Next Ruby

Crafting a new ideal language with DSL and ruby-next

Ruby dev team goals

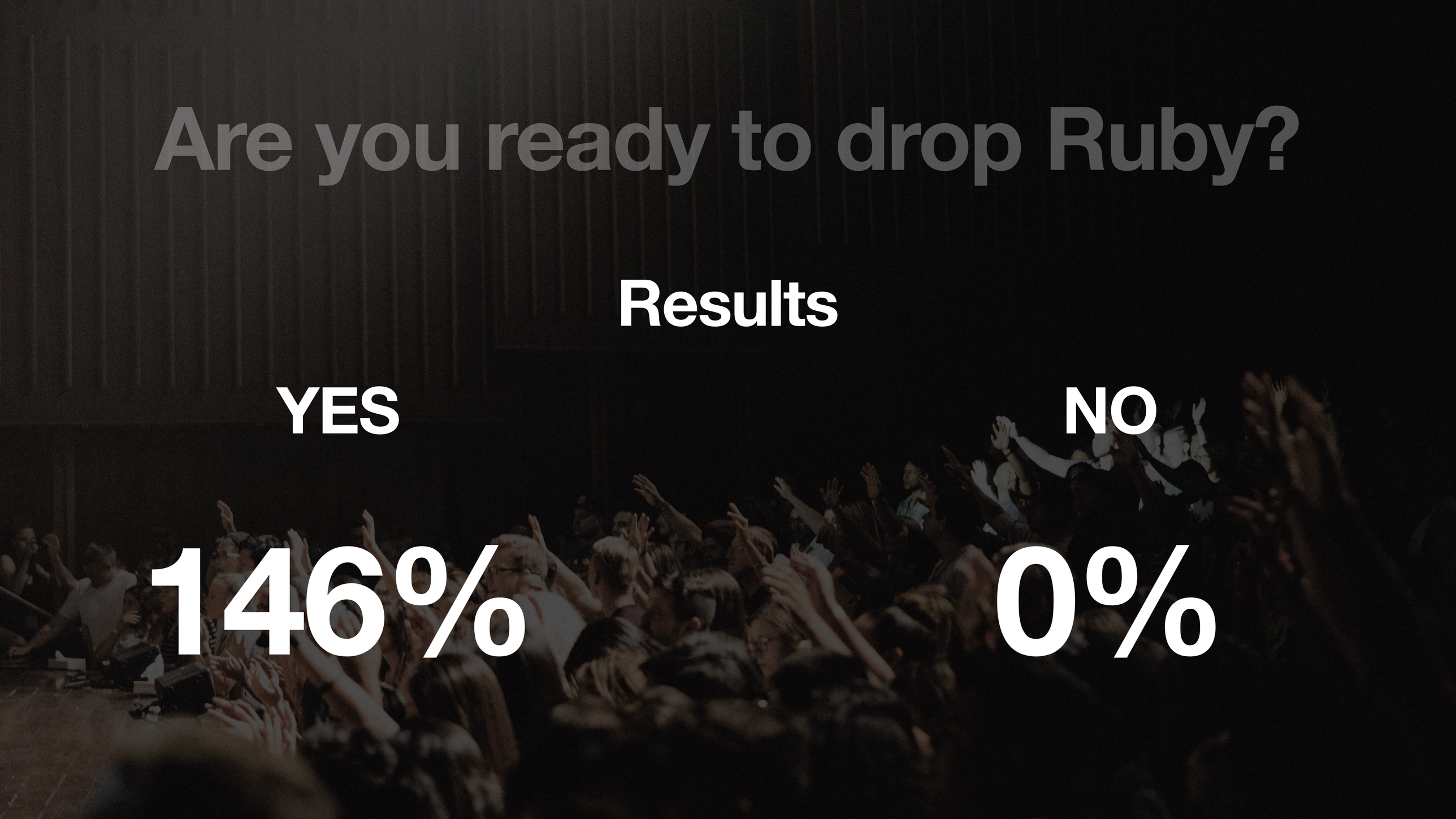
- Performance (JIT)
- Concurrency (Ractors)
- Typing (RBS)

Definitely YES

HR inbox

Are you ready to drop Ruby?

Are you ready to drop Ruby?



Wait a minute
Who are you?

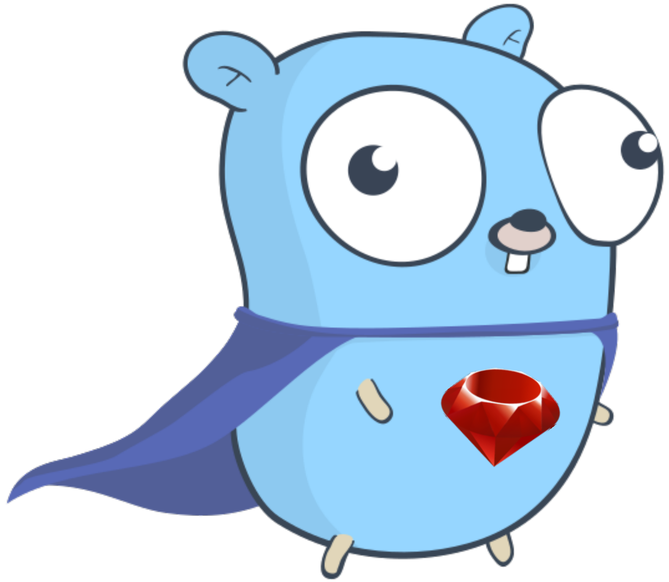
What's next?

How to move away from Ruby?

- **Complete rewrite**
- **Microservices**
- **Perfect solution, silver bullet...**

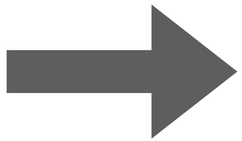
Let's write Go

Let's write Go in Ruby!



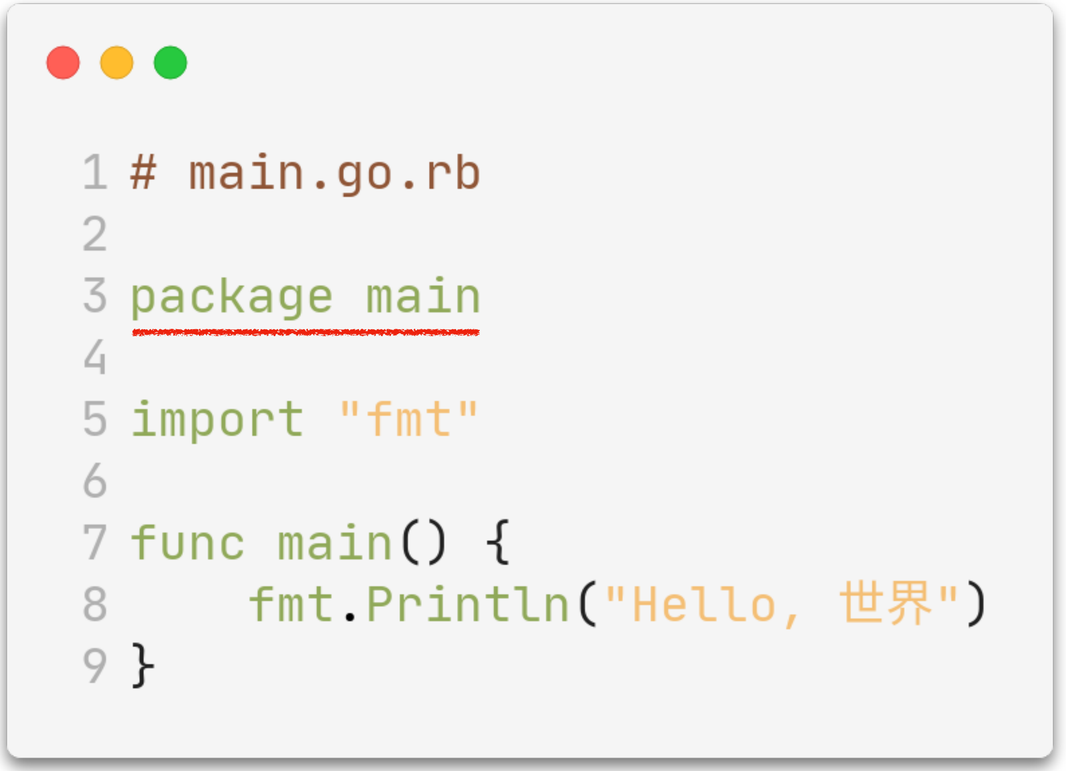
A Tour of Go: Hello World

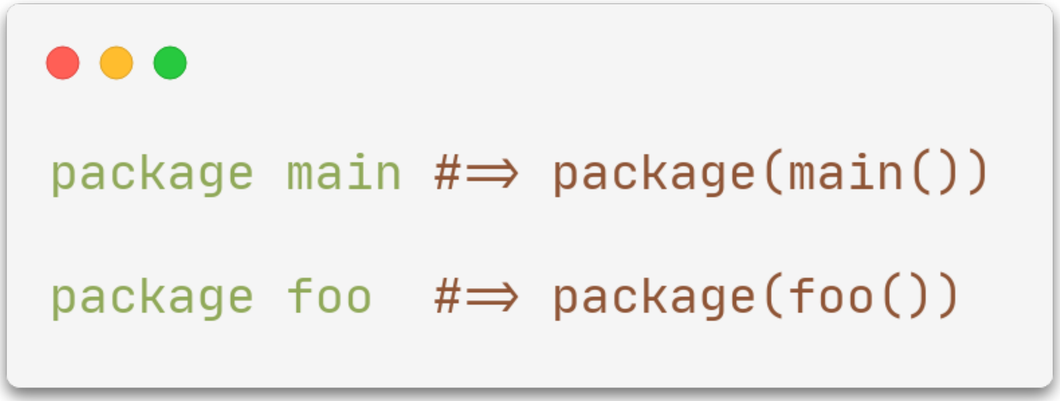
```go
// main.go

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```



```ruby
# main.go.rb

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

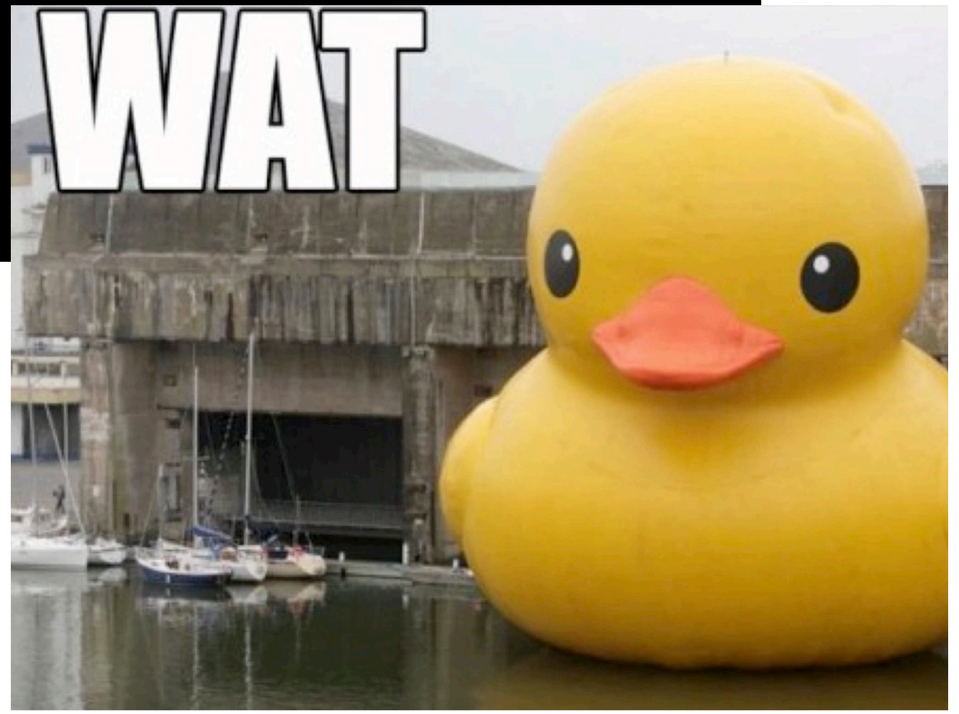
```
$ ruby main.go.rb
main.go.rb:1:in `<main>': undefined local variable or method
`main' for main:Object (NameError)
```


1 # main.go.rb
2
3 package main
4
5 import "fmt"
6
7 func main() {
8   fmt.Println("Hello, 世界")
9 }


package main #⟹ package(main())
package foo #⟹ package(foo())

```
package main #⟹ package(main())

package foo  #⟹ package(foo())
```

```
failbowl:~(master!?) $ irb
>> ruby has no bare words
NameError: undefined local variable or method `words' fo
r main:Object
        from (irb):1
>> def method_missing(*args); args.join(" "); end
=> nil
>> ruby has bare words
=> "ruby has bare words"
>> bare words can even have bangs!
=> "bare words can even have bangs!"
>>
```



```
package main #⟹ package(main())

package foo  #⟹ package(foo())
```

```
1 class << self
2   def method_missing(name, *args)
3     name
4   end
5 end
6
7 # package main #⟹ package(:main)
```

Go package

Functions, types, variables, and constants are visible to all source files within the same package

Go package

Functions, types, variables, and constants are visible to all source files within the same package

Feels like a Ruby Module

```
package foo

# ⇒

module GoRuby
  module Foo
    # defined functions
  end
end
```

```ruby
class << self
  def package(pkg)
    mod_name = pkg.to_s.capitalize

    unless GoRuby.const_defined?(mod_name)
      GoRuby.module_eval("#{mod_name} = Module.new { extend self }")
    end

    @__go_package__ = GoRuby.const_get(mod_name)
  end
end
```

```
# main.go.rb

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```


import "fmt"
⇒
def fmt
GoRuby::Fmt
end

```ruby
class << self
  def import(pkg)
    mod_name = pkg.camelize
    raise "unknown package #{pkg}" unless GoRuby.const_defined?(mod_name)

    define_method(pkg) { GoRuby.const_get(mod_name) }
  end
end
```

```
# main.go.rb

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

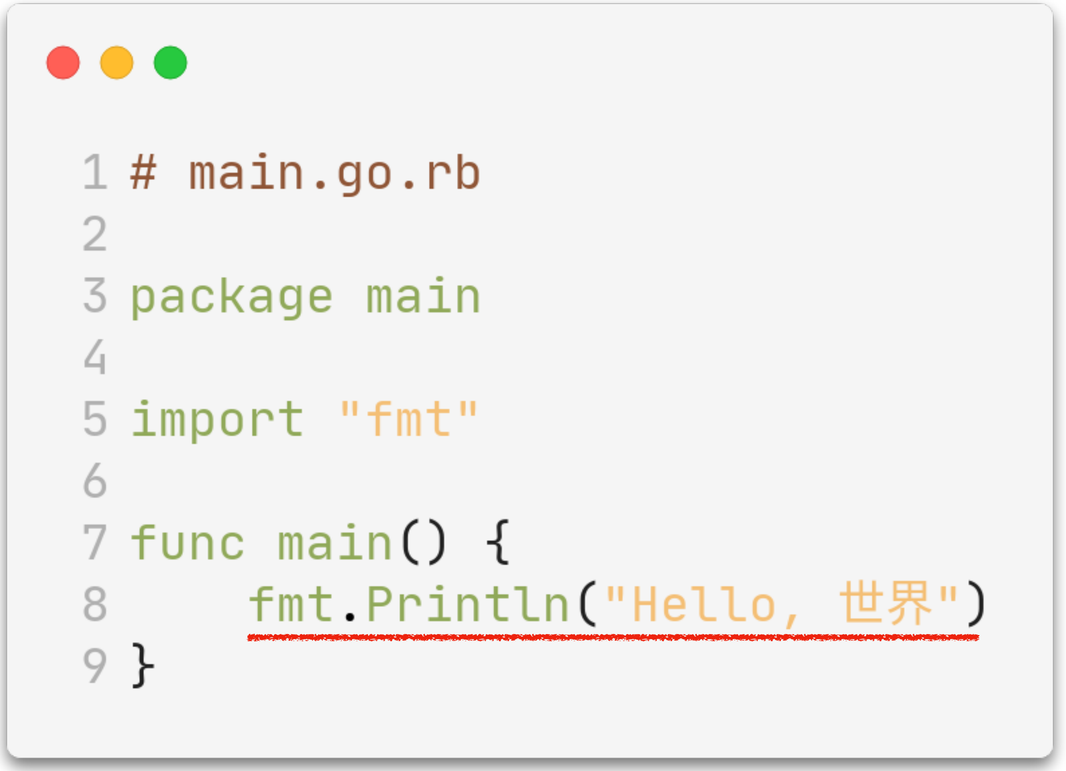
```
func main() {
  fmt.PrintLn("Hello, 世界")
}

# ⇒ ?

# 1 block goes to foo
func(main()) {
  fmt.PrintLn("Hello, 世界")
}

# 2 block goes to main
func(main() {
  fmt.PrintLn("Hello, 世界")
})
```

```ruby
class << self
  def method_missing(name, *args, &block)
    if block
      [name, block.to_proc]
    else
      name
    end
  end
end
```

One method_missing to rule them all

```
func main() {
  fmt.PrintLn("Hello, 世界")
}

# ⇒

module GoRuby
  module Main
    def main
      fmt.PrintLn("Hello, 世界")
    end
  end
end
```

```ruby
class << self
  def func(attrs)
    current_package = @__go_package__
    name, block = attrs

    if current_package.respond_to?(name)
      raise "#{name} already defined for package #{current_package}"
    end

    current_package.module_eval { define_method(name, block) }
  end
end
```



```
# main.go.rb

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

```ruby
module GoRuby
  module Fmt
    class << self
      def Println(*attrs)
        str = "#{attrs.join(' ')}\n"
        $stdout << str
        [str.bytesize, nil]
      end
    end
  end
end
```

```ruby
# main.go.rb

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Go, Ruby, Go!

```
$ ruby -r'./lib/go_ruby.rb' main.go.rb
```

```
# main.go.rb

package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```


1 at_exit do
2   GoRuby::Main.main
3 end

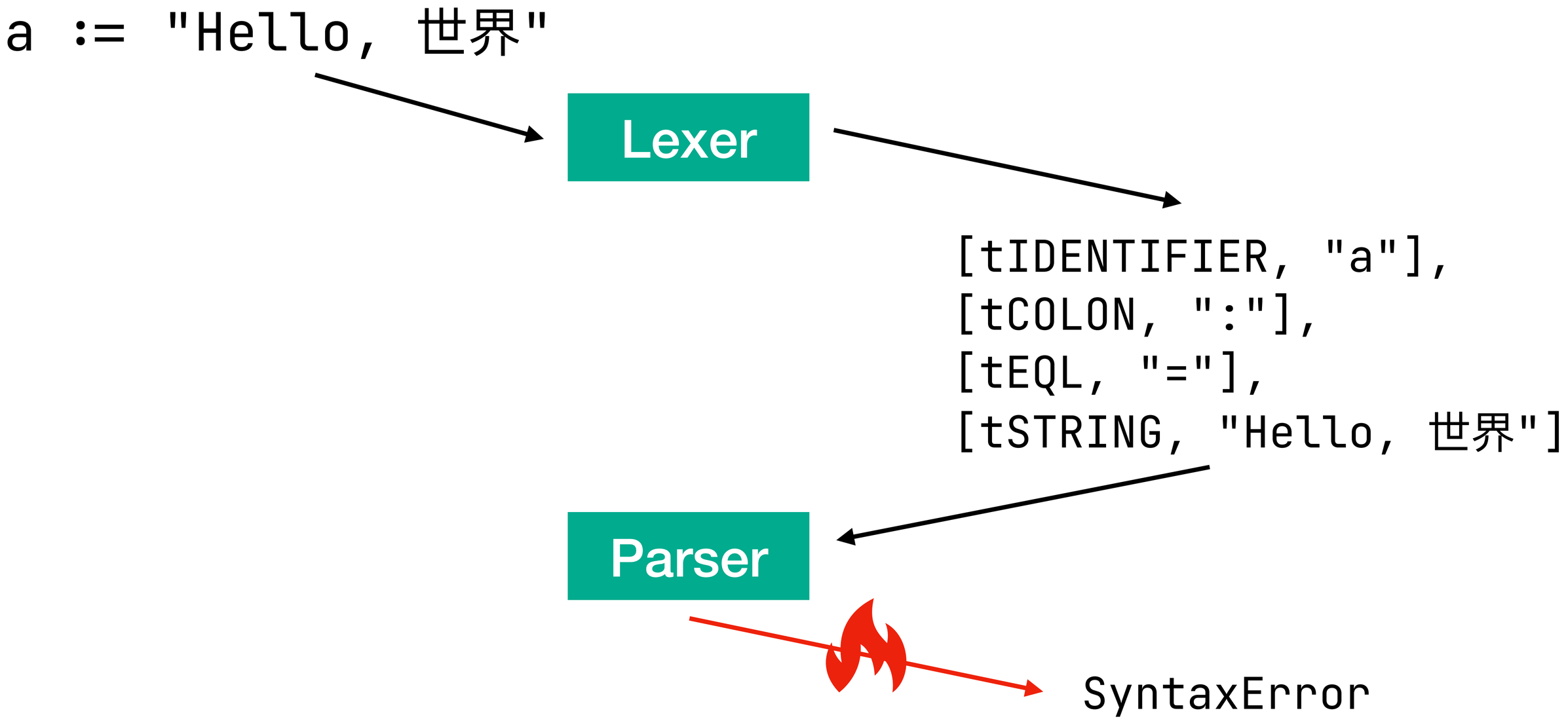
Go, Ruby, Go!

```
$ ruby -r'./lib/go_ruby.rb' main.go.rb
Hello, 世界
```


YOU'RE
WINNER !
WIN
WIN
risovach.ru

```ruby
# main.go.rb

package main

import "fmt"

func main() {
  str := "Hello, 世界"
  fmt.Println(str)
}
```

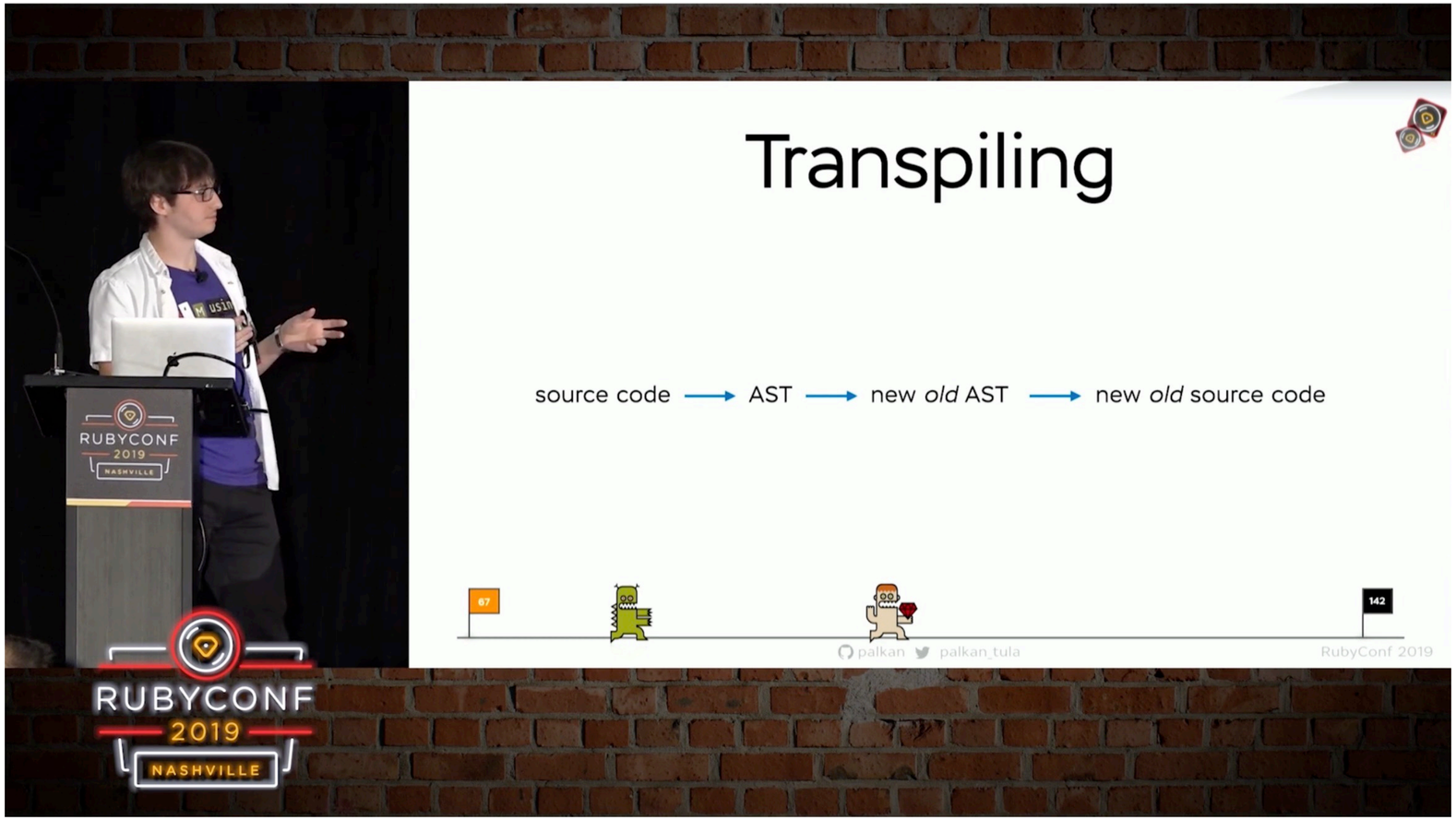
```
$ ruby -e "a := 1"

  -e:1: syntax error, unexpected '=', expecting literal content or
terminator or tSTRING_DBEG or tSTRING_DVAR
  a := 1
```

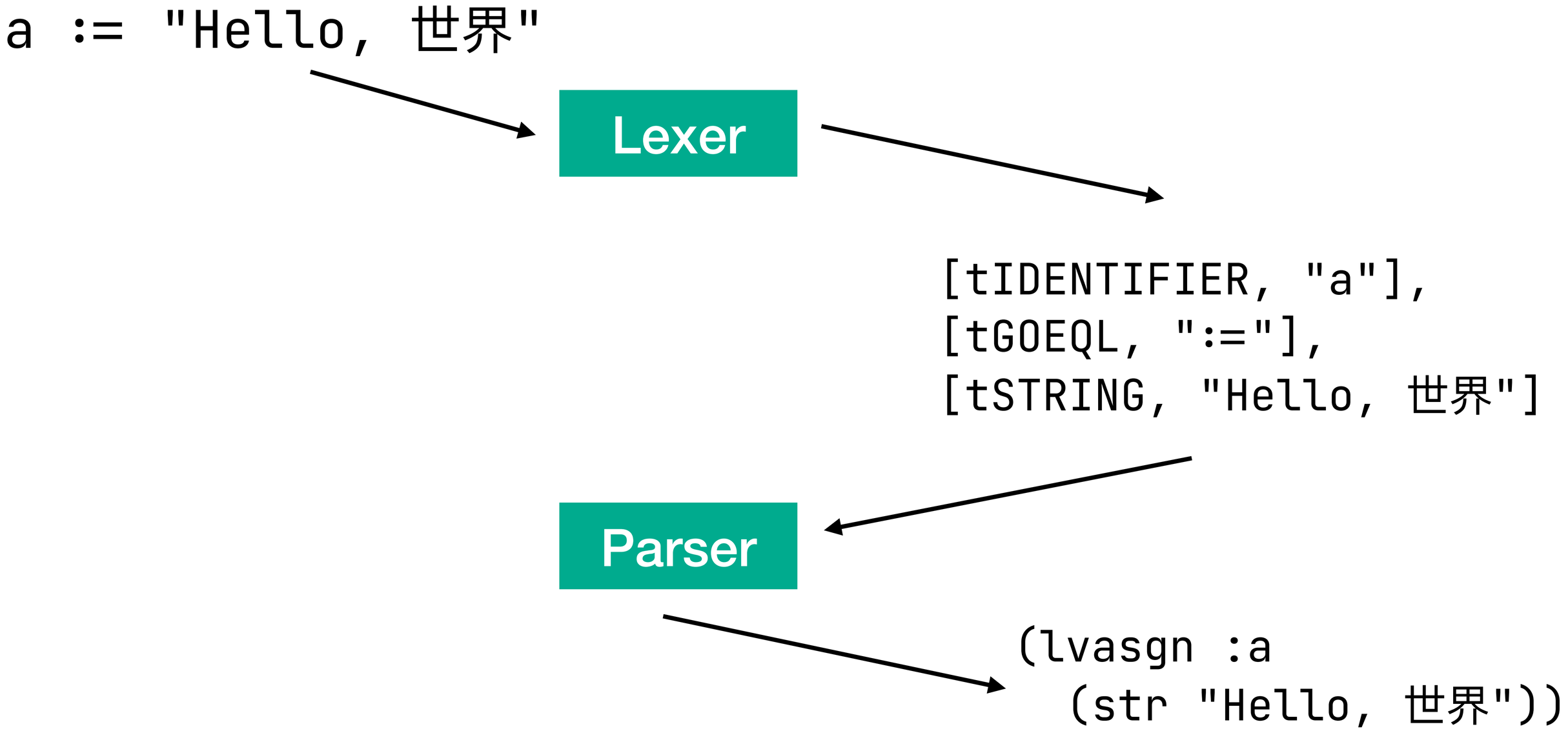

a := "Hello, 世界"
Lexer
[tIDENTIFIER, "a"],
[tCOLON, ":"],
[tEQL, "="],
[tSTRING, "Hello, 世界"]
Parser
SyntaxError

How to fix this?

- Ask the Core team to add `:=`
- Fork Ruby
- Juggle AST like it's 1999

Coming up next, ruby-next



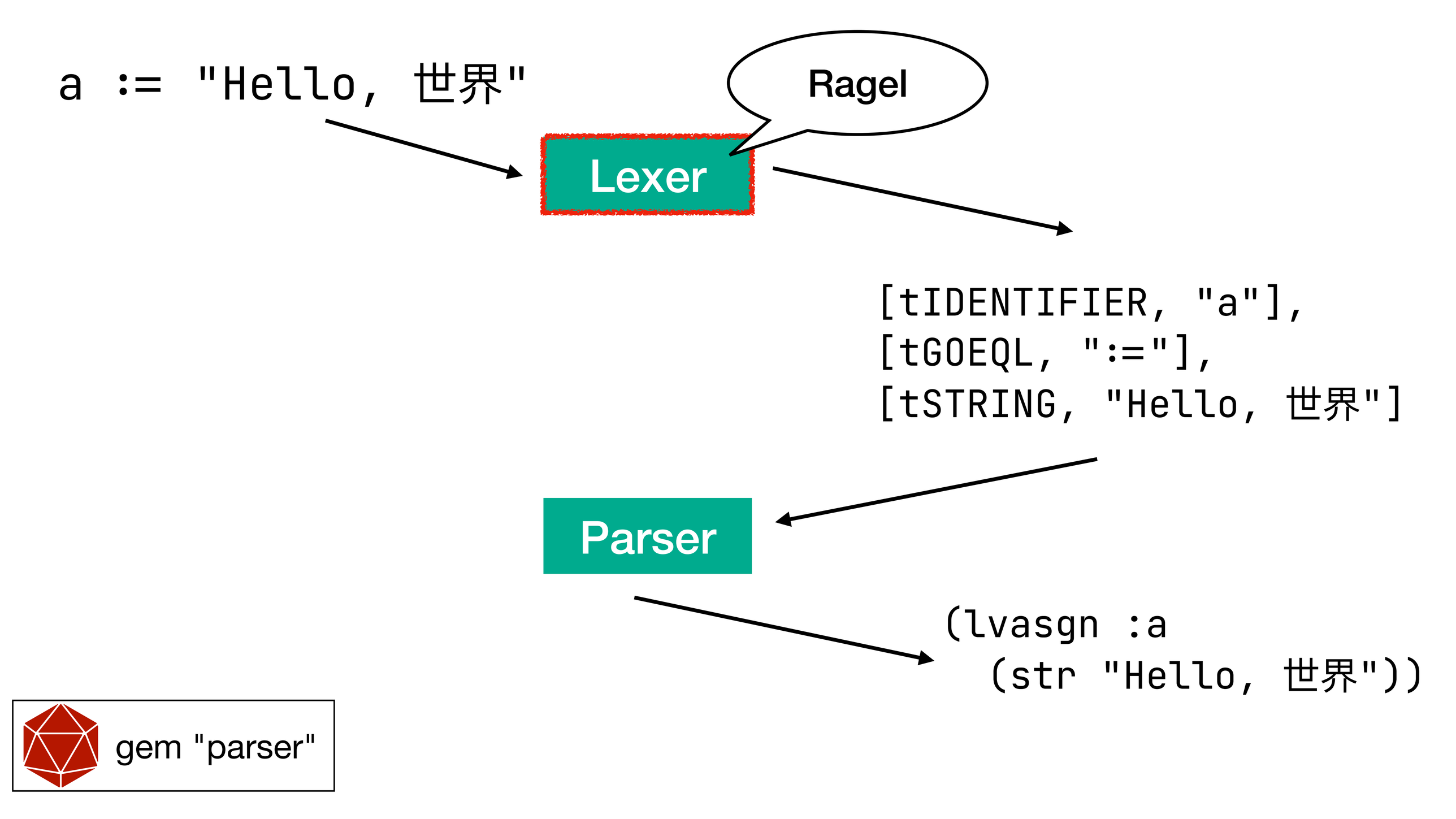

a := "Hello, 世界"
Lexer
[tIDENTIFIER, "a"],
[tGOEQL, ":="],
[tSTRING, "Hello, 世界"]
Parser
(lvasgn :a
(str "Hello, 世界"))

```
a := "Hello, 世界"
```


```
[tIDENTIFIER, "a"],
[tGOEQL, ":="],
[tSTRING, "Hello, 世界"]
```


```
(lvasgn :a
  (str "Hello, 世界"))
```

gem "parser"


a := "Hello, 世界"
Ragel
Lexer
[tIDENTIFIER, "a"],
[tGOEQL, ":="],
[tSTRING, "Hello, 世界"]
Parser
(lvasgn :a
(str "Hello, 世界"))
gem "parser"


Serious Shchi

Ragel State Machine Compiler

- Finite-state machine compiler and a parser generator.
- Works with a lot of languages (C, C++, Java, Go, Ruby, etc.)


$ wc -l lib/parser/lexer.rl
2556 lib/parser/lexer.rl
$ ragel -F1 -R lib/parser/lexer.rl -o lib/parser/lexer.rb
$ wc -l lib/parser/lexer.rb
23779 lib/parser/lexer.rb

```
  attr_reader   :source_buffer

  attr_accessor :diagnostics
  attr_accessor :static_env
  attr_accessor :force_utf32

  attr_accessor :cond, :cmdarg, :in_kwarg, :context, :command_start

  attr_accessor :tokens, :comments

  attr_reader :paren_nest, :cmdarg_stack, :cond_stack, :lambda_stack

  def initialize(version)
    @version    = version
    @static_env = nil
    @context    = nil

    @tokens     = nil
    @comments   = nil

    reset
  end

  def reset(reset_state=true)
    # Ragel state:
    if reset_state
      # Unit tests set state prior to resetting lexer.
      @cs     = self.class.lex_en_line_begin

      @cond   = StackState.new('cond')
      @cmdarg = StackState.new('cmdarg')
      @cond_stack   = []
      @cmdarg_stack = []
    end

    @force_utf32   = false # Set to true by some tests

    @source_pts    = nil # @source as a codepoint array

    @p             = 0   # stream position (saved manually in #advance)
    @ts            = nil # token start
    @te            = nil # token end
    @act           = 0   # next action

    @stack         = []  # state stack
    @top           = 0   # state stack top pointer

    # Lexer state:
    @token_queue   = []
    @literal_stack = []

    @eq_begin_s    = nil # location of last encountered =begin
    @sharp_s       = nil # location of last encountered #

    @newline_s     = nil # location of last encountered newline

    @num_base      = nil # last numeric base
    @num_digits_s  = nil # starting position of numeric digits
    @num_suffix_s  = nil # starting position of numeric suffix
    @num_xfrm      = nil # numeric suffix-induced transformation

    @escape_s      = nil # starting position of current sequence
    @escape        = nil # last escaped sequence, as string

    @herebody_s    = nil # starting position of current heredoc line

    # Ruby 1.9 ->() lambdas emit a distinct token if do/{ is
    # encountered after a matching closing parenthesis.
    @paren_nest    = 0
    @lambda_stack  = []

    # After encountering the closing line of <<~SQUIGGLY_HEREDOC,
    # we store the indentation level and give it out to the parser
    # on request. It is not possible to infer indentation level just
    # from the AST because escape sequences such as `\ ` or `\t` are
    # expanded inside the lexer, but count as non-whitespace for
    # indentation purposes.
    @dedent_level  = nil

    # If the lexer is in `command state' (aka expr_value)
    # at the entry to #advance, it will transition to expr_cmdarg
    # instead of expr_arg at certain points.
    @command_start = true

    # True at the end of "def foo a:"
    @in_kwarg      = false

    # State before =begin / =end block comment
    @cs_before_block_comment = self.class.lex_en_line_begin
  end
```

```ruby
  attr_reader   :source_buffer

  attr_accessor :diagnostics
  attr_accessor :static_env
  attr_accessor :force_utf32

  attr_accessor :cond, :cmdarg, :in_kwarg, :context, :command_start

  attr_accessor :tokens, :comments

  attr_reader :paren_nest, :cmdarg_stack, :cond_stack, :lambda_stack

  def initialize(version)
    @version    = version
    @static_env = nil
    @context    = nil

    @tokens     = nil
    @comments   = nil

    reset
  end

  def reset(reset_state=true)
    # Ragel state:
    if reset_state
      # Unit tests set state prior to resetting lexer.
      @cs       = self.class.lex_en_line_begin

      @cond   = StackState.new('cond')
      @cmdarg = StackState.new('cmdarg')
      @cond_stack   = []
      @cmdarg_stack = []
    end

    @force_utf32   = false # Set to true by some tests

    @source_pts    = nil # @source as a codepoint array

    @p             = 0   # stream position (saved manually in #advance)
    @ts            = nil # token start
    @te            = nil # token end
    @act           = 0   # next action

    @stack         = []  # state stack
    @top           = 0   # state stack top pointer

    # Lexer state:
    @token_queue   = []
    @literal_stack = []

    @eq_begin_s    = nil # location of last encountered =begin
    @sharp_s       = nil # location of last encountered #

    @newline_s     = nil # location of last encountered newline

    @num_base      = nil # last numeric base
    @num_digits_s  = nil # starting position of numeric digits
    @num_suffix_s  = nil # starting position of numeric suffix
    @num_xfrm      = nil # numeric suffix-induced transformation

    @escape_s      = nil # starting position of current sequence
    @escape        = nil # last escaped sequence, as string

    @herebody_s    = nil # starting position of current heredoc line

    # Ruby 1.9 ->() lambdas emit a distinct token if do/{ is
    # encountered after a matching closing parenthesis.
    @paren_nest    = 0
    @lambda_stack  = []

    # After encountering the closing line of <<~SQUIGGLY_HEREDOC,
    # we store the indentation level and give it out to the parser
    # on request. It is not possible to infer indentation level just
    # from the AST because escape sequences such as `\ ` or `\t` are
    # expanded inside the lexer, but count as non-whitespace for
    # indentation purposes.
    @dedent_level  = nil

    # If the lexer is in `command state' (aka expr_value)
    # at the entry to #advance, it will transition to expr_cmdarg
    # instead of expr_arg at certain points.
    @command_start = true

    # True at the end of "def foo a:"
    @in_kwarg      = false

    # State before =begin / =end block comment
    @cs_before_block_comment = self.class.lex_en_line_begin
  end
```

```ruby
  def eof_codepoint?(point)
    [0x04, 0x1a, 0x00].include? point
  end

  def version?(*versions)
    versions.include?(@version)
  end

  def stack_pop
    @top -= 1
    @stack[@top]
  end

  def encode_escape(ord)
    ord.chr.force_encoding(@source_buffer.source.encoding)
  end

  def tok(s = @ts, e = @te)
    @source_buffer.slice(s...e)
  end

  def range(s = @ts, e = @te)
    Parser::Source::Range.new(@source_buffer, s, e)
  end

  def emit(type, value = tok, s = @ts, e = @te)
    token = [ type, [ value, range(s, e) ] ]

    @token_queue.push(token)

    @tokens.push(token) if @tokens

    token
  end

  def emit_table(table, s = @ts, e = @te)
    value = tok(s, e)

    emit(table[value], value, s, e)
  end

  def emit_do(do_block=false)
    if @cond.active?
      emit(:kDO_COND, 'do'.freeze)
    elsif @cmdarg.active? || do_block
      emit(:kDO_BLOCK, 'do'.freeze)
    else
      emit(:kDO, 'do'.freeze)
    end
  end

  def arg_or_cmdarg(cmd_state)
    if cmd_state
      self.class.lex_en_expr_cmdarg
    else
      self.class.lex_en_expr_arg
    end
  end

  def emit_comment(s = @ts, e = @te)
    if @comments
      @comments.push(Parser::Source::Comment.new(range(s, e)))
    end

    if @tokens
      @tokens.push([ :tCOMMENT, [ tok(s, e), range(s, e) ] ])
    end

    nil
  end

  def diagnostic(type, reason, arguments=nil, location=range, highlights=[])
    @diagnostics.process(
        Parser::Diagnostic.new(type, reason, arguments, location,
highlights))
  end

  #
  # === LITERAL STACK ===
  #

  def push_literal(*args)
    new_literal = Literal.new(self, *args)
    @literal_stack.push(new_literal)
    next_state_for_literal(new_literal)
  end

  def next_state_for_literal(literal)
    if literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited_words
      else
        self.class.lex_en_plain_backslash_delimited_words
      end
    elsif literal.words? && !literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_words
      else
        self.class.lex_en_plain_words
      end
    elsif !literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited
      else
        self.class.lex_en_plain_backslash_delimited
      end
    else
      if literal.interpolate?
        self.class.lex_en_interp_string
      else
        self.class.lex_en_plain_string
      end
    end
  end

  def literal
    @literal_stack.last
  end

  def pop_literal
    old_literal = @literal_stack.pop

    @dedent_level = old_literal.dedent_level

    if old_literal.type == :tREGEXP_BEG
      # Fetch modifiers.
      self.class.lex_en_regexp_modifiers
    else
      self.class.lex_en_expr_end
    end
```

```
  attr_reader   :source_buffer

  attr_accessor :diagnostics
  attr_accessor :static_env
  attr_accessor :force_utf32

  attr_accessor :cond, :cmdarg, :in_kwarg, :context, :command_start

  attr_accessor :tokens, :comments

  attr_reader :paren_nest, :cmdarg_stack, :cond_stack, :lambda_stack

  def initialize(version)
    @version    = version
    @static_env = nil
    @context    = nil

    @tokens     = nil
    @comments   = nil

    reset
  end

  def reset(reset_state=true)
    # Ragel state:
    if reset_state
      # Unit tests set state prior to resetting lexer.
      @cs     = self.class.lex_en_line_begin

      @cond   = StackState.new('cond')
      @cmdarg = StackState.new('cmdarg')
      @cond_stack   = []
      @cmdarg_stack = []
    end

    @force_utf32   = false # Set to true by some tests

    @source_pts    = nil # @source as a codepoint array

    @p             = 0   # stream position (saved manually in #advance)
    @ts            = nil # token start
    @te            = nil # token end
    @act           = 0   # next action

    @stack         = []  # state stack
    @top           = 0   # state stack top pointer

    # Lexer state:
    @token_queue   = []
    @literal_stack = []

    @eq_begin_s    = nil # location of last encountered =begin
    @sharp_s       = nil # location of last encountered #

    @newline_s     = nil # location of last encountered newline

    @num_base      = nil # last numeric base
    @num_digits_s  = nil # starting position of numeric digits
    @num_suffix_s  = nil # starting position of numeric suffix
    @num_xfrm      = nil # numeric suffix-induced transformation

    @escape_s      = nil # starting position of current sequence
    @escape        = nil # last escaped sequence, as string

    @herebody_s    = nil # starting position of current heredoc line

    # Ruby 1.9 ->() lambdas emit a distinct token if do/{ is
    # encountered after a matching closing parenthesis.
    @paren_nest    = 0
    @lambda_stack  = []

    # After encountering the closing line of <<~SQUIGGLY_HEREDOC,
    # we store the indentation level and give it out to the parser
    # on request. It is not possible to infer indentation level just
    # from the AST because escape sequences such as `\ ` or `\t` are
    # expanded inside the lexer, but count as non-whitespace for
    # indentation purposes.
    @dedent_level  = nil

    # If the lexer is in `command state' (aka expr_value)
    # at the entry to #advance, it will transition to expr_cmdarg
    # instead of expr_arg at certain points.
    @command_start = true

    # True at the end of "def foo a:"
    @in_kwarg      = false

    # State before =begin / =end block comment
    @cs_before_block_comment = self.class.lex_en_line_begin
  end
```

```
  def eof_codepoint?(point)
    [0x04, 0x1a, 0x00].include? point
  end

  def version?(*versions)
    versions.include?(@version)
  end

  def stack_pop
    @top -= 1
    @stack[@top]
  end

  def encode_escape(ord)
    ord.chr.force_encoding(@source_buffer.source.encoding)
  end

  def tok(s = @ts, e = @te)
    @source_buffer.slice(s...e)
  end

  def range(s = @ts, e = @te)
    Parser::Source::Range.new(@source_buffer, s, e)
  end

  def emit(type, value = tok, s = @ts, e = @te)
    token = [ type, [ value, range(s, e) ] ]

    @token_queue.push(token)

    @tokens.push(token) if @tokens

    token
  end

  def emit_table(table, s = @ts, e = @te)
    value = tok(s, e)

    emit(table[value], value, s, e)
  end

  def emit_do(do_block=false)
    if @cond.active?
      emit(:kDO_COND, 'do'.freeze)
    elsif @cmdarg.active? || do_block
      emit(:kDO_BLOCK, 'do'.freeze)
    else
      emit(:kDO, 'do'.freeze)
    end
  end

  def arg_or_cmdarg(cmd_state)
    if cmd_state
      self.class.lex_en_expr_cmdarg
    else
      self.class.lex_en_expr_arg
    end
  end

  def emit_comment(s = @ts, e = @te)
    if @comments
      @comments.push(Parser::Source::Comment.new(range(s, e)))
    end

    if @tokens
      @tokens.push([ :tCOMMENT, [ tok(s, e), range(s, e) ] ])
    end

    nil
  end

  def diagnostic(type, reason, arguments=nil, location=range, highlights=[])
    @diagnostics.process(
        Parser::Diagnostic.new(type, reason, arguments, location,
highlights))
  end

  #
  # === LITERAL STACK ===
  #

  def push_literal(*args)
    new_literal = Literal.new(self, *args)
    @literal_stack.push(new_literal)
    next_state_for_literal(new_literal)
  end

  def next_state_for_literal(literal)
    if literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited_words
      else
        self.class.lex_en_plain_backslash_delimited_words
      end
    elsif literal.words? && !literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_words
      else
        self.class.lex_en_plain_words
      end
    elsif !literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited
      else
        self.class.lex_en_plain_backslash_delimited
      end
    else
      if literal.interpolate?
        self.class.lex_en_interp_string
      else
        self.class.lex_en_plain_string
      end
    end
  end

  def literal
    @literal_stack.last
  end

  def pop_literal
    old_literal = @literal_stack.pop

    @dedent_level = old_literal.dedent_level

    if old_literal.type == :tREGEXP_BEG
      # Fetch modifiers.
      self.class.lex_en_regexp_modifiers
    else
      self.class.lex_en_expr_end
    end
  end
```

```
# Mapping of strings to parser tokens.

PUNCTUATION = {
  '='   => :tEQL,      '&'   => :tAMPER2,  '|'   => :tPIPE,
  '!'   => :tBANG,     '^'   => :tCARET,   '+'   => :tPLUS,
  '-'   => :tMINUS,    '*'   => :tSTAR2,   '/'   => :tDIVIDE,
  '%'   => :tPERCENT,  '~'   => :tTILDE,   ','   => :tCOMMA,
  ';'   => :tSEMI,     '.'   => :tDOT,     '..'  => :tDOT2,
  '...' => :tDOT3,     '['   => :tLBRACK2, ']'   => :tRBRACK,
  '('   => :tLPAREN2, ')'   => :tRPAREN,  '?'   => :tEH,
  ':'   => :tCOLON,    '&&'  => :tANDOP,   '||'  => :tOROP,
  '-@'  => :tUMINUS,   '+@'  => :tUPLUS,   '~@'  => :tTILDE,
  '**'  => :tPOW,      '->'  => :tLAMBDA,  '=~'  => :tMATCH,
  '!~'  => :tNMATCH,   '=='  => :tEQ,      '!='  => :tNEQ,
  '>'   => :tGT,       '>>'  => :tRSHFT,   '>='  => :tGEQ,
  '<'   => :tLT,       '<<'  => :tLSHFT,   '<='  => :tLEQ,
  '=>'  => :tASSOC,    '::'  => :tCOLON2,  '===' => :tEQQ,
  '<=>' => :tCMP,      '[]'  => :tAREF,    '[]=' => :tASET,
  '{'   => :tLCURLY,   '}'   => :tRCURLY,  '`'   => :tBACK_REF2,
  '!@'  => :tBANG,     '&.'  => :tANDDOT,
}

PUNCTUATION_BEGIN = {
  '&'   => :tAMPER,    '*'   => :tSTAR,    '**'  => :tDSTAR,
  '+'   => :tUPLUS,    '-'   => :tUMINUS,  '::'  => :tCOLON3,
  '('   => :tLPAREN,   '{'   => :tLBRACE,  '['   => :tLBRACK,
}

KEYWORDS = {
  'if'     => :kIF_MOD,      'unless'   => :kUNLESS_MOD,
  'while'  => :kWHILE_MOD,   'until'    => :kUNTIL_MOD,
  'rescue' => :kRESCUE_MOD,  'defined?' => :kDEFINED,
  'BEGIN'  => :klBEGIN,      'END'      => :klEND,
}

KEYWORDS_BEGIN = {
  'if'     => :kIF,          'unless'   => :kUNLESS,
  'while'  => :kWHILE,       'until'    => :kUNTIL,
  'rescue' => :kRESCUE,      'defined?' => :kDEFINED,
  'BEGIN'  => :klBEGIN,      'END'      => :klEND,
}

%w(class module def undef begin end then elsif else ensure case when
   for break next redo retry in do return yield super self nil true
   false and or not alias __FILE__ __LINE__ __ENCODING__).each do
|keyword|
  KEYWORDS_BEGIN[keyword] = KEYWORDS[keyword] = :"k#{keyword.upcase}"
end
```

```
  attr_reader   :source_buffer

  attr_accessor :diagnostics
  attr_accessor :static_env
  attr_accessor :force_utf32

  attr_accessor :cond, :cmdarg, :in_kwarg, :context, :command_start

  attr_accessor :tokens, :comments

  attr_reader :paren_nest, :cmdarg_stack, :cond_stack, :lambda_stack

  def initialize(version)
    @version    = version
    @static_env = nil
    @context    = nil

    @tokens     = nil
    @comments   = nil

    reset
  end

  def reset(reset_state=true)
    # Ragel state:
    if reset_state
      # Unit tests set state prior to resetting lexer.
      @cs     = self.class.lex_en_line_begin

      @cond   = StackState.new('cond')
      @cmdarg = StackState.new('cmdarg')
      @cond_stack   = []
      @cmdarg_stack = []
    end

    @force_utf32   = false # Set to true by some tests

    @source_pts    = nil # @source as a codepoint array

    @p             = 0   # stream position (saved manually in #advance)
    @ts            = nil # token start
    @te            = nil # token end
    @act           = 0   # next action

    @stack         = []  # state stack
    @top           = 0   # state stack top pointer

    # Lexer state:
    @token_queue   = []
    @literal_stack = []

    @eq_begin_s    = nil # location of last encountered =begin
    @sharp_s       = nil # location of last encountered #

    @newline_s     = nil # location of last encountered newline

    @num_base      = nil # last numeric base
    @num_digits_s  = nil # starting position of numeric digits
    @num_suffix_s  = nil # starting position of numeric suffix
    @num_xfrm      = nil # numeric suffix-induced transformation

    @escape_s      = nil # starting position of current sequence
    @escape        = nil # last escaped sequence, as string

    @herebody_s    = nil # starting position of current heredoc line

    # Ruby 1.9 ->() lambdas emit a distinct token if do/{ is
    # encountered after a matching closing parenthesis.
    @paren_nest    = 0
    @lambda_stack  = []

    # After encountering the closing line of <<~SQUIGGLY_HEREDOC,
    # we store the indentation level and give it out to the parser
    # on request. It is not possible to infer indentation level just
    # from the AST because escape sequences such as `\ ` or `\t` are
    # expanded inside the lexer, but count as non-whitespace for
    # indentation purposes.
    @dedent_level  = nil

    # If the lexer is in `command state' (aka expr_value)
    # at the entry to #advance, it will transition to expr_cmdarg
    # instead of expr_arg at certain points.
    @command_start = true

    # True at the end of "def foo a:"
    @in_kwarg      = false

    # State before =begin / =end block comment
    @cs_before_block_comment = self.class.lex_en_line_begin
  end
```

```
  def eof_codepoint?(point)
    [0x04, 0x1a, 0x00].include? point
  end

  def version?(*versions)
    versions.include?(@version)
  end

  def stack_pop
    @top -= 1
    @stack[@top]
  end

  def encode_escape(ord)
    ord.chr.force_encoding(@source_buffer.source.encoding)
  end

  def tok(s = @ts, e = @te)
    @source_buffer.slice(s...e)
  end

  def range(s = @ts, e = @te)
    Parser::Source::Range.new(@source_buffer, s, e)
  end

  def emit(type, value = tok, s = @ts, e = @te)
    token = [ type, [ value, range(s, e) ] ]

    @token_queue.push(token)

    @tokens.push(token) if @tokens

    token
  end

  def emit_table(table, s = @ts, e = @te)
    value = tok(s, e)

    emit(table[value], value, s, e)
  end

  def emit_do(do_block=false)
    if @cond.active?
      emit(:kDO_COND, 'do'.freeze)
    elsif @cmdarg.active? || do_block
      emit(:kDO_BLOCK, 'do'.freeze)
    else
      emit(:kDO, 'do'.freeze)
    end
  end

  def arg_or_cmdarg(cmd_state)
    if cmd_state
      self.class.lex_en_expr_cmdarg
    else
      self.class.lex_en_expr_arg
    end
  end

  def emit_comment(s = @ts, e = @te)
    if @comments
      @comments.push(Parser::Source::Comment.new(range(s, e)))
    end

    if @tokens
      @tokens.push([ :tCOMMENT, [ tok(s, e), range(s, e) ] ])
    end

    nil
  end

  def diagnostic(type, reason, arguments=nil, location=range, highlights=[])
    @diagnostics.process(
        Parser::Diagnostic.new(type, reason, arguments, location,
highlights))
  end

  #
  # === LITERAL STACK ===
  #

  def push_literal(*args)
    new_literal = Literal.new(self, *args)
    @literal_stack.push(new_literal)
    next_state_for_literal(new_literal)
  end

  def next_state_for_literal(literal)
    if literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited_words
      else
        self.class.lex_en_plain_backslash_delimited_words
      end
    elsif literal.words? && !literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_words
      else
        self.class.lex_en_plain_words
      end
    elsif !literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited
      else
        self.class.lex_en_plain_backslash_delimited
      end
    else
      if literal.interpolate?
        self.class.lex_en_interp_string
      else
        self.class.lex_en_plain_string
      end
    end
  end

  def literal
    @literal_stack.last
  end

  def pop_literal
    old_literal = @literal_stack.pop

    @dedent_level = old_literal.dedent_level

    if old_literal.type == :tREGEXP_BEG
      # Fetch modifiers.
      self.class.lex_en_regexp_modifiers
    else
      self.class.lex_en_expr_end
    end
```

```
# Mapping of strings to parser tokens.

PUNCTUATION = {
  '='   => :tEQL,      '&'   => :tAMPER2,  '|'   => :tPIPE,
  '!'   => :tBANG,     '^'   => :tCARET,   '+'   => :tPLUS,
  '-'   => :tMINUS,    '*'   => :tSTAR2,   '/'   => :tDIVIDE,
  '%'   => :tPERCENT,  '~'   => :tTILDE,   ','   => :tCOMMA,
  ';'   => :tSEMI,     '.'   => :tDOT,     '..'  => :tDOT2,
  '...' => :tDOT3,     '['   => :tLBRACK2, ']'   => :tRBRACK,
  '('   => :tLPAREN2, ')'   => :tRPAREN,  '?'   => :tEH,
  ':'   => :tCOLON,    '&&'  => :tANDOP,   '||'  => :tOROP,
  '-@'  => :tUMINUS,   '+@'  => :tUPLUS,   '~@'  => :tTILDE,
  '**'  => :tPOW,      '->'  => :tLAMBDA,  '=~'  => :tMATCH,
  '!~'  => :tNMATCH,   '=='  => :tEQ,      '!='  => :tNEQ,
  '>'   => :tGT,       '>>'  => :tRSHFT,   '>='  => :tGEQ,
  '<'   => :tLT,       '<<'  => :tLSHFT,   '<='  => :tLEQ,
  '=>'  => :tASSOC,    '::'  => :tCOLON2,  '===' => :tEQQ,
  '<=>' => :tCMP,      '[]'  => :tAREF,    '[]=' => :tASET,
  '{'   => :tLCURLY,   '}'   => :tRCURLY,  '`'   => :tBACK_REF2,
  '!@'  => :tBANG,     '&.'  => :tANDDOT,
}

PUNCTUATION_BEGIN = {
  '&'   => :tAMPER,    '*'   => :tSTAR,    '**'  => :tDSTAR,
  '+'   => :tUPLUS,    '-'   => :tUMINUS,  '::'  => :tCOLON3,
  '('   => :tLPAREN,   '{'   => :tLBRACE,  '['   => :tLBRACK,
}

KEYWORDS = {
  'if'      => :kIF_MOD,      'unless'   => :kUNLESS_MOD,
  'while'   => :kWHILE_MOD,   'until'    => :kUNTIL_MOD,
  'rescue'  => :kRESCUE_MOD,  'defined?' => :kDEFINED,
  'BEGIN'   => :klBEGIN,      'END'      => :klEND,
}

KEYWORDS_BEGIN = {
  'if'      => :kIF,          'unless'   => :kUNLESS,
  'while'   => :kWHILE,       'until'    => :kUNTIL,
  'rescue'  => :kRESCUE,      'defined?' => :kDEFINED,
  'BEGIN'   => :klBEGIN,      'END'      => :klEND,
}

%w(class module def undef begin end then elsif else ensure case when
   for break next redo retry in do return yield super self nil true
   false and or not alias __FILE__ __LINE__ __ENCODING__).each do
|keyword|
  KEYWORDS_BEGIN[keyword] = KEYWORDS[keyword] = :"k#{keyword.upcase}"
end
```

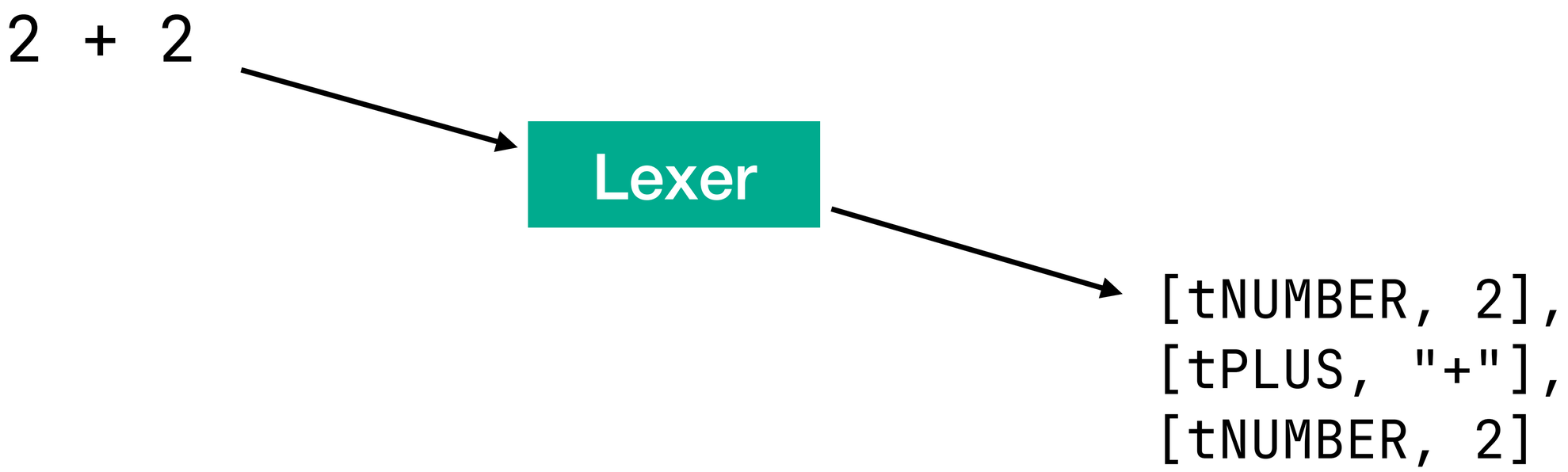
```
# === CHARACTER CLASSES ===
#
# Pay close attention to the differences between c_any and any.
# c_any does not include EOF and so will cause incorrect behavior
# for machine subtraction (any-except rules) and default transitions
# for scanners.

action do_nl {
  # Record position of a newline for precise location reporting on tNL
  # tokens.
  #
  # This action is embedded directly into c_nl, as it is idempotent and
  # there are no cases when we need to skip it.
  @newline_s = p
}

c_nl       = '\n' $ do_nl;
c_space    = [ \t\r\f\v];
c_space_nl = c_space | c_nl;

c_eof      = 0x04 | 0x1a | 0 | zlen; # ^D, ^Z, \0, EOF
c_eol      = c_nl | c_eof;
c_any      = any - c_eof;

c_nl_zlen  = c_nl | zlen;
c_line     = any - c_nl_zlen;

c_unicode  = c_any - 0x00..0x7f;
c_upper    = [A-Z];
c_lower    = [a-z_]  | c_unicode;
c_alpha    = c_lower | c_upper;
c_alnum    = c_alpha | [0-9];

action do_eof {
  # Sit at EOF indefinitely. #advance would return $eof each time.
  # This allows to feed the lexer more data if needed; this is only used
  # in tests.
  #
  # Note that this action is not embedded into e_eof like e_heredoc_nl and
e_bs
  # below. This is due to the fact that scanner state at EOF is observed
  # by tests, and encapsulating it in a rule would break the
introspection.
  fhold; fbreak;
}

#
# === TOKEN DEFINITIONS ===
#

# All operators are punctuation. There is more to punctuation
# than just operators. Operators can be overridden by user;
# punctuation can not.

# A list of operators which are valid in the function name context, but
# have different semantics in others.
operator_fname      = '[]' | '[]=' | '`'  | '+@' | '-@' | '~@' | '!@' ;

# A list of operators which can occur within an assignment shortcut (+ →
+=).
operator_arithmetic = '&'  | '|'  | '&&' | '||' | '^'  | '+'  | '-'  |
                      '*'  | '/'  | '%'  | '**' | '~'  | '<<' | '>>' | '&.' ;

# A list of all user-definable operators not covered by groups above.
operator_rest       = '=~' | '!~' | '=='  | '===' | '!'  | '<=>' |
                      '<'  | '<=' | '>'  | '>=' | '<=>' | '=>' ;

# Note that `{` and `}` need to be referred to as e_lbrace and e_rbrace,
# as they are ambiguous with interpolation `#{}` and should be counted.
# These braces are not present in punctuation lists.

# A list of punctuation which has different meaning when used at the
# beginning of expression.
punctuation_begin   = '-'  | '+'  | '::' | '(' | '[' |
                      '*'  | '**' | '&' ;

# A list of all punctuation except punctuation_begin.
punctuation_end     = ','  | '='  | '->' | '(' | '[' | ']' |
                      '::' | '?'  | ':'  | '.' | '..' | '...' ;

# A list of keywords which have different meaning at the beginning of
expression.
keyword_modifier    = 'if'     | 'unless' | 'while'  | 'until' | 'rescue'
;

# A list of keywords which accept an argument-like expression, i.e. have
the
# same post-processing as method calls or commands. Example: `yield 1`,
# `yield(1)`, `yield(1)*2`, are interpreted as if `yield` was a function.
keyword_with_arg    = 'yield'  | 'super'  | 'not'    | 'defined?' ;

# A list of keywords which accept a literal function name as an argument.
keyword_with_fname  = 'def'    | 'undef'  | 'alias' ;

# A list of keywords which accept an expression after them.
keyword_with_value  = 'else'   | 'case'   | 'ensure' | 'module' | 'elsif'
| 'then'   |
                      'for'    | 'in'     | 'do'     | 'when'   | 'begin'
| 'class' |
                      'and'    | 'or'     ;

# A list of keywords which accept a value, and treat the keywords from
# `keyword_modifier` list as modifiers.
keyword_with_mid    = 'rescue' | 'return' | 'break'  | 'next'   ;

# A list of keywords which do not accept an expression after them.
keyword_with_end    = 'end'    | 'self'   | 'true'   | 'false'  | 'retry'
  |
                      'redo'   | 'nil'    | 'BEGIN'  | 'END'    |
'__FILE__' |
                      '__LINE__' | '__ENCODING__';

# All keywords.
keyword             = keyword_with_value | keyword_with_mid |
                      keyword_with_end   | keyword_with_arg |
                      keyword_with_fname | keyword_modifier ;

constant       = c_upper c_alnum*;
bareword       = c_alpha c_alnum*;

call_or_var    = c_lower c_alnum*;
class_var      = '@@' bareword;
instance_var   = '@' bareword;
global_var     = '$'
    ( bareword | digit+
    | [ `~=*$?!@/\\;,.:<>"] #`
    | '-' c_alnum
    )
;

# Ruby accepts (and fails on) variables with leading digit
# in literal context, but not in unquoted symbol body.
class_var_v    = '@@' c_alnum+;
instance_var_v = '@' c_alnum+;

label          = bareword [?!]? ':';
```

```
  attr_reader   :source_buffer

  attr_accessor :diagnostics
  attr_accessor :static_env
  attr_accessor :force_utf32

  attr_accessor :cond, :cmdarg, :in_kwarg, :context, :command_start

  attr_accessor :tokens, :comments

  attr_reader :paren_nest, :cmdarg_stack, :cond_stack, :lambda_stack

  def initialize(version)
    @version    = version
    @static_env = nil
    @context    = nil

    @tokens     = nil
    @comments   = nil

    reset
  end

  def reset(reset_state=true)
    # Ragel state:
    if reset_state
      # Unit tests set state prior to resetting lexer.
      @cs       = self.class.lex_en_line_begin

      @cond   = StackState.new('cond')
      @cmdarg = StackState.new('cmdarg')
      @cond_stack   = []
      @cmdarg_stack = []
    end

    @force_utf32    = false # Set to true by some tests

    @source_pts     = nil # @source as a codepoint array

    @p              = 0   # stream position (saved manually in #advance)
    @ts             = nil # token start
    @te             = nil # token end
    @act            = 0   # next action

    @stack          = []  # state stack
    @top            = 0   # state stack top pointer

    # Lexer state:
    @token_queue    = []
    @literal_stack  = []

    @eq_begin_s     = nil # location of last encountered =begin
    @sharp_s        = nil # location of last encountered #

    @newline_s      = nil # location of last encountered newline

    @num_base       = nil # last numeric base
    @num_digits_s   = nil # starting position of numeric digits
    @num_suffix_s   = nil # starting position of numeric suffix
    @num_xfrm       = nil # numeric suffix-induced transformation

    @escape_s       = nil # starting position of current sequence
    @escape         = nil # last escaped sequence, as string

    @herebody_s     = nil # starting position of current heredoc line

    # Ruby 1.9 ->() lambdas emit a distinct token if do/{ is
    # encountered after a matching closing parenthesis.
    @paren_nest     = 0
    @lambda_stack   = []

    # After encountering the closing line of <<~SQUIGGLY_HEREDOC,
    # we store the indentation level and give it out to the parser
    # on request. It is not possible to infer indentation level just
    # from the AST because escape sequences such as `\ ` or `\t` are
    # expanded inside the lexer, but count as non-whitespace for
    # indentation purposes.
    @dedent_level   = nil

    # If the lexer is in `command state' (aka expr_value)
    # at the entry to #advance, it will transition to expr_cmdarg
    # instead of expr_arg at certain points.
    @command_start = true

    # True at the end of "def foo a:"
    @in_kwarg       = false

    # State before =begin / =end block comment
    @cs_before_block_comment = self.class.lex_en_line_begin
  end
```

```
  def eof_codepoint?(point)
    [0x04, 0x1a, 0x00].include? point
  end

  def version?(*versions)
    versions.include?(@version)
  end

  def stack_pop
    @top -= 1
    @stack[@top]
  end

  def encode_escape(ord)
    ord.chr.force_encoding(@source_buffer.source.encoding)
  end

  def tok(s = @ts, e = @te)
    @source_buffer.slice(s...e)
  end

  def range(s = @ts, e = @te)
    Parser::Source::Range.new(@source_buffer, s, e)
  end

  def emit(type, value = tok, s = @ts, e = @te)
    token = [ type, [ value, range(s, e) ] ]

    @token_queue.push(token)

    @tokens.push(token) if @tokens

    token
  end

  def emit_table(table, s = @ts, e = @te)
    value = tok(s, e)

    emit(table[value], value, s, e)
  end

  def emit_do(do_block=false)
    if @cond.active?
      emit(:kDO_COND, 'do'.freeze)
    elsif @cmdarg.active? || do_block
      emit(:kDO_BLOCK, 'do'.freeze)
    else
      emit(:kDO, 'do'.freeze)
    end
  end

  def arg_or_cmdarg(cmd_state)
    if cmd_state
      self.class.lex_en_expr_cmdarg
    else
      self.class.lex_en_expr_arg
    end
  end

  def emit_comment(s = @ts, e = @te)
    if @comments
      @comments.push(Parser::Source::Comment.new(range(s, e)))
    end

    if @tokens
      @tokens.push([ :tCOMMENT, [ tok(s, e), range(s, e) ] ])
    end

    nil
  end

  def diagnostic(type, reason, arguments=nil, location=range, highlights=[])
    @diagnostics.process(
        Parser::Diagnostic.new(type, reason, arguments, location,
highlights))
  end

  #
  # === LITERAL STACK ===
  #

  def push_literal(*args)
    new_literal = Literal.new(self, *args)
    @literal_stack.push(new_literal)
    next_state_for_literal(new_literal)
  end

  def next_state_for_literal(literal)
    if literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited_words
      else
        self.class.lex_en_plain_backslash_delimited_words
      end
    elsif literal.words? && !literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_words
      else
        self.class.lex_en_plain_words
      end
    elsif !literal.words? && literal.backslash_delimited?
      if literal.interpolate?
        self.class.lex_en_interp_backslash_delimited
      else
        self.class.lex_en_plain_backslash_delimited
      end
    else
      if literal.interpolate?
        self.class.lex_en_interp_string
      else
        self.class.lex_en_plain_string
      end
    end
  end

  def literal
    @literal_stack.last
  end

  def pop_literal
    old_literal = @literal_stack.pop

    @dedent_level = old_literal.dedent_level

    if old_literal.type == :tREGEXP_BEG
      # Fetch modifiers.
      self.class.lex_en_regexp_modifiers
    else
      self.class.lex_en_expr_end
    end
  end
```

```
# Mapping of strings to parser tokens.

PUNCTUATION = {
  '='   => :tEQL,     '&'   => :tAMPER2,  '|'   => :tPIPE,
  '!'   => :tBANG,    '^'   => :tCARET,   '+'   => :tPLUS,
  '-'   => :tMINUS,   '*'   => :tSTAR2,   '/'   => :tDIVIDE,
  '%'   => :tPERCENT, '~'   => :tTILDE,   ','   => :tCOMMA,
  ';'   => :tSEMI,    '.'   => :tDOT,     '..'  => :tDOT2,
  '...' => :tDOT3,    '['   => :tLBRACK2, ']'   => :tRBRACK,
  '('   => :tLPAREN2, ')'   => :tRPAREN,  '?'   => :tEH,
  ':'   => :tCOLON,   '&&'  => :tANDOP,   '||'  => :tOROP,
  '-@'  => :tUMINUS,  '+@'  => :tUPLUS,   '~@'  => :tTILDE,
  '**'  => :tPOW,     '->'  => :tLAMBDA,  '=~'  => :tMATCH,
  '!~'  => :tNMATCH,  '=='  => :tEQ,      '!='  => :tNEQ,
  '>'   => :tGT,      '>>'  => :tRSHFT,   '>='  => :tGEQ,
  '<'   => :tLT,      '<<'  => :tLSHFT,   '<='  => :tLEQ,
  '=>'  => :tASSOC,   '::'  => :tCOLON2,  '===' => :tEQQ,
  '<=>' => :tCMP,     '[]'  => :tAREF,    '[]=' => :tASET,
  '{'   => :tLCURLY,  '}'   => :tRCURLY,  '`'   => :tBACK_REF2,
  '!@'  => :tBANG,    '&.'  => :tANDDOT,
}

PUNCTUATION_BEGIN = {
  '&'   => :tAMPER,   '*'   => :tSTAR,    '**'  => :tDSTAR,
  '+'   => :tUPLUS,   '-'   => :tUMINUS,  '::'  => :tCOLON3,
  '('   => :tLPAREN,  '{'   => :tLBRACE,  '['   => :tLBRACK,
}

KEYWORDS = {
  'if'      => :kIF_MOD,      'unless'   => :kUNLESS_MOD,
  'while'   => :kWHILE_MOD,   'until'    => :kUNTIL_MOD,
  'rescue'  => :kRESCUE_MOD,  'defined?' => :kDEFINED,
  'BEGIN'   => :klBEGIN,      'END'      => :klEND,
}

KEYWORDS_BEGIN = {
  'if'      => :kIF,          'unless'   => :kUNLESS,
  'while'   => :kWHILE,       'until'    => :kUNTIL,
  'rescue'  => :kRESCUE,      'defined?' => :kDEFINED,
  'BEGIN'   => :klBEGIN,      'END'      => :klEND,
}

%w(class module def undef begin end then elsif else ensure case when
   for break next redo retry in do return yield super self nil true
   false and or not alias __FILE__ __LINE__ __ENCODING__).each do
|keyword|
  KEYWORDS_BEGIN[keyword] = KEYWORDS[keyword] = :"k#{keyword.upcase}"
end
```

```
# === CHARACTER CLASSES ===
#
# Pay close attention to the differences between c_any and any.
# c_any does not include EOF and so will cause incorrect behavior
# for machine subtraction (any-except rules) and default transitions
# for scanners.

action do_nl {
  # Record position of a newline for precise location reporting on tNL
  # tokens.
  #
  # This action is embedded directly into c_nl, as it is idempotent and
  # there are no cases when we need to skip it.
  @newline_s = p
}

c_nl      = '\n' $ do_nl;
c_space   = [ \t\r\f\v];
c_space_nl = c_space | c_nl;

c_eof     = 0x04 | 0x1a | 0 | zlen; # ^D, ^Z, \0, EOF
c_eol     = c_nl | c_eof;
c_any     = any - c_eof;

c_nl_zlen  = c_nl | zlen;
c_line     = any - c_nl_zlen;

c_unicode  = c_any - 0x00..0x7f;
c_upper    = [A-Z];
c_lower    = [a-z_]  | c_unicode;
c_alpha    = c_lower | c_upper;
c_alnum    = c_alpha | [0-9];

action do_eof {
  # Sit at EOF indefinitely. #advance would return $eof each time.
  # This allows to feed the lexer more data if needed; this is only used
  # in tests.
  #
  # Note that this action is not embedded into e_eof like e_heredoc_nl and
e_bs
  # below. This is due to the fact that scanner state at EOF is observed
  # by tests, and encapsulating it in a rule would break the
introspection.
  fhold; fbreak;
}

#
# === TOKEN DEFINITIONS ===
#

# All operators are punctuation. There is more to punctuation
# than just operators. Operators can be overridden by user;
# punctuation can not.

# A list of operators which are valid in the function name context, but
# have different semantics in others.
operator_fname      = '[]' | '[]=' | '`'  | '-@' | '+@' | '~@'  | '!@' ;

# A list of operators which can occur within an assignment shortcut (+ →
+=).
operator_arithmetic = '&'  | '|'  | '&&' | '||' | '^'  | '+'   | '-'  |
                      '*'  | '/'  | '%'  | '**' | '<<' | '>>'  | '&.' ;

# A list of all user-definable operators not covered by groups above.
operator_rest       = '=~' | '!~' | '~'  | '!'  | '<=>' |
                      '<'  | '<=' | '>'  | '>=' | '===' | '=>' ;

# Note that `{` and `}` need to be referred to as e_lbrace and e_rbrace,
# as they are ambiguous with interpolation `#{}` and should be counted.
# These braces are not present in punctuation lists.

# A list of punctuation which has different meaning when used at the
# beginning of expression.
punctuation_begin   = '-'  | '+'  | '::' | '('  | '[' |
                      '*'  | '**' | '&' ;

# A list of all punctuation except punctuation_begin.
punctuation_end     = ','  | '='  | '->' | '{'  | ']' | '}' |
                      '::' | '?'  | ':'  | '.'  | '..' | '...' ;

# A list of keywords which have different meaning at the beginning of
expression.
keyword_modifier    = 'if'     | 'unless' | 'while'  | 'until' | 'rescue'
;

# A list of keywords which accept an argument-like expression, i.e. have
the
# same post-processing as method calls or commands. Example: `yield 1`,
# `yield(1)`, `yield[1]`, are interpreted as if `yield` was a function.
keyword_with_arg    = 'yield'  | 'super'  | 'not'    | 'defined?' ;

# A list of keywords which accept a literal function name as an argument.
keyword_with_fname  = 'def'    | 'undef'  | 'alias' ;

# A list of keywords which accept an expression after them.
keyword_with_value  = 'else'   | 'case'   | 'ensure' | 'module' | 'elsif'
| 'then'   |
                      'for'    | 'in'     | 'do'     | 'when'   | 'begin'
| 'class' |
                      'and'    | 'or'     ;

# A list of keywords which accept a value, and treat the keywords from
# `keyword_modifier` list as modifiers.
keyword_with_mid    = 'rescue' | 'return' | 'break'  | 'next'   ;

# A list of keywords which do not accept an expression after them.
keyword_with_end    = 'end'    | 'self'   | 'true'   | 'false'  | 'retry'
  |
                      'redo'   | 'nil'    | 'BEGIN'  | 'END'    |
'__FILE__' |
                      '__LINE__' | '__ENCODING__';

# All keywords.
keyword             = keyword_with_value | keyword_with_mid |
                      keyword_with_end   | keyword_with_arg |
                      keyword_with_fname | keyword_modifier ;


constant        = c_upper c_alnum*;
bareword        = c_alpha c_alnum*;

call_or_var     = c_lower c_alnum*;
class_var       = '@@' bareword;
instance_var    = '@' bareword;
global_var      = '$'
    ( bareword | digit+
    | [~*$?!@/\\;,.=:<>"] #
    | '-' c_alnum
    )
;

# Ruby accepts (and fails on) variables with leading digit
# in literal context, but not in unquoted symbol body.
class_var_v     = '@@' c_alnum+;
instance_var_v  = '@' c_alnum+;

label           = bareword [?!]? ':';
```


2 + 2
Lexer
[tNUMBER, 2],
[tPLUS, "+"],
[tNUMBER, 2]

```ruby
class Lexer
  def initialize
    @data = nil # array of input symbols
    @ts = nil # token start index
    @te = nil # token end index
    @eof = nil # EOF index
    @tokens = [] # resulting array of tokens
  end
end
```

```
class Lexer
  %%{ #%
    machine ultimate_math_machine;

    access @;
    variable eof @eof;

    number   = [0-9]+;
    operator = "+" | "-" | "/" | "*";
    paren    = "(" | ")";

    main := |*
      number           => { puts "number [#{@ts},#{@te}]" };
      operator | paren => { puts "operator [#{@ts},#{@te}]" };
      space;
    *|;
  }%% #%
end
```

```
class Lexer
  def run(input)
    @data = input.unpack("c*")
    @eof = input.length

    %% write data; #%# Ragel static data init
    %% write init; #%# State machine init
    %% write exec; #%# State machine execution
  end
end
```

```ruby
class Lexer
  def run(input)
    @data = input.unpack("c*")
    @eof = input.length

    %% write data; #%# Ragel static data init
    %% write init; #%# State machine init
    %% write exec; #%# State machine execution
  end
end
```

```
$ ragel -R lexer.rl -o lexer.rb

$ ruby -r './lexer.rb' -e 'Lexer.new.run("40 + 2")'
number [0,2]
operator [3,4]
number [5,6]
```

```
class Lexer
  def current_token
    @data[@ts...@te].pack("c*")
  end

  def emit(type, tok = current_token)
    @tokens.push([type, tok])
  end
end
```

```ruby
class Lexer
  def current_token
    @data[@ts...@te].pack("c*")
  end

  def emit(type, tok = current_token)
    @tokens.push([type, tok])
  end
end
```

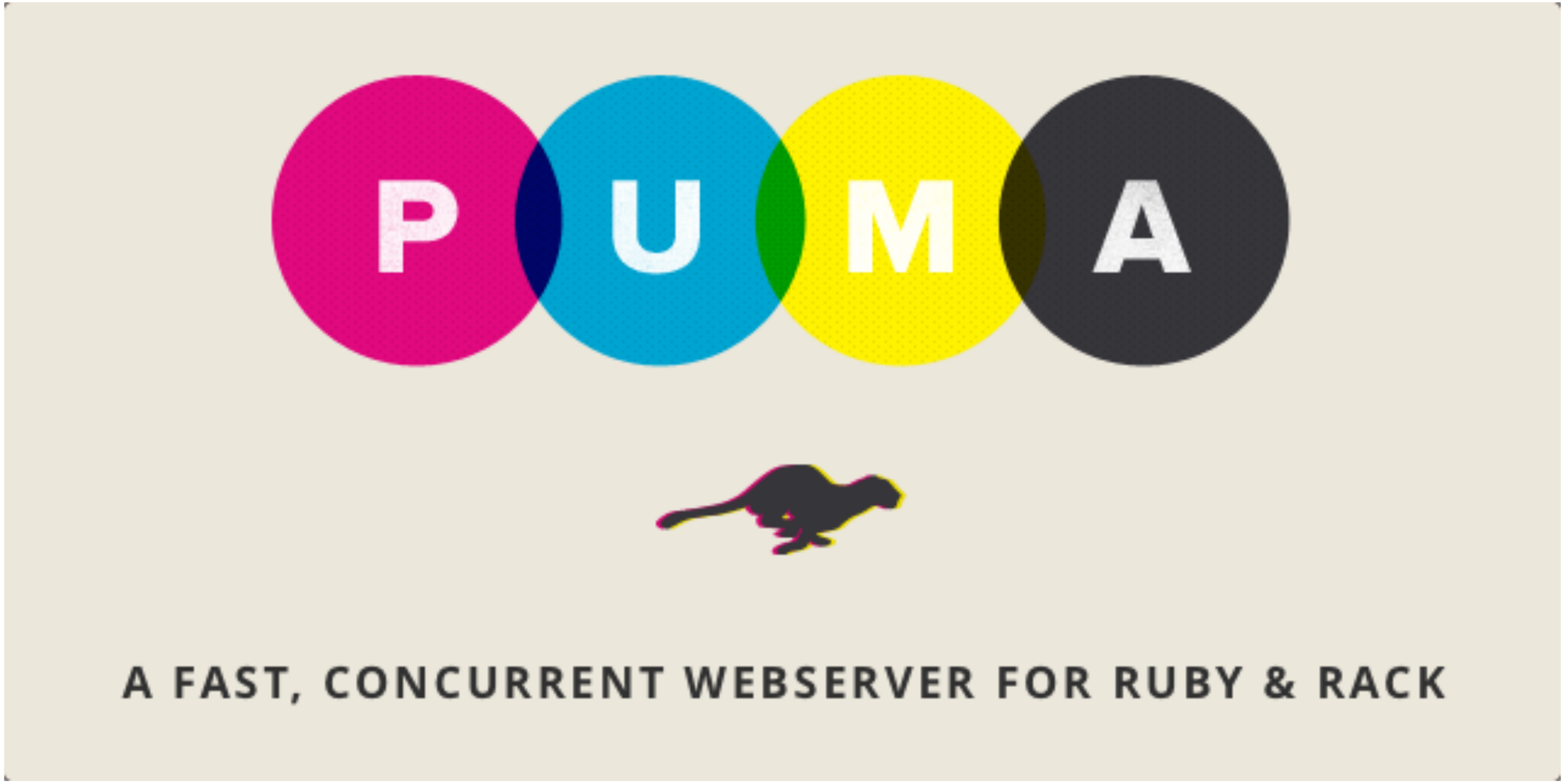
```ruby
class Lexer
  PUNCTUATION = {
    '+' ⇒ :tPLUS,   '-' ⇒ :tMINUS,
    '*' ⇒ :tSTAR,   '/' ⇒ :tDIVIDE,
    '(' ⇒ :tLPAREN, ')' ⇒ :tRPAREN
  }

  def emit_table(table)
    token = current_token
    emit(table[token], token)
  end
end
```

```
1 class Lexer
2   %%{ #%
3     main := |*
4       number             ⇒ { emit(:tNUMBER) };
5       operator | paren ⇒ { emit_table(PUNCTUATION) };
6       space;
7     *|;
8   }%% #%
9 end
```

```
class Lexer
  %%{ #%
    main := |*
      number            ⇒ { emit(:tNUMBER) };
      operator | paren ⇒ { emit_table(PUNCTUATION) };
      space;
    *|;
  }%% #%
end
```

```
class Lexer
  def run(input)
    @data = input.unpack("c*")
    @eof = input.length

    %% write data; #%
    %% write init; #%
    %% write exec; #%

    @tokens
  end
end
```

```
$ ragel -R lexer.rl -o lexer.rb

$ ruby -r './lexer.rb' -e 'p Lexer.new.run("2 + (8 * 5)")'
[[:tNUMBER, "2"], [:tPLUS, "+"], [:tLPAREN, "("], [:tNUMBER, "8"],
[:tSTAR, "*"], [:tNUMBER, "5"], [:tRPAREN, ")"]]
```


PUMA
A FAST, CONCURRENT WEBSERVER FOR RUBY & RACK


$ wc -l lib/parser/lexer.rl
2556 lib/parser/lexer.rl
$ ragel -F1 -R lib/parser/lexer.rl -o lib/parser/lexer.rb
$ wc -l lib/parser/lexer.rb
23779 lib/parser/lexer.rb

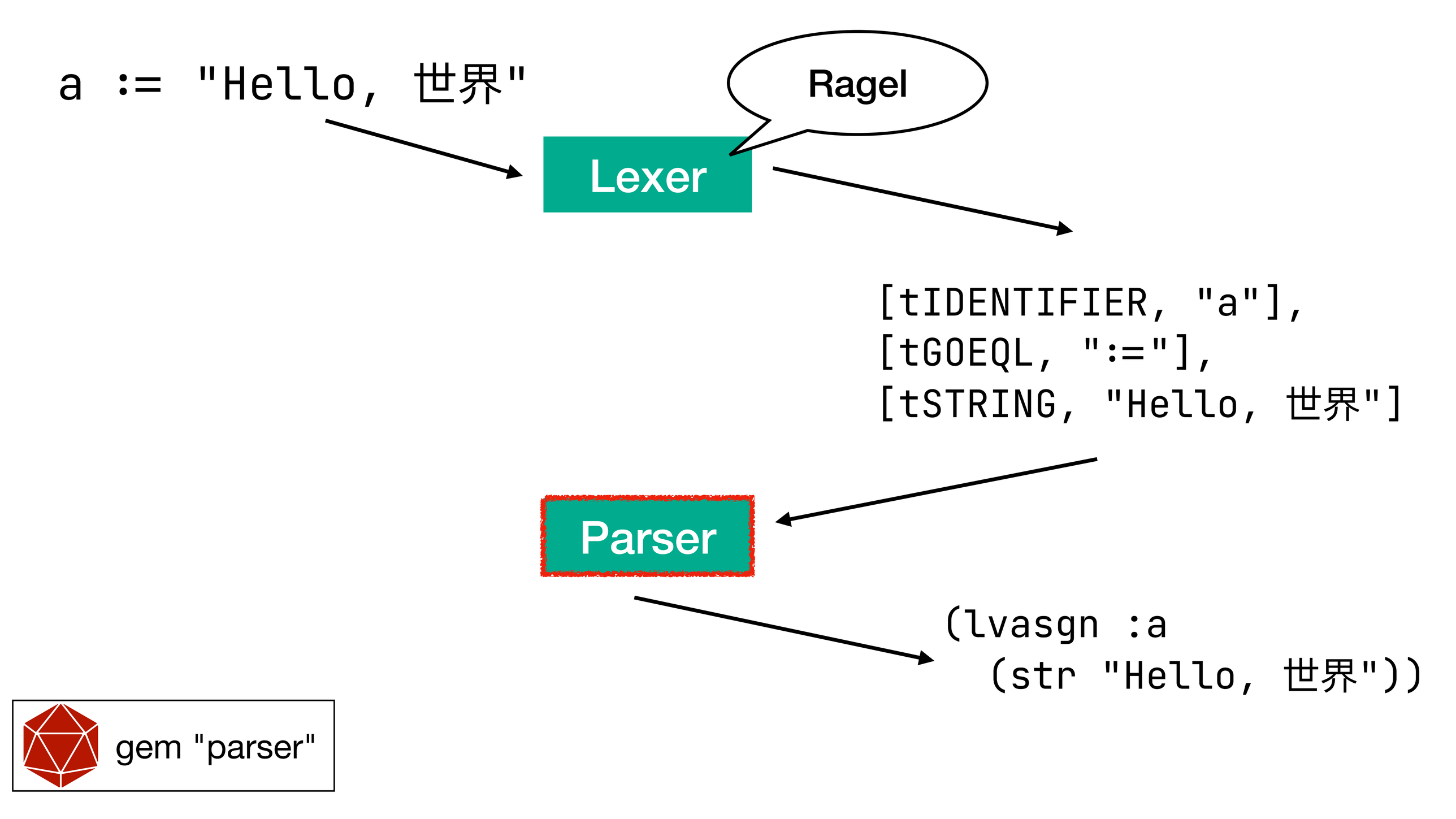
```
443   PUNCTUATION = {
444     '='   => :tEQL,      '&'   => :tAMPER2,  '|'   => :tPIPE,
445     '!'   => :tBANG,     '^'   => :tCARET,   '+'   => :tPLUS,
446     '-'   => :tMINUS,    '*'   => :tSTAR2,   '/'   => :tDIVIDE,
447     '%'   => :tPERCENT,  '~'   => :tTILDE,   ','   => :tCOMMA,
448     ';'   => :tSEMI,     '.'   => :tDOT,     '..'  => :tDOT2,
449     '...' => :tDOT3,     '['   => :tLBRACK2, ']'   => :tRBRACK,
450     '('   => :tLPAREN2,  ')'   => :tRPAREN,  '?'   => :tEH,
451     ':'   => :tCOLON,    '&&'  => :tANDOP,   '||'  => :tOROP,
452     '-@'  => :tUMINUS,   '+@'  => :tUPLUS,   '~@'  => :tTILDE,
453     '**'  => :tPOW,      '->'  => :tLAMBDA,  '=~'  => :tMATCH,
454     '!~'  => :tNMATCH,   '=='  => :tEQ,      '!='  => :tNEQ,
455     '>'   => :tGT,       '>>'  => :tRSHFT,   '>='  => :tGEQ,
456     '<'   => :tLT,       '<<'  => :tLSHFT,   '<='  => :tLEQ,
457     '=>'  => :tASSOC,    '::'  => :tCOLON2,  '===' => :tEQQ,
458     '<=>' => :tCMP,      '[]'  => :tAREF,    '[]=' => :tASET,
459     '{'   => :tLCURLY,   '}'   => :tRCURLY,  '`'   => :tBACK_REF2,
460     '!@'  => :tBANG,     '&.'  => :tANDDOT,  '.:'  => :tMETHREF,
461     ':='  => :tGOEQL
462   }
```

```
569 # A list of all punctuation except punctuation_begin.
570 punctuation_end     = ','   | '='   | ':='   | '→' | '('   | '['   |
571                       ']'   | '::'  | '?'    | ':'  | '.'   | '..'  | '...' ;
```

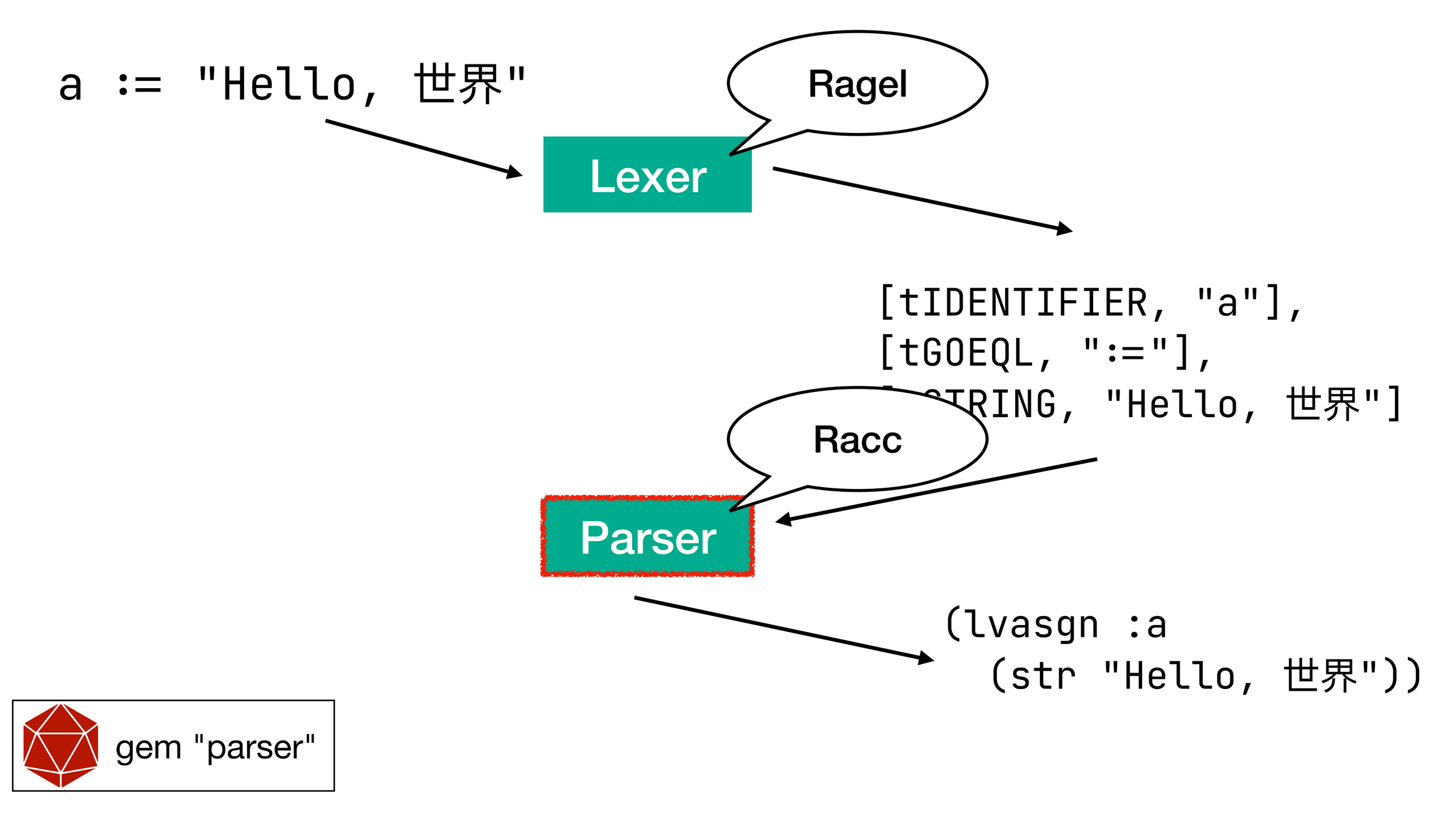
```
1399 # expr_fname := |*
1400 # ...
1401         '::'
1402             => { fhold; fhold; fgoto expr_end; };
1403
1404         ':='
1405             => { fhold; fhold; fgoto expr_end; };
1406
1407         ':'
1408             => { fhold; fgoto expr_beg; };
```

```ruby
# test/ruby-next/test_lexer.rb

  def test_go_eql
    setup_lexer "next"

    assert_scanned('foo := 42',
      :tIDENTIFIER, 'foo', [0, 3],
      :tGOEQL,       ':=',  [4, 6],
      :tINTEGER,      42, [7, 9])
  end
```

```
$ bundle exec rake test
Run options: --seed 35817

# Running:

....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
.....................................................................................................................

Finished in 7.056126s, 152.0664 runs/s, 48174.1681 assertions/s.

1073 runs, 339923 assertions, 0 failures, 0 errors, 0 skips
3 additional tests on 221724 nodes ran successfully
```

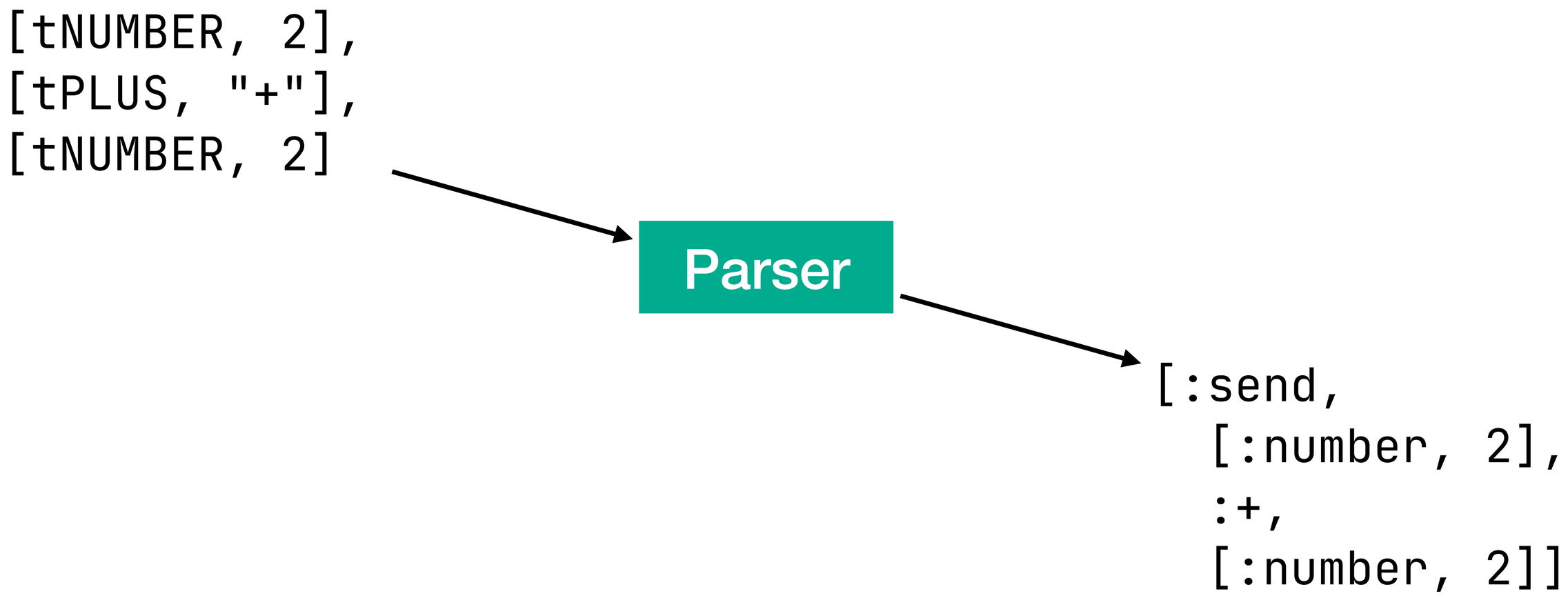

a := "Hello, 世界"
Ragel
Lexer
[tIDENTIFIER, "a"],
[tGOEQL, ":="],
[tSTRING, "Hello, 世界"]
Parser
(lvasgn :a
(str "Hello, 世界"))
gem "parser"


I VOTED
Maricopa County Recorder & Elections Department

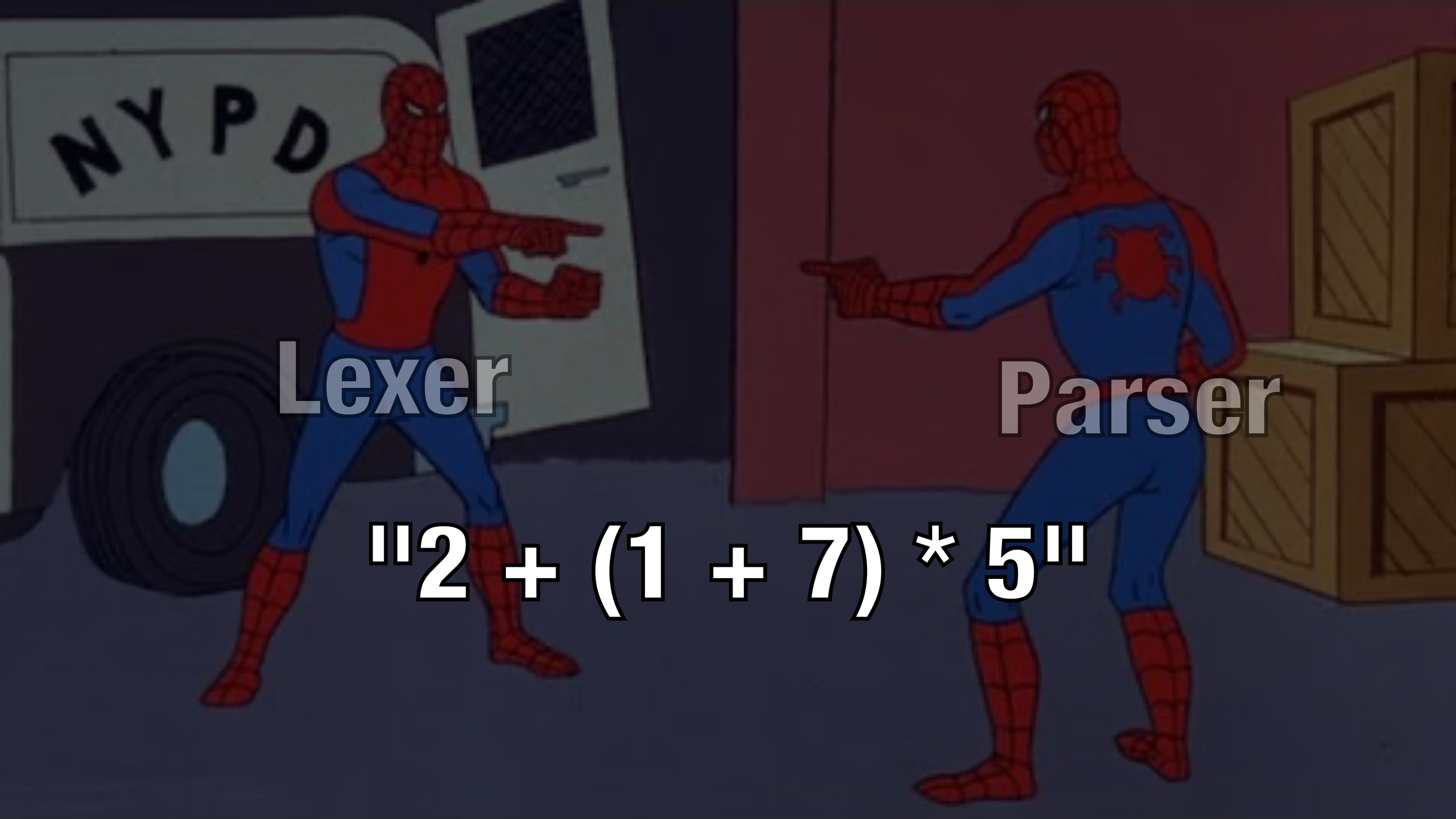

a := "Hello, 世界"
Ragel
Lexer
[tIDENTIFIER, "a"],
[tGOEQL, ":="],
Racc
Parser
(lvasgn :a
(str "Hello, 世界"))
gem "parser"

~~Rake~~ ~~Rack~~ Racc

- Yacc/Bison like parser generator
- Grammar file should contain:
  - parser class with defined #next_token
  - rules block


[tNUMBER, 2],
[tPLUS, "+"],
[tNUMBER, 2]
Parser
[:send,
[:number, 2],
:+,
[:number, 2]]


NYPD
Lexer
Parser


NYPD
Lexer
Parser
"2 + (1 + 7) * 5"

```
class MathParser
  token tNUMBER tPLUS   tMINUS  tSTAR
        tDIVIDE tLPAREN tRPAREN

  prechigh
    left tSTAR tDIVIDE
    left tPLUS tMINUS
  preclow
end
```

```
class MathParser
  # ...
  rule
    exp: operation
       | paren
       | number

    number: tNUMBER { result = [:number, val[0]] }

    paren: tLPAREN exp tRPAREN { result = val[1] }

    operation: exp tPLUS exp   { result = [:send, val[0], val[1], val[2]] }
             | exp tMINUS exp  { result = [:send, val[0], val[1], val[2]] }
             | exp tSTAR exp   { result = [:send, val[0], val[1], val[2]] }
             | exp tDIVIDE exp { result = [:send, val[0], val[1], val[2]] }
end
```

```ruby
# class MathParser ... end

---- header
require_relative "./lexer.rb"

---- inner

def parse(arg)
  @tokens = Lexer.new.run(arg)
  do_parse
end

def next_token
  @tokens.shift
end
```

```
$ racc parser.y -o parser.rb

$ ruby -r './parser.rb' -e 'pp MathParser.new.parse("5 * (4 + 3) + 2");'
[:send,
  [:send,
    [:number, "5"],
    "*",
    [:send,
      [:number, "4"],
      "+",
      [:number, "3"]]],
  "+",
  [:number, "2"]]
```

Word of the Day : August 26, 2021

parser

noun | pars·er

```
764 #...
765       arg: lhs tEQL arg_rhs
766               {
767                 result = @builder.assign(val[0], val[1], val[2])
768               }
769             #...
```

```
764 #...
765       arg: lhs tEQL arg_rhs
766               {
767                 result = @builder.assign(val[0], val[1], val[2])
768               }
769             #...
```

```
697 # lib/parser/builders/default.rb
698
699 def assign(lhs, eql_t, rhs)
700   (lhs << rhs).updated(nil, nil,
701     :location ⇒ lhs.loc.
702       with_operator(loc(eql_t)).
703       with_expression(join_exprs(lhs, rhs)))
704 end
```

```
class Parser::RubyNext

token kCLASS kMODULE kDEF kUNDEF kBEGIN kRESCUE kENSURE kEND kIF kUNLESS
      kTHEN kELSIF kELSE kCASE kWHEN kWHILE kUNTIL kFOR kBREAK kNEXT
      kREDO kRETRY kIN kDO kDO_COND kDO_BLOCK kDO_LAMBDA kRETURN kYIELD kSUPER
      kSELF kNIL kTRUE kFALSE kAND kOR kNOT kIF_MOD kUNLESS_MOD kWHILE_MOD
      kUNTIL_MOD kRESCUE_MOD kALIAS kDEFINED klBEGIN klEND k__LINE__
      k__FILE__ k__ENCODING__ tIDENTIFIER tFID tGVAR tIVAR tCONSTANT
      tLABEL tCVAR tNTH_REF tBACK_REF tSTRING_CONTENT tINTEGER tFLOAT
      tUPLUS tUMINUS tUNARY_NUM tPOW tCMP tEQ tEQQ tNEQ
      tGEQ tLEQ tANDOP tOROP tMATCH tNMATCH tDOT tDOT2 tDOT3 tAREF
      tASET tLSHFT tRSHFT tCOLON2 tCOLON3 tOP_ASGN tASSOC tLPAREN
      tLPAREN2 tRPAREN tLPAREN_ARG tLBRACK tLBRACK2 tRBRACK tLBRACE
      tLBRACE_ARG tSTAR tSTAR2 tAMPER tAMPER2 tTILDE tPERCENT tDIVIDE
      tDSTAR tPLUS tMINUS tLT tGT tPIPE tBANG tCARET tLCURLY tRCURLY
      tBACK_REF2 tSYMBEG tSTRING_BEG tXSTRING_BEG tREGEXP_BEG tREGEXP_OPT
      tWORDS_BEG tQWORDS_BEG tSYMBOLS_BEG tQSYMBOLS_BEG tSTRING_DBEG
      tSTRING_DVAR tSTRING_END tSTRING_DEND tSTRING tSYMBOL
      tNL tEH tCOLON tCOMMA tSPACE tSEMI tLAMBDA tLAMBEG tCHARACTER
      tRATIONAL tIMAGINARY tLABEL_END tANDDOT tMETHREF tBDOT2 tBDOT3
      tGOEQL
```

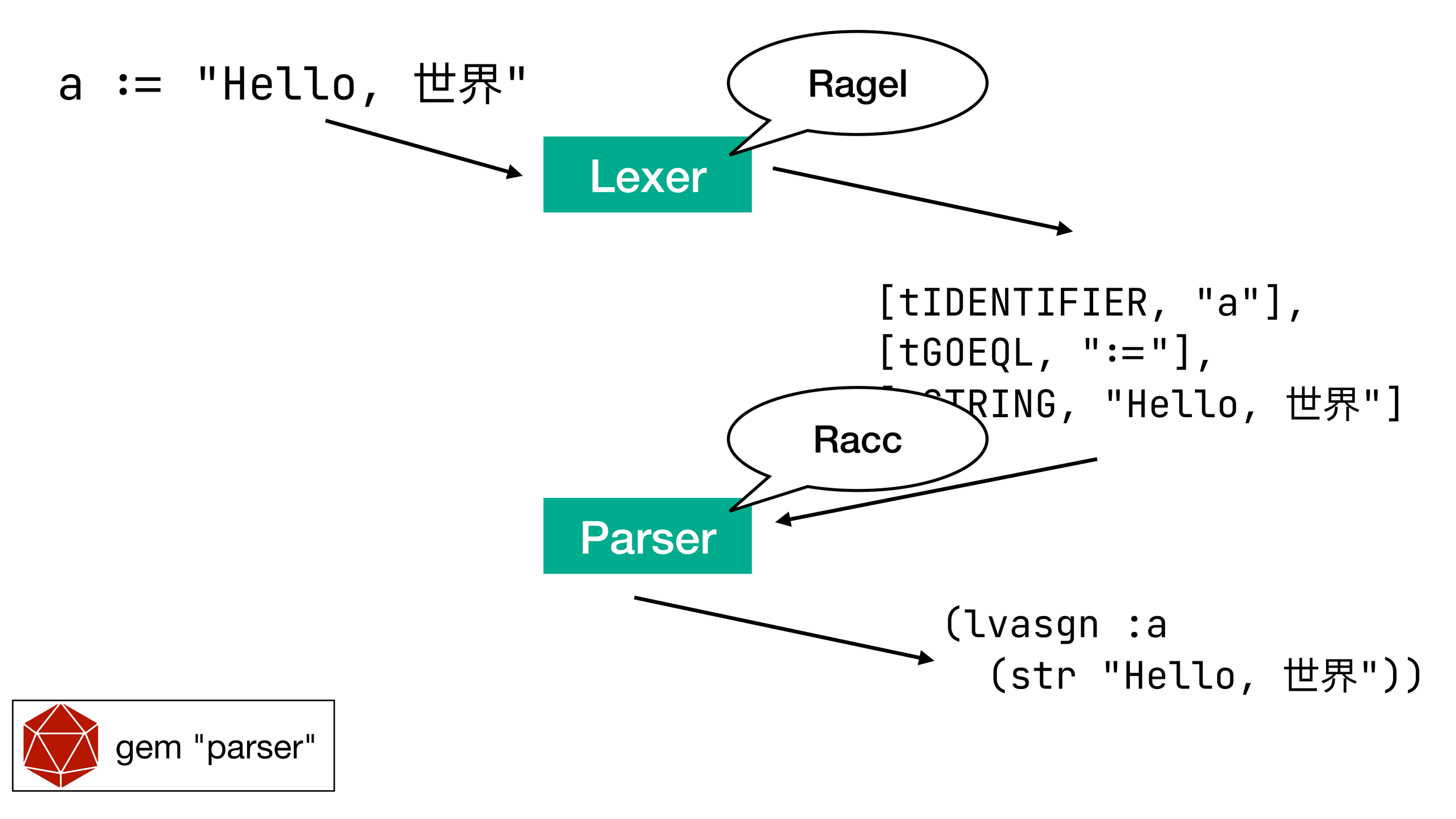
```
223     command_asgn: lhs tEQL command_rhs
224                     {
225                       result = @builder.assign(val[0], val[1], val[2])
226                     }
227                 | lhs tGOEQL command_rhs
228                     {
229                       result = @builder.assign(val[0], val[1], val[2])
230                     }
231
```

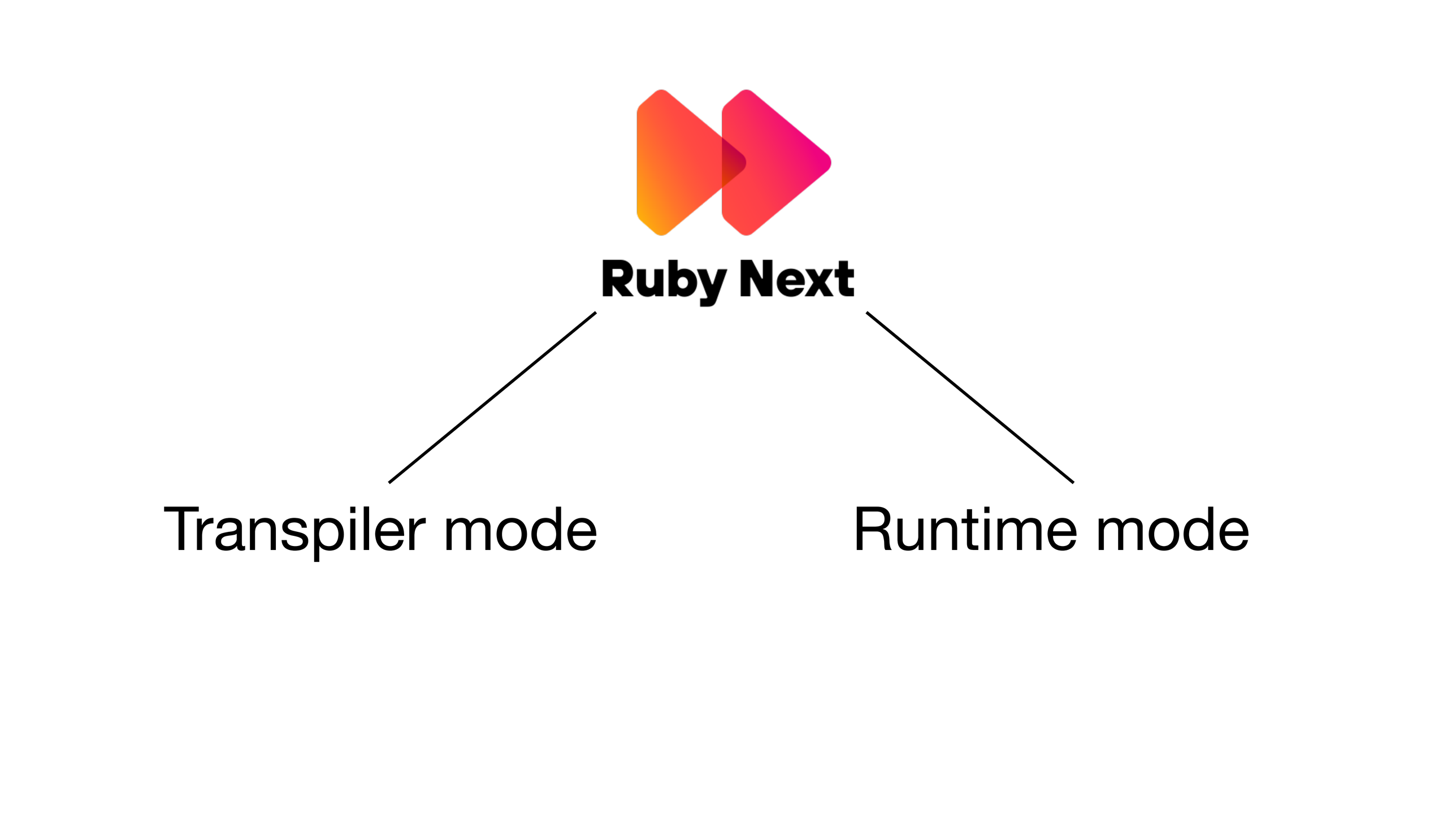
```
765       arg: lhs tEQL arg_rhs
766               {
767                 result = @builder.assign(val[0], val[1], val[2])
768               }
769           | lhs tGOEQL arg_rhs
770               {
771                 result = @builder.assign(val[0], val[1], val[2])
772               }
773           # ...
```

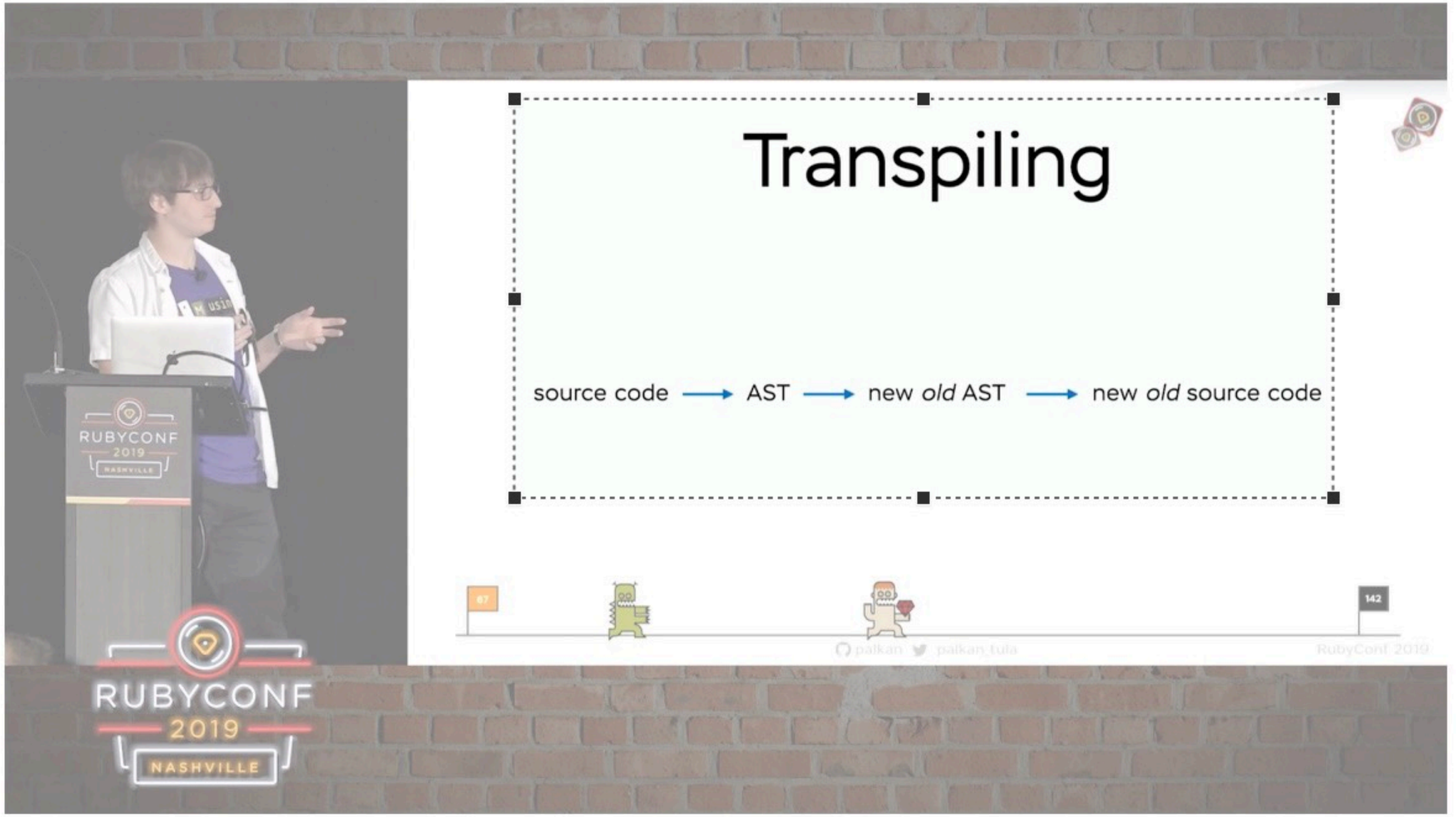
```ruby
# test/ruby-next/test_parser.rb

  def test_go_eql
    assert_parses(
      s(:lvasgn, :foo, s(:int, 42)),
      %q{foo := 42},
      %q{    ^^ operator
        |~~~~~~~~~ expression},
      SINCE_NEXT)
  end
```

```
$ bundle exec rake test
Run options: --seed 35817

# Running:

....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
.....................................................................................................................

Finished in 7.056126s, 152.0664 runs/s, 48174.1681 assertions/s.

1073 runs, 339923 assertions, 0 failures, 0 errors, 0 skips
3 additional tests on 221724 nodes ran successfully
```

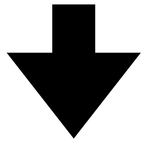

Vin again


a := "Hello, 世界"
Ragel
Lexer
[tIDENTIFIER, "a"],
[tGOEQL, ":="],
Racc
Parser
(lvasgn :a
(str "Hello, 世界"))
gem "parser"


Ruby Next
Transpiler mode
Runtime mode

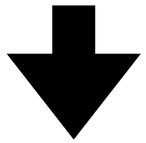

RUBYCONF
2019
NASHVILLE
Transpiling
source code → AST → new *old* AST → new *old* source code
67
142
palkan
palkan_tula
RubyConf 2019
RUBYCONF
2019
NASHVILLE
Done

```
a := "Hello, 世界"
```



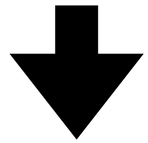
```
(lvasgn :a
  (str "Hello, 世界"))
```

```
a := "Hello, 世界"
```



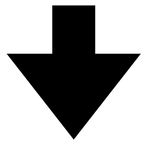
```
(lvasgn :a
  (str "Hello, 世界"))
```

```
a = "Hello, 世界"
```



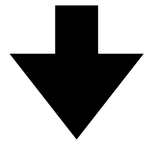
```
(lvasgn :a
  (str "Hello, 世界"))
```

```
a := "Hello, 世界"
  ^^ tGOEQL ":="
```



```
(lvasgn :a
  (str "Hello, 世界"))
```

```
a = "Hello, 世界"
  ^ tEQL "="
```



```
(lvasgn :a
  (str "Hello, 世界"))
```

```ruby
# frozen_string_literal: true

module RubyNext
  module Language
    module Rewriters
      class GoAssign < Base
        NAME = "go-assign"
        SYNTAX_PROBE = "foo := 42"
        MIN_SUPPORTED_VERSION = Gem::Version.new(RubyNext::NEXT_VERSION)

        def on_lvasgn(node)
          return if node.loc.operator == "="

          context.track! self

          replace(node.loc.operator, "=")
        end
      end
    end
  end
end
```


$ ruby -r './lib/go_ruby.rb' ./main.go.rb
Hello, 世界


If you give a man a fish...

skryukov / ruby-go Private

Unwatch 1 Star 0 Fork 0

Code Issues Pull requests Actions Projects Security Insights Settings

main 1 branch 0 tags Go to file Add file Code

skryukov Initial commit 15823f2 now 1 commit

README.md Initial commit now

README.md

ruby-go

Stupidest thing I've never done

About

Stupidest thing I've never done

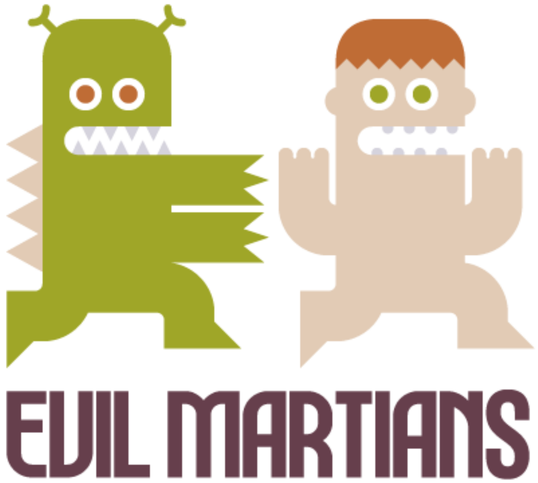
Readme

Releases

No releases published
Create a new release

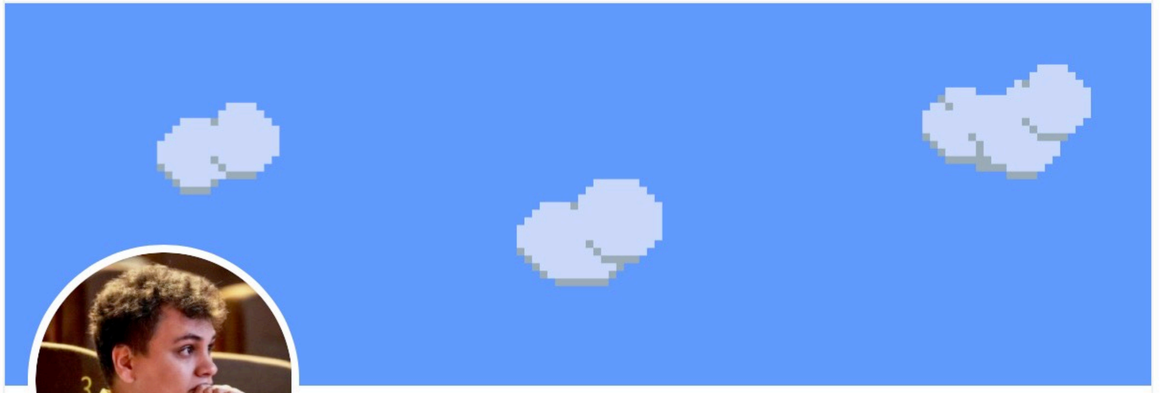
Packages

No packages published
Publish your first package



https://evl.ms/obey

Svyatoslav Kryukov
1 Tweet



Edit profile

Svyatoslav Kryukov
@kryukov_s

Backend at @evilmartians

Moscow, Russia github.com/skryukov Joined April 2017

54 Following **23** Followers

Tweets | Tweets & replies | Media | Likes

Svyatoslav Kryukov @kryukov_s · Mar 4

Verifying myself: I am skryukov on Keybase.io.
wZNHF5ZAQR1j50rXxU0w9h-xexpLw7D2V7ue / keybase.io/skryukov/sigs/...


Questions?