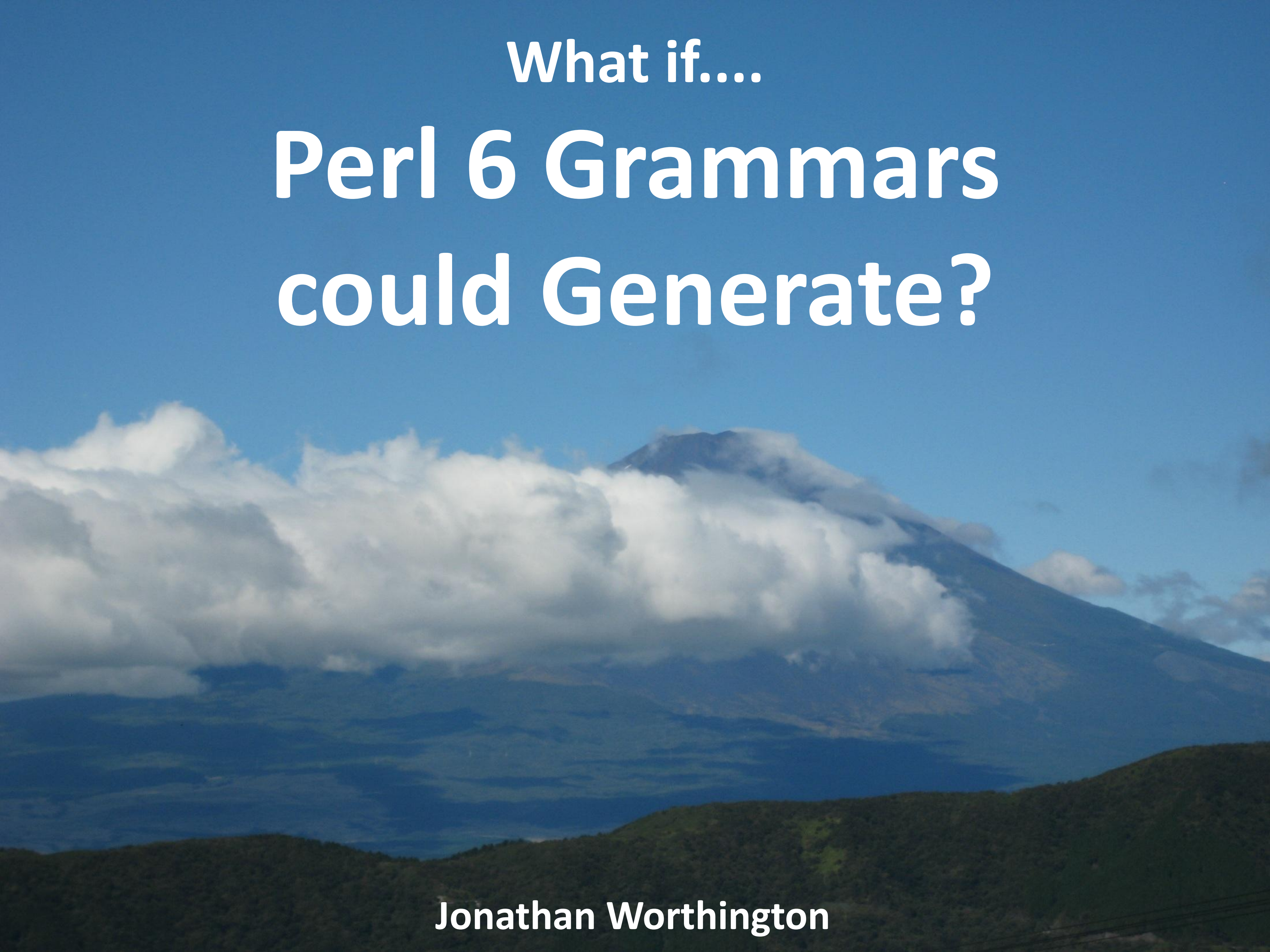


What if....

# Perl 6 Grammars could Generate?

Jonathan Worthington

Hi.

I'm jnthn.

# I do stuff…

| | |
|---|---|
| Perl 6 | Rakudo Perl 6<br>Work on the JVM port<br>Oh, and also Moar |
| $dayjob | I teach and mentor<br>On software architecture<br>Git and .Net too |
| Other | Traveling the world<br>Hunting for Indian food<br>Drinking awesome beer |

# I do stuff...

| | |
|---|---|
| Perl 6 | Rakudo Perl 6<br>Work on the JVM port<br>Oh, and also Moar |
| $dayjob | I teach and mentor<br>On software architecture<br>Git and .Net too |
| Other | Traveling the world<br>Hunting for Indian food<br>Drinking awesome beer |

It's good to know multiple languages and communities. Can steal ideas back and forth.

# Rx

Reactive eXtensions

Realized that the observer pattern is the mathematical dual of the enumerator pattern

Thus, you can define the familiar enumerator combinators on observables too

That's awesome!

Huh???!

Let me show you...

# Lazy lists

## Allow us to talk about potentially infinite lists:

```
my @a       := 1..Inf;
my @primes  := @a.grep(*.is-prime);
my @nprimes := @primes.map({ "{++state $n}: $_" });
.say for @nprimes[^10];
```

## Or with the feed syntax:

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

# Lazy lists are all about pulling

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

# Lazy lists are all about pulling

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

Hey, `map`! I need, like, 10 things!

# Lazy lists are all about pulling

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

**Hey, `map`! I need, like, 10 things!**

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

**Hey, `grep`! I need 10 things to map!**

# Lazy lists are all about pulling

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

Hey, map! I need, like, 10 things!

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

Hey, grep! I need 10 things to map!

```
.say for (1..Inf
    ==> grep(*.is-prime)
    ==> map({ "{++state $n}: $_" })
    )[^10];
```

Hey Range, I need some values...

Each thing blocks on getting values from the previous thing

What if we want to push instead?

# Observables

```
role Observable {
    has @!observers;

    method subscribe($observer) {
        push @!observers, $observer;
        $observer
    }

    method unsubscribe($observer) {
        @!observers .= grep({ $^o !=== $observer });
    }

    method publish($obj) {
        @!observers>>.handle($obj)
    }
}
```

# A simple event source

**An `Observable` thing that publishes each line entered, just to give us an easy example**

```
class ReadLineSource does Observable {
    has $.fh;
    method enterloop() {
        loop {
            self.publish($.fh.get());
        }
    }
}
```

# Redefine stuff like `grep`

**They subscribe to an `Observable`, are themselves `Observable`, and publish stuff**

```
multi grep($matcher, Observable $ob) {
    my class GrepSubscriber does Observable {
        has $.matcher;
        method handle($obj) {
            if $obj ~~ $.matcher {
                self.publish($obj);
            }
        }
    }
    $ob.subscribe(GrepSubscriber.new(:$matcher))
}
```

# Then we can do stuff like…

```
my $src = ReadLineSource.new(fh => $*IN);
```

# Then we can do stuff like...

```
my $src = ReadLineSource.new(fh => $*IN);
$src
    ==> grep(/^\d+$/)
    ==> into my $nums;
```

# Then we can do stuff like…

```
my $src = ReadLineSource.new(fh => $*IN);
$src
    ==> grep(/^\d+$/)
    ==> into my $nums;

$nums
    ==> grep(*.Int.is-prime)
    ==> call(-> $p { say "That's prime!" });
```

# Then we can do stuff like…

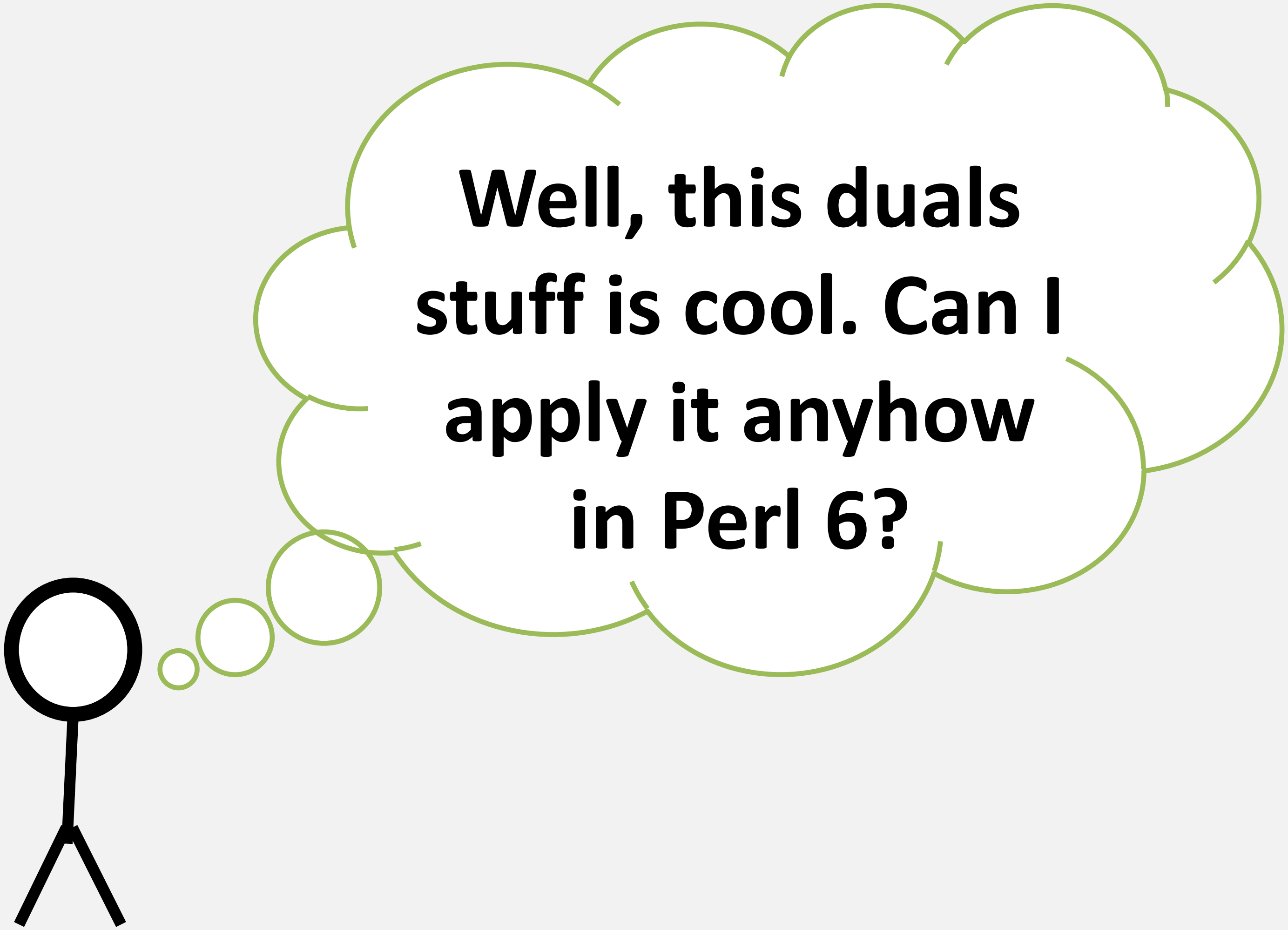
```
my $src = ReadLineSource.new(fh => $*IN);
$src
    ==> grep(/^\d+$/)
    ==> into my $nums;

$nums
    ==> grep(*.Int.is-prime)
    ==> call(-> $p { say "That's prime!" });

$nums
    ==> map(-> $n {
            state $total += $n;
            $total >= 100 ?? 'More than 100' !! ()
        })
    ==> first()
    ==> call(-> $msg { say $msg });
```

New data is pushed through the pipeline as it is available.

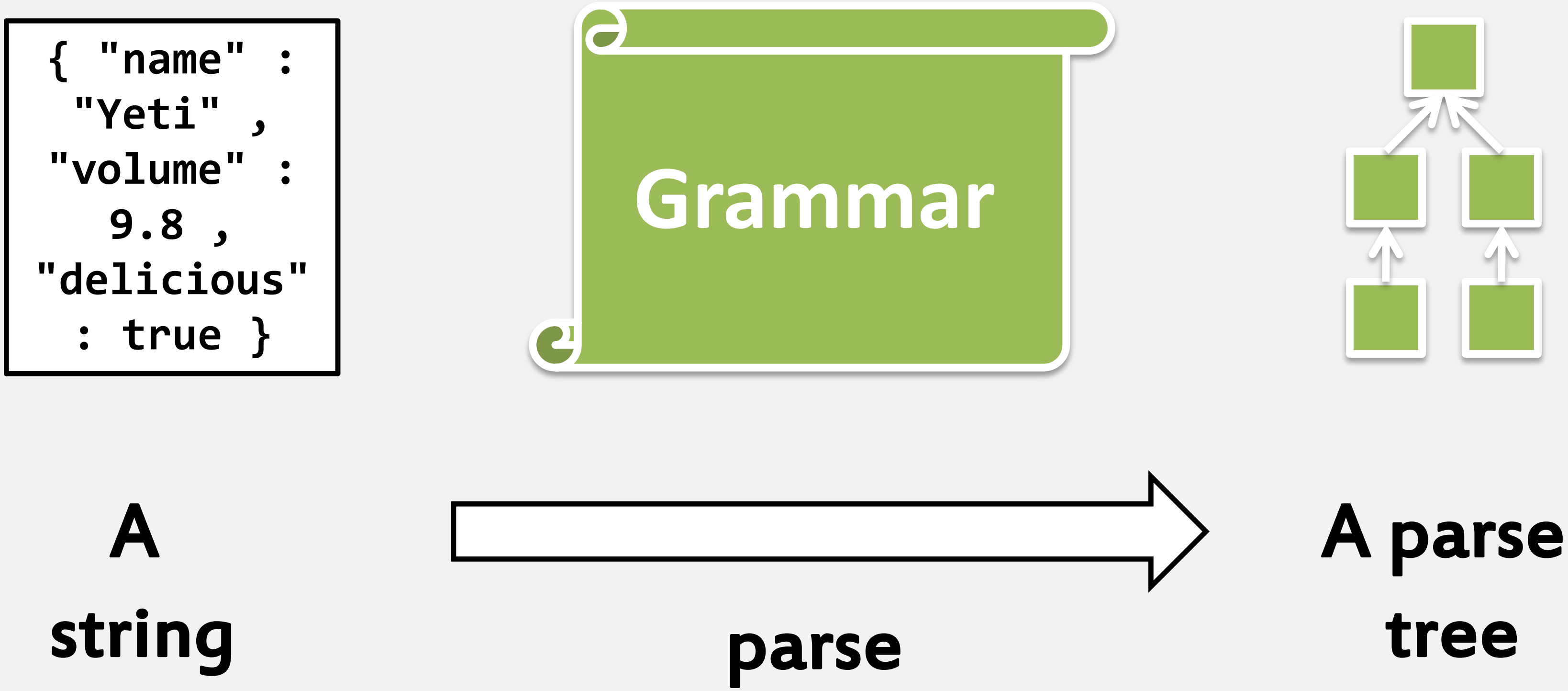
No blocking. Nice for async.



Well, this duals stuff is cool. Can I apply it anyhow in Perl 6?

# Grammars

```
SomeGrammar.parse($string)
```



```
{ "name" :
 "Yeti" ,
"volume" :
  9.8 ,
"delicious"
 : true }
```

A string

parse

A parse tree

# Example: JSON::Tiny

```
grammar JSON::Tiny::Grammar {
    token TOP       { ^ [ <object> | <array> ] $ }
    …
}
```

# Example: JSON::Tiny

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

# Example: JSON::Tiny

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

# Example: JSON::Tiny

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

    proto token value {*}
    …
}
```

# Example: JSON::Tiny

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

    proto token value {*}
    token value:sym<true>    { <sym>    }
    token value:sym<false>   { <sym>    }
    …
}
```

# Example: JSON::Tiny

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

    proto token value {*}
    token value:sym<true>    { <sym>    }
    token value:sym<false>   { <sym>    }
    token value:sym<object>  { <object> }
    token value:sym<array>   { <array>  }
    token value:sym<string>  { <string> }
    # etc.
}
```

# Actions

**We would like to get arrays and hashes out, not just a bunch of `Match` objects**

**An action method is like a callback that runs after each grammar rule completes**

**It can build up another data structure alongside the parse tree**

# Example: JSON::Tiny Actions

```
class JSON::Tiny::Actions {
    method value:sym<number>($/) { make +$/.Str }
    method value:sym<true>($/) { make Bool::True }
    method value:sym<false>($/) { make Bool::False }
    …
}
```

# Example: JSON::Tiny Actions

```
class JSON::Tiny::Actions {
    method value:sym<number>($/) { make +$/.Str }
    method value:sym<true>($/) { make Bool::True }
    method value:sym<false>($/) { make Bool::False }
    …
    method object($/) {
        make $<pairlist>.ast.hash;
    }
    method pairlist($/) {
        make $<pair>>>.ast.flat;
    }
    method pair($/) {
        make $<string>.ast => $<value>.ast;
    }
    …
}
```

# Putting it all together…

Using the grammar and actions together, we can turn a JSON string into a Perl 6 data structure

```
sub from-json($text) is export {
    my $a = JSON::Tiny::Actions.new();
    my $o = JSON::Tiny::Grammar.parse($text,
        :actions($a));
    return $o.ast;
}
```

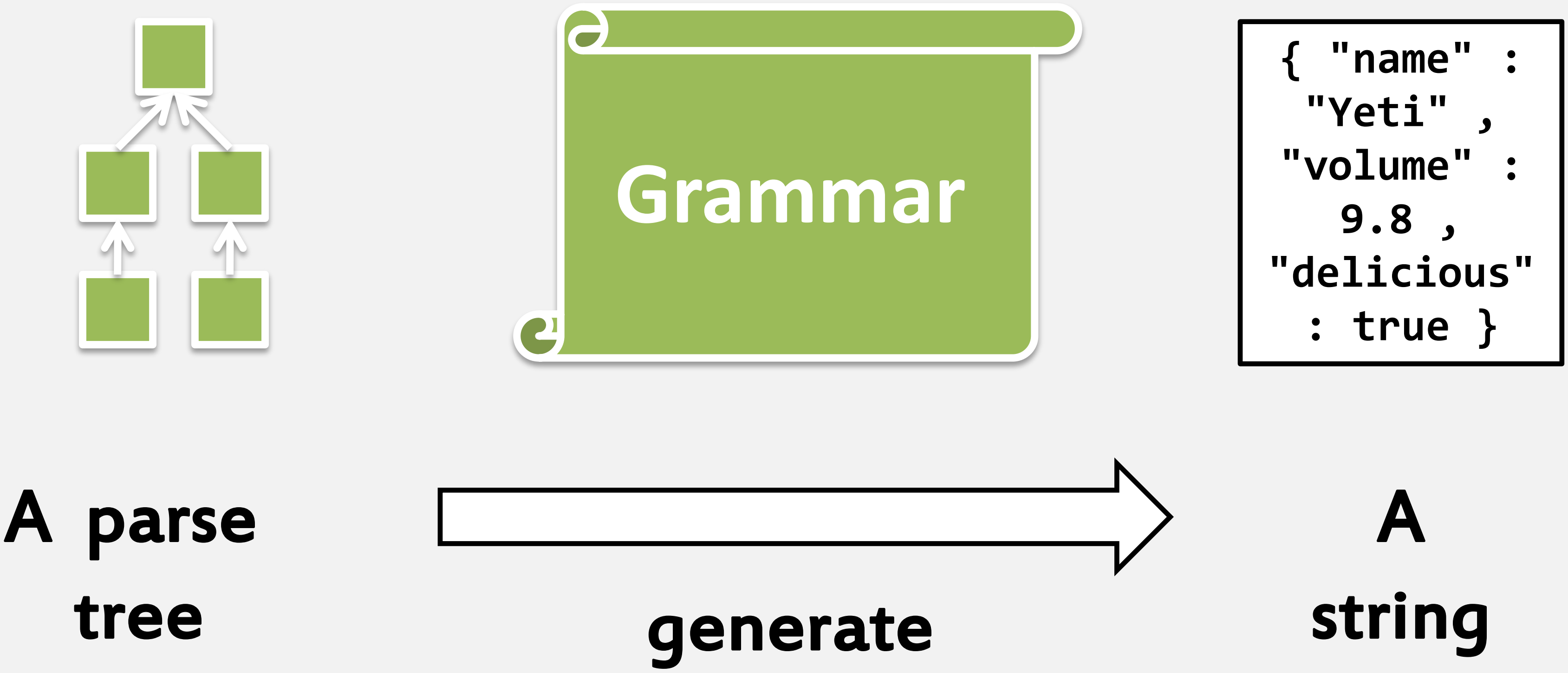
# What if…

# What if…

```
SomeGrammar.generate($tree)
```

# What if...

`SomeGrammar.generate($tree)`



```
{ "name" :
 "Yeti" ,
"volume" :
 9.8 ,
"delicious"
 : true }
```

A parse tree

generate

A string

# A simple example

```
grammar SimpleSentence {
    token TOP { <sentence> }
    rule sentence { <subject> <verb> <object>'.' }
}

say SimpleSentence.generate(\(
    sentence => \(
        subject => 'Petrucci',
        verb    => 'plays',
        object  => 'guitar'
    )));
```

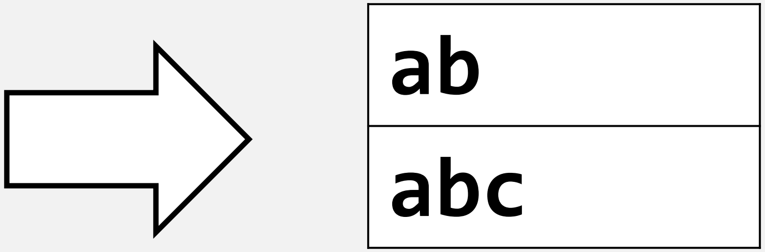
```
Petrucci plays guitar.
```

# Breaking it down...

**Each regex construct taking part in parsing can be seen as returning a (potentially empty) lazy list of all the ways it can match**

**Regex:** `[ab||abc]`

**Input:** `abcdef`



ab

abc

**For generation, each regex construct takes a tree and returns a lazy list of possible strings**

# Backing off

Some paths we go down won't work out well, and we need to back off and try another way

For example, if we can't parse an `<object>`…

```
[
|| <subject> <verb> <object>'.'
|| <subject> <verb>'.'
]
```

…then we have to fall back to the other path.

# Generation will need back-off too

**How will this work for generation?**

**Our starting point for generation is the tree. If we don't have an object entry in the tree…**

```
[
|| <subject> <verb> <object>'.'
|| <subject> <verb>'.'
]
```

Not in tree

**…then that is the indication to abort this path.**

# Getting the grammar rules

During compilation, the compiler turns each rule in a grammar into a tree

We can steal these trees!

How? By writing an EXPORT sub that mixes a role in to the compiler's actions class

Yay for having the compiler written in Perl 6!

# Getting the grammar rules

```
our sub EXPORT() {
    setup_ast_capture(%*LANG);
}


…
```

# Getting the grammar rules

```
our sub EXPORT() {
    setup_ast_capture(%*LANG);
}

sub setup_ast_capture(%lang) {
    %lang<MAIN-actions> := %lang<MAIN-actions> but role {
        …
    }
}
```

# Getting the grammar rules

```
our sub EXPORT() {
    setup_ast_capture(%*LANG);
}

sub setup_ast_capture(%lang) {
    %lang<MAIN-actions> := %lang<MAIN-actions> but role {
        method regex_def(Mu $m) {
            my $nibble := $m.hash<nibble>;
            callsame;
            $*PACKAGE.HOW.save_rx_ast($*PACKAGE,
                $*DECLARAND.name, $nibble.ast);
        }
    }
}
```

# Adding the generate method

**Just a custom meta-object for grammar that composes an extra role by default**

```
my module EXPORTHOW {
    class grammar is Metamodel::GrammarHOW {
        method new_type(|) {
            my $type := callsame();
            $type.HOW.add_role($type, Generative);
            $type
        }

        # Also, storage for the ASTs/generators
    }
}
```

# The Generative role

```
my role Generative {
    method generate($match = \(), :$rule = 'TOP', :$g) {
        my @gen := self.^generator($rule).generate(self,
            $match);
        if $g {
            gather {
                while @gen {
                    take @gen.shift.Str;
                }
            }
        }
        else {
            @gen[0].Str
        }
    }
}
```

# One Generator per grammar rule

```
my class Generator {
    has Mu $!ast;
    has &!generator;
    …

    method generate($g, $match) {
        (&!generator //= self.compile($!ast))($g, $match)
    }

    method compile(Mu $ast) {
        given $ast.rxtype // 'concat' {
            …
        }
    }
}
```

# Literals: just one way to do it

**Any literal string in a grammar is easy for generation: it always generates the literal**

```
when 'literal' {
    return -> $, $ { [$ast.list[0]] }
}
```

**Note that it's put inside of an array, since the design here expects to get a list of all possible results. For literals, there's one possibility.**

# Sequential alternations: more fun

```
when 'altseq' {
    my @generators = $ast.list.map({ self.compile($_) });
    …
}
```

# Sequential alternations: more fun

```
when 'altseq' {
    my @generators = $ast.list.map({ self.compile($_) });
    return -> $g, $match {
        gather {
            …
        }
    }
}
```

# Sequential alternations: more fun

```
when 'altseq' {
    my @generators = $ast.list.map({ self.compile($_) });
    return -> $g, $match {
        gather {
            for @generators -> $altgen {
                my @results := $altgen.($g, $match).list;
                while @results {
                    take @results.shift();
                }
                CATCH {
                    when X::Grammar::Generative::Unable { }
                }
            }
            …
        }
    }
}
```

# Sequential alternations: more fun

```
when 'altseq' {
    my @generators = $ast.list.map({ self.compile($_) });
    return -> $g, $match {
        gather {
            for @generators -> $altgen {
                my @results := $altgen.($g, $match).list;
                while @results {
                    take @results.shift();
                }
                CATCH {
                    when X::Grammar::Generative::Unable { }
                }
            }
            X::Grammar::Generative::Unable.new.throw()
        }
    }
}
```

# And so it goes on…

Define similar things for…

Concatenation
Alternation
Quantifiers
Subrule calls
Anchors

# Some work later...

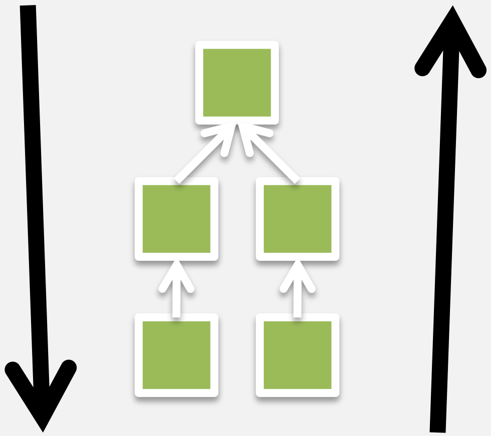
```
say JSON::Tiny::Grammar.generate(\(
    object => \(
        pairlist => \(
            pair => [
                \(
                    string              => '"name"',
                    'value:sym<string>' => '"Yeti"'
                ),
                \(
                    string              => '"volume"',
                    'value:sym<number>' => 9.8
                )
            ]
        ))))
```

```
{     "name" : "Yeti" ,  "volume" : 9.8  }
```

Nice, but what is the equivalent to action methods?

# What actions do

**Parsing is recursive descent-ish (with NFAs to trim impossible paths early). Action methods are run on the way back up to the top**

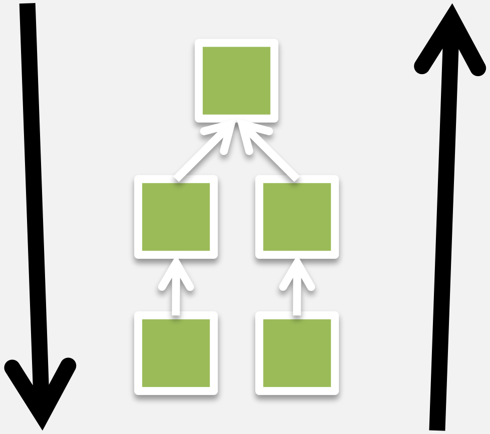


**Parse on the way down**

**Run actions on the way up**

# Turning it backwards

**We already have the data structure, and want a string. Thus, the backwards action methods need to be run on the way down**



**Run backward actions on the way down**

**Assemble the result string on the way up**

# Backtions

**Take an object, produce a bunch of captures.**
**Use `dice` to specify objects that need further deconstruction further down in the tree.**

```
class JSON::Tiny::Backtions {
    multi method TOP(@a) {
        \(array => dice(@a))
    }

    multi method TOP(%h) {
        \(object => dice(%h))
    }
    …
}
```

# More backtions

```
method object(%h) {
    \(pairlist => dice(%h.pairs))
}

method pairlist(@pairs) {
    my @diced = @pairs.map(&dice);
    \(pair => @diced)
}

method pair($p) {
    \(string => qq["$p.key()"],
      value => dice($p.value))
}

multi method value(Real $n) {
    \('value:sym<number>' => $n)
}
```

# So, finally…

**Putting the pieces together, we can go straight from Perl 6 data structure back to text**

```
say JSON::Tiny::Grammar.generate(
    {
        name => 'Yeti',
        volume => 9.8,
        delicious => True
    },
    backtions => JSON::Tiny::Backtions);
```

```
{     "name" : "Yeti" ,  "volume" : 9.8 ,  "delicious"
: true  }
```

# Grammar::Generative

**Getting these capabilities just needs…**

```
use Grammar::Generative;
```

**Any grammar in the lexical scope of such a `use` statement will get a `generate` method**

**Warning: it's (very) experimental still ☺**

# Thank you!

**Get the code from…**

github.com/jnthn/grammar-generative

**Questions?**