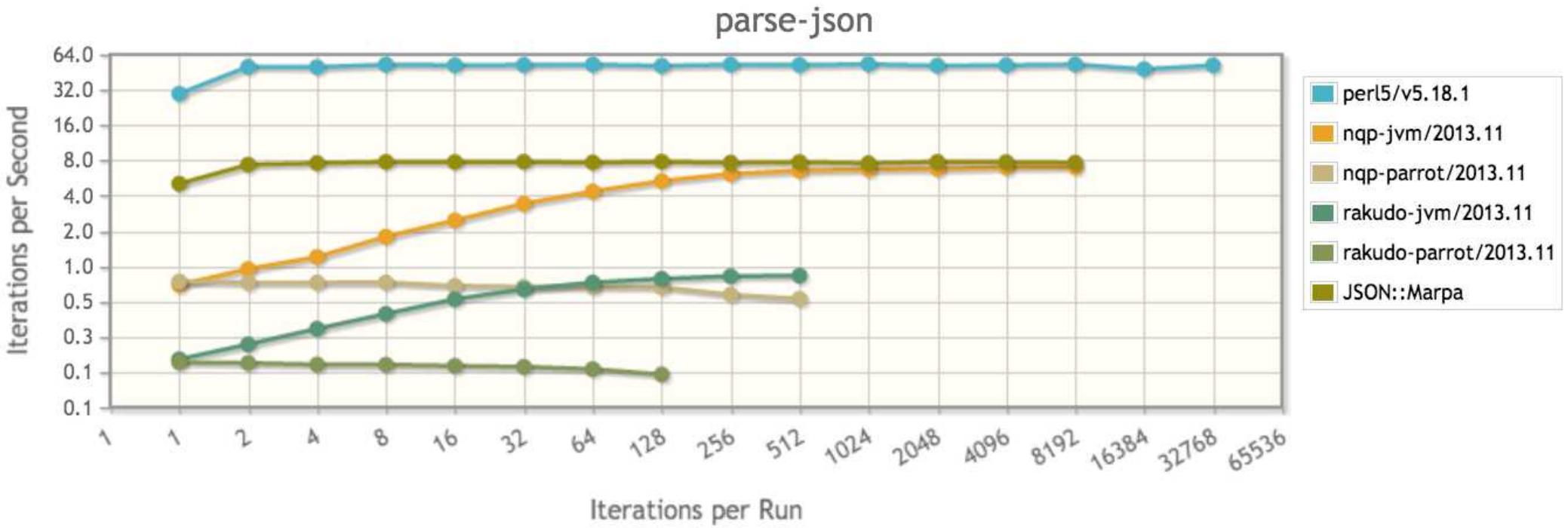


```
# Object or array
my $res = eval {
  local $_ = $bytes;

  # Leading whitespace
  m/\G$WHITESPACE_RE/gc;

  # Array
  my $ref;
  if (m/\G\[/gc) { $ref = _decode_array() }

  # Object
  elsif (m/\G\{/gc) { $ref = _decode_object() }

  # Invalid character
  else { _exception('Expected array or object') }

  # Leftover data
  unless (m/\G$WHITESPACE_RE\z/gc) {
    my $got = ref $ref eq 'ARRAY' ? 'array' : 'object';
    _exception("Unexpected data after $got");
  }

  $ref;
};
```

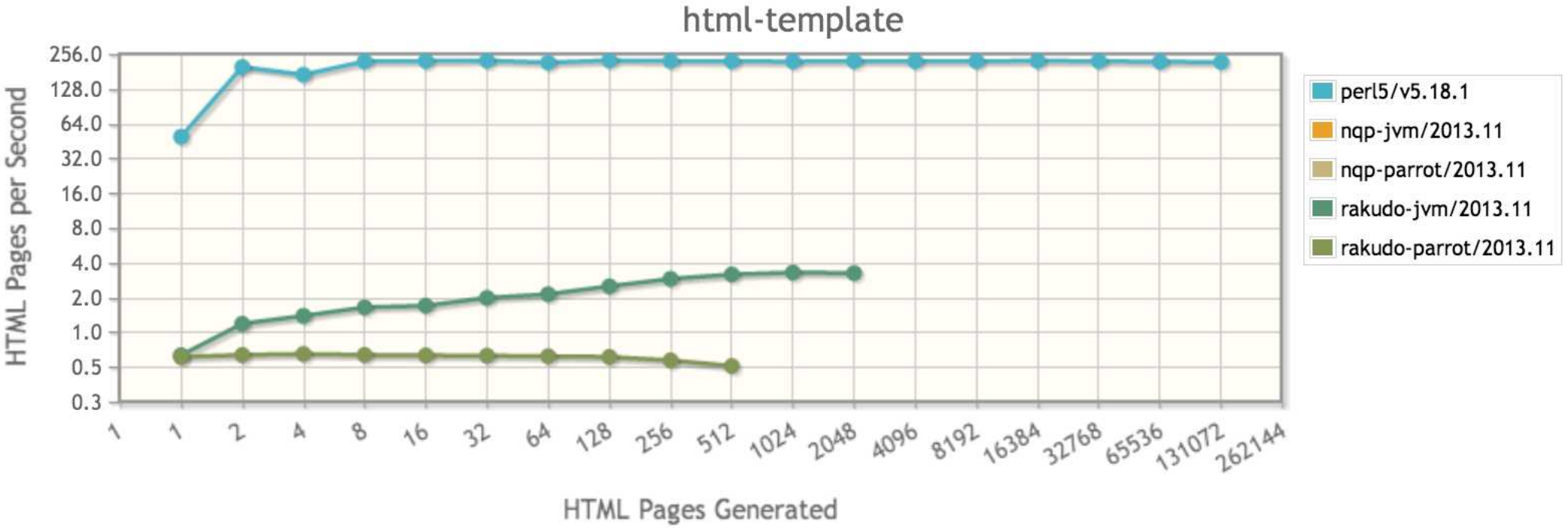
```
grammar JSON::Tiny::Grammar {
    token TOP        { ^ \s* [ <object> | <array> ] \s* $ }
    rule object      { '{' ~ '}' <pairlist>     }
    rule pairlist    { <?> <pair> * % \,              }
    rule pair        { <?> <string> ':' <value>     }
    rule array       { '[' ~ ']' <arraylist>    }
    rule arraylist   { <?> <value>* % [ \, ]        }

    proto token value {*}
    token value:sym<number> {
        '-'?
        [ 0 | <[1..9]> <[0..9]>* ]
        [ \. <[0..9]>+ ]?
        [ <[eE]> [\+|\-]? <[0..9]>+ ]?
    }
    token value:sym<true>    { <sym>    }
    token value:sym<false>   { <sym>    }
    token value:sym<null>    { <sym>    }
    token value:sym<object>  { <object> }
    token value:sym<array>   { <array>  }
    token value:sym<string>  { <string> }
```

```
package JSON::Marpa {
    use Marpa::R2;

    package JSON::Marpa::_Decoder {
        use constant JSON_SLIF => <<'MARPA_END';
:default ::= action => ::first
:start ::= JSON

JSON ::= Object | Array

Object ::= openbrace PairList closebrace action => bracketed

PairList ::= Pair* separator => comma action => pairlist

Pair ::= String colon Value action => pair

Array ::= openbracket ArrayList closebracket action => bracketed

ArrayList ::= Value* separator => comma action => arraylist
```

JSON::Tiny vs JSON::Tiny vs JSON::Marpa



HTML::Template vs HTML::Template



```
package Foo;

sub bar () is export
{
    do-stuff();
}
```

```
package Foo;

sub bar () is export
{
    do-stuff();
}
```

```
use Foo; # imports bar
```

```
package Foo;

sub bar () is export
{
    do-stuff();
}
```

```
use Foo <bar>; # no EXPORT() ?!
```

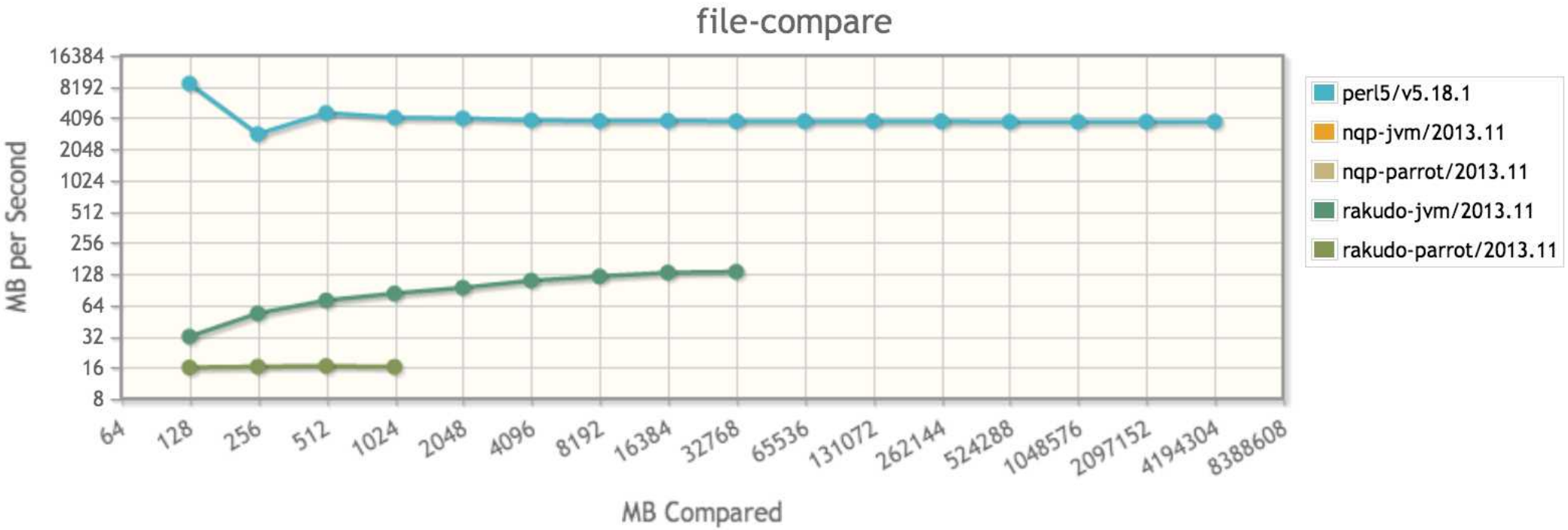
```
package Foo;

sub bar () is export
{
    do-stuff();
}
```

```
need Foo;
Foo::bar(); # no &bar
```

```
package Foo;

our sub bar () is export
{
    do-stuff();
}
```

```
need Foo;
Foo::bar();
```

File::Compare vs File::Compare



So...

P6

part of the

Programmer's Toolkit?