PEGs, Treetop, and Converting Regular Expressions to NFAs

Jason Dew and Gary Fredericks

Parsing Expression Grammars and Treetop

Jason Dew

Outline

1. Introduction to PEGs
2. Introduction to Treetop
3. References and Questions

PEGs

- PEG := parsing expression grammar
- A generalization of regular expressions
- Similar to context-free grammars
- Unlike BNF, parse trees are unambiguous

Formal Definition

N: a finite set of non-terminal symbols

∑: a finite set of terminal symbols

P: a finite set of parsing rules

$\mathbf{e_s}$: the starting expression

Formal Definition

Parsing rules take the form: A := e

non-terminal (A)

parsing expression (or some combination) (e)

Parsing Expressions

Several ways to combine expressions:

- sequence: "foo" "bar”
- ordered choice: "foo" / "bar”
- zero or more: "foo"*
- one or more: "foo"+
- optional: "foo"?

Parsing Expressions

Lookahead assertions (these do *not* consume any input):

- positive lookahead: "foo" &"bar”
- negative lookahead: "foo" !"baz"

Implementations

Java: parboiled, rats!
C: peg, leg, pegc
C++: boost
Python: ppeg, pypeg, pijnu
Javascript: kouprey
Perl 6: (part of the language)
Erlang: neotoma
Clojure: clj-peg
F#: fparsec

and finally... Ruby has Treetop

Treetop

A DSL (domain-specific language)
written in Ruby
for implementing PEGs

Syntax

Two main keywords in the DSL: grammar and rule

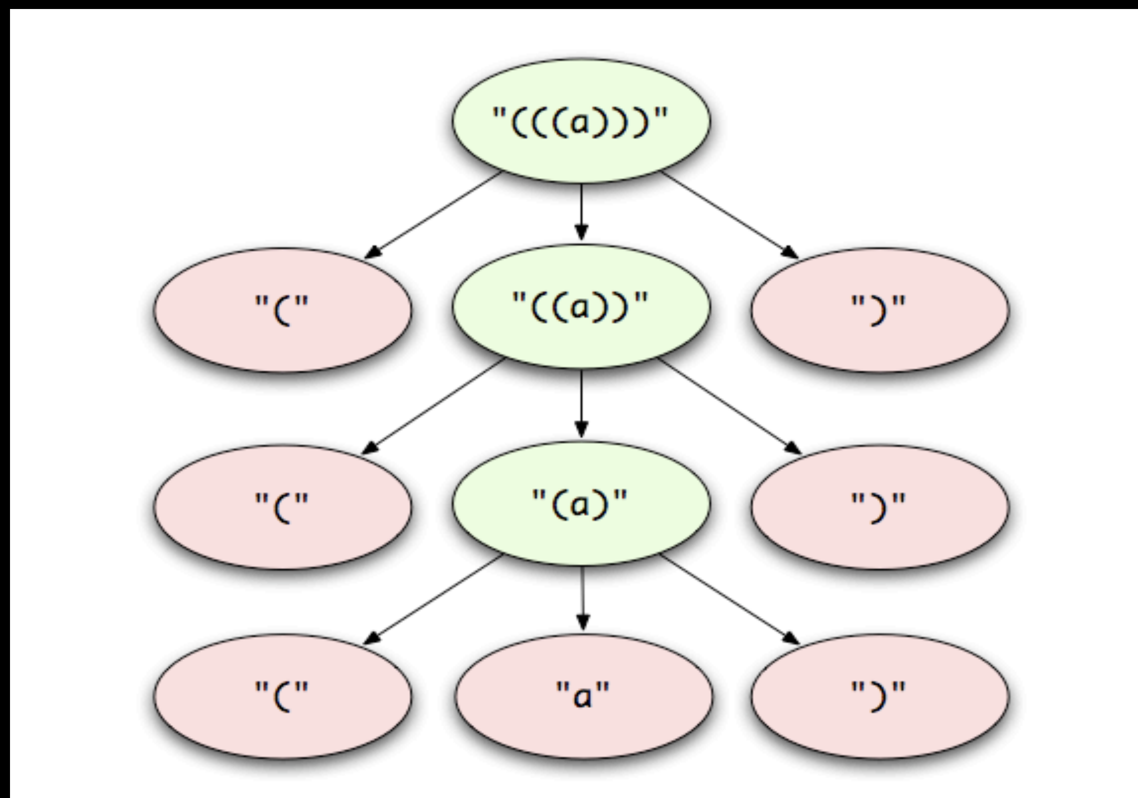
```
grammar Arithmetic
  rule additive
    multitive '+' additive / multitive
  end

  rule multitive
    primary '*' multitive / primary
  end

  rule primary
    '(' additive ')' / number
  end

  rule number
    [1-9] [0-9]*
  end
end
```

Semantics

Consider the following PEG and the input string `(((a)))`:

```
grammar ParenthesizedLanguage
  rule parenthesized_letter
    '(' parenthesized_letter ')'
    /
    [a-z]
  end
end
```

the resulting parse tree:



And now for the cool part

- each of the nodes are instances of `Treetop::Runtime::SyntaxNode`
- semantics get defined here
- all of Ruby is available to you

Example

```
grammar ParenthesizedLanguage
  rule parenthesized_letter
    '(' parenthesized_letter ')' {
      def depth
        parenthesized_letter.depth + 1
      end
    }
    /
    [a-z] {
      def depth
        0
      end
    }
  end
end
```

Example (sans code duplication)

```
# in .treetop file
grammar ParenthesizedLanguage
  rule parenthesized_letter
    '(' parenthesized_letter ')' <ParenthesizedNode>
    /
    [a-z] <ParenthesizedNode>
  end
end
```

```
# in separate .rb file
class ParenthesizedNode < Treetop::Runtime::SyntaxNode
  def depth
    if nonterminal?
      parenthesized_letter.depth + 1
    else
      0
    end
  end
end
```

Treetop::Runtime::SyntaxNode

Methods available:

- `#terminal?` : true if this node corresponds to a terminal symbol, false otherwise
- `#non_terminal?` : true if this node corresponds to a non-terminal symbol, false otherwise
- `#text_value` : returns the matched text
- `#elements` : returns the child nodes (only for non-terminal nodes)

References and Questions

http://en.wikipedia.org/wiki/Parsing_expression_grammar
http://treetop.rubyforge.org/

RE → εNFA

Gary Fredericks

Plan

1. Demonstrate Application
2. Show Treetop Parse Tree
3. Class NFA
    1. Simple one-character NFA
    2. Combined NFAs
        1. Question Mark
        2. Kleene Star
        3. Concatenation
        4. Or
4. Optimizations

Application

- http://gfredericks.com/main/sandbox/regex

Temporary shortcut:

- gfredericks.com/531

Treetop Grammar

```
grammar Regex
  rule reg
    expression
  end

  rule expression
    term ("|" term)* <RegexNFA::Expression>
  end

  rule term
    modified_factor+ <RegexNFA::Term>
  end
```

Treetop Grammar (cont)

```
rule modified_factor
  factor modifier? <RegexNFA::ModifiedFactor>
end

rule factor
  "(" expression ")" <RegexNFA::Factor> / literal / characterClass
end

rule modifier
  optional / one_or_more / zero_or_more / specified_number
end
```

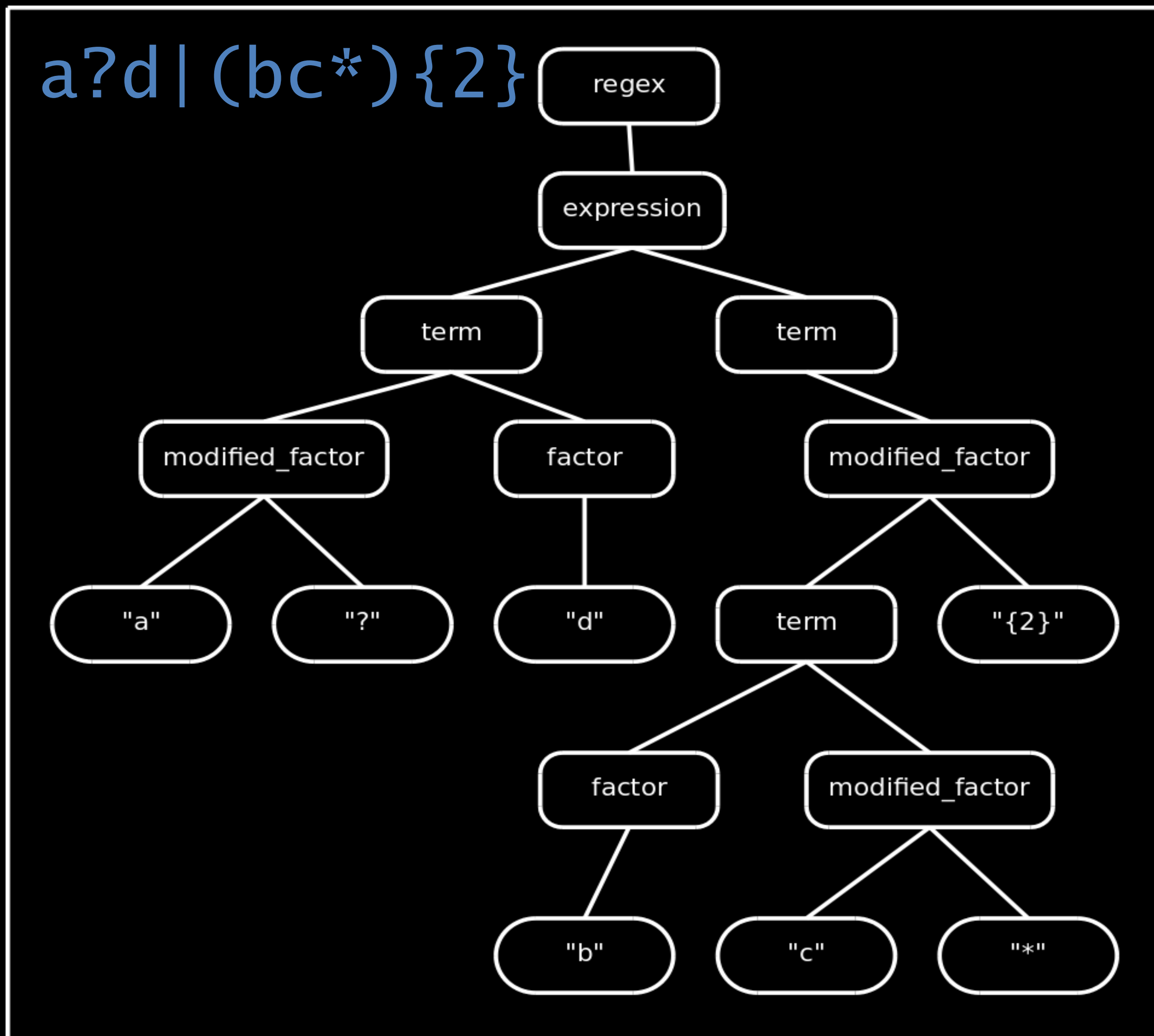
Treetop Grammar (cont)

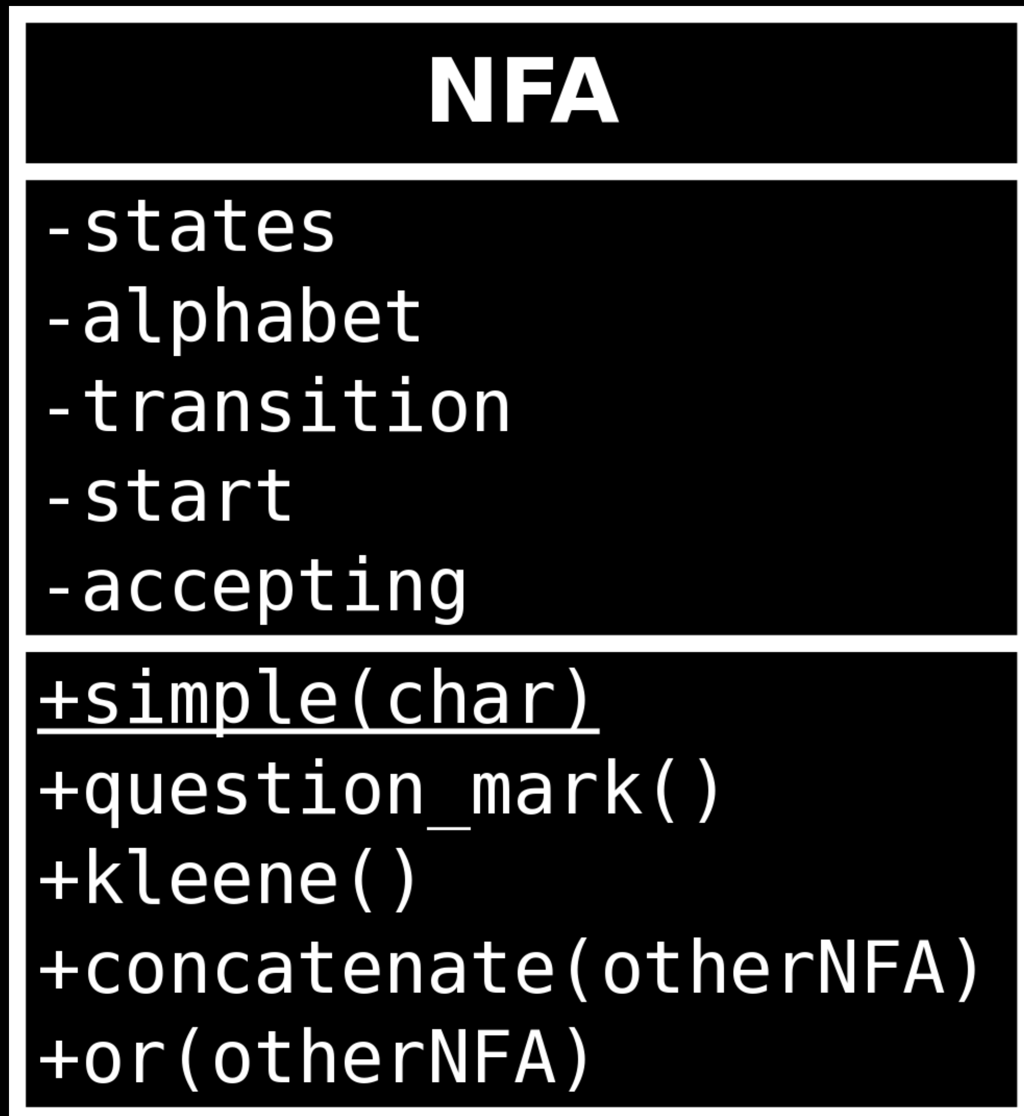
```
rule optional
  "?" <RegexNFA::Optional>
end

rule one_or_more
  "+" <RegexNFA::OneOrMore>
end

rule zero_or_more
  "*" <RegexNFA::ZeroOrMore>
end

rule specified_number
  "{" [0-9]+ ("," [0-9]* )? "}" <RegexNFA::SpecifiedNumber>
end
…
```

Treetop Example

- a?d|(bc*){2} matches
  - ad
  - d
  - bb
  - bcb
  - bbccccccc
  - bcccccccbccccccc

Treetop Syntax Tree




NFA
-states
-alphabet
-transition
-start
-accepting
+simple(char)
+question_mark()
+kleene()
+concatenate(otherNFA)
+or(otherNFA)

Simplifying Assumptions

Every NFA has a start state with <u>only</u> outgoing transitions

Every NFA has a single accepting state with <u>only</u> incoming transitions

(This means no self-transitions in either case)

NFA.simple(char)

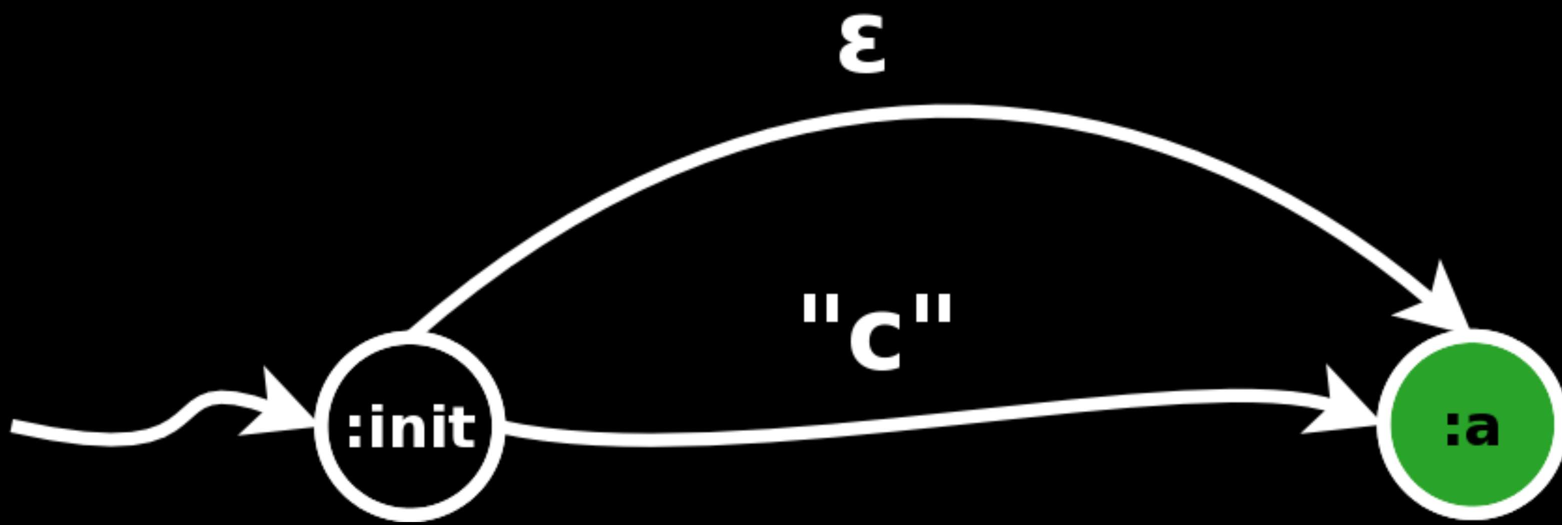
```
my_simple = NFA.simple(“c”)
```

NFA.simple(char)

```
def NFA.simple(alphabet,symbol)
  NFA.new(
    [:init,:a],           # states
    alphabet,             # alphabet
    lambda do |st,sym|    # transition
      (st==:init and sym==symbol) ?
        [:a] : []
    end,
    :init,                # start state
    [:a])                 # accepting states
end
```

NFA::question_mark

```
my_simple.question_mark
```

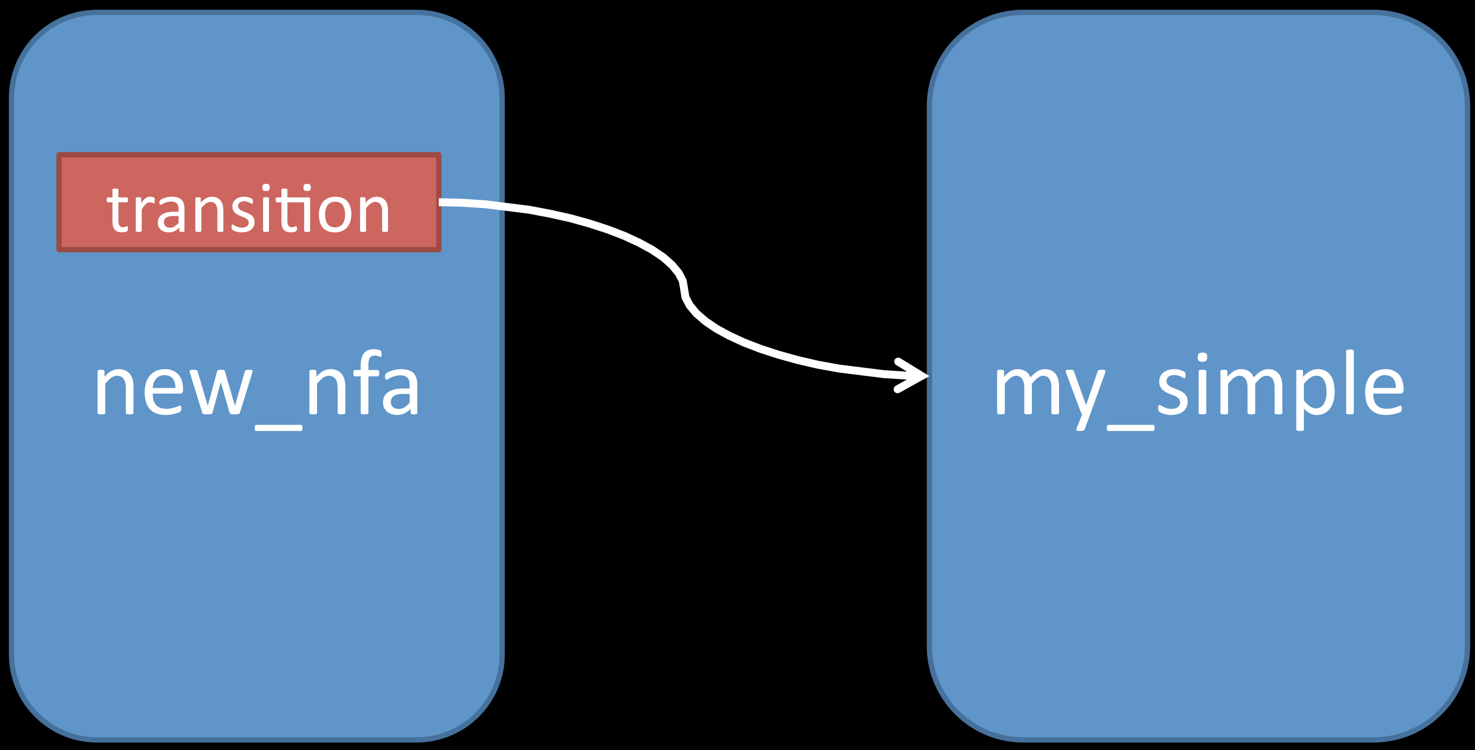


NFA::question_mark

```
def question_mark
  trans = lambda do |st, sym|
    original = @transition.call(st,sym)
    if(st == @start and sym.nil?)
      original += @accepting
    end
    return original
  end
  NFA.new(@states, @alphabet, trans,
          @start, @accepting)
end
```

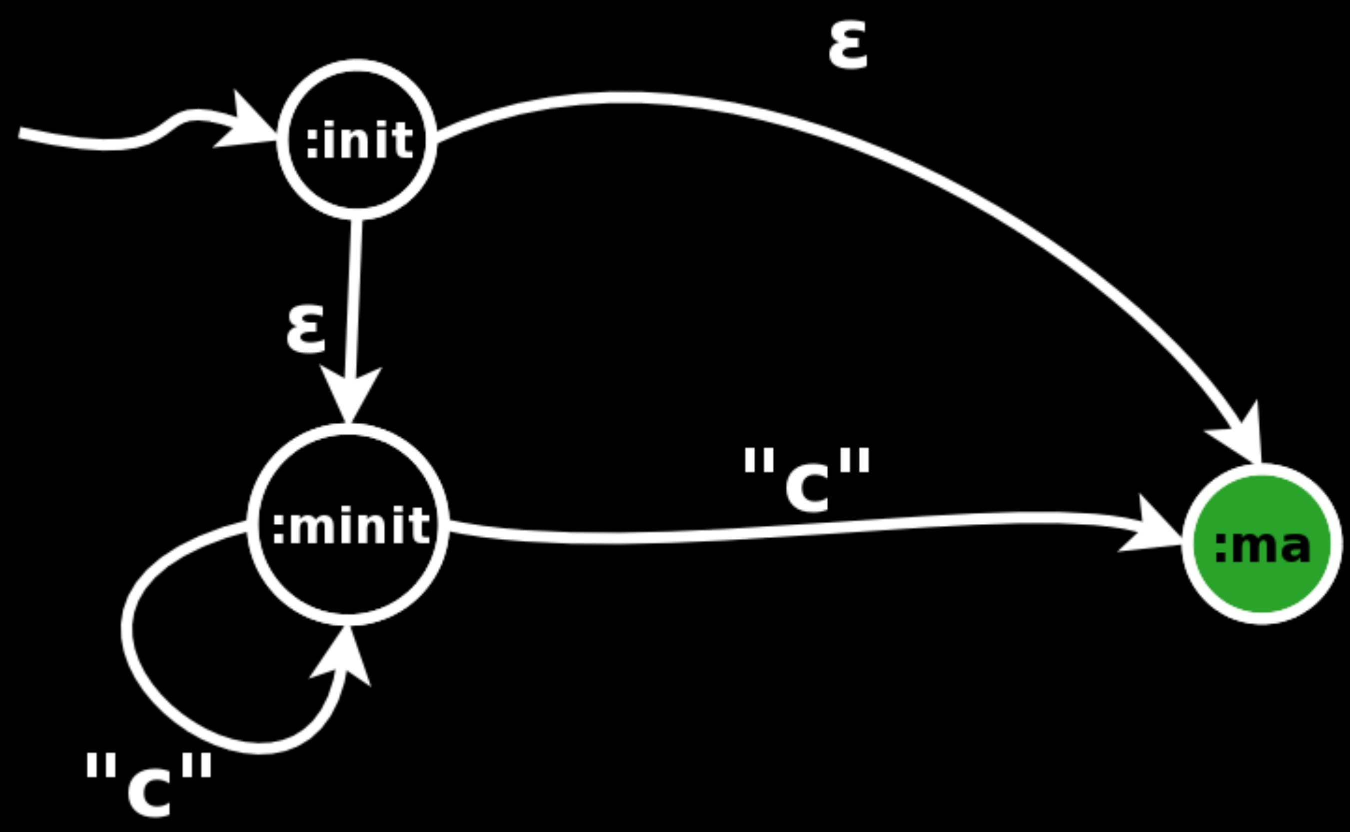
What's going on here? (closures)

new_nfa = my_simple.question_mark



NFA::star

```
my_simple.star
```



NFA::star

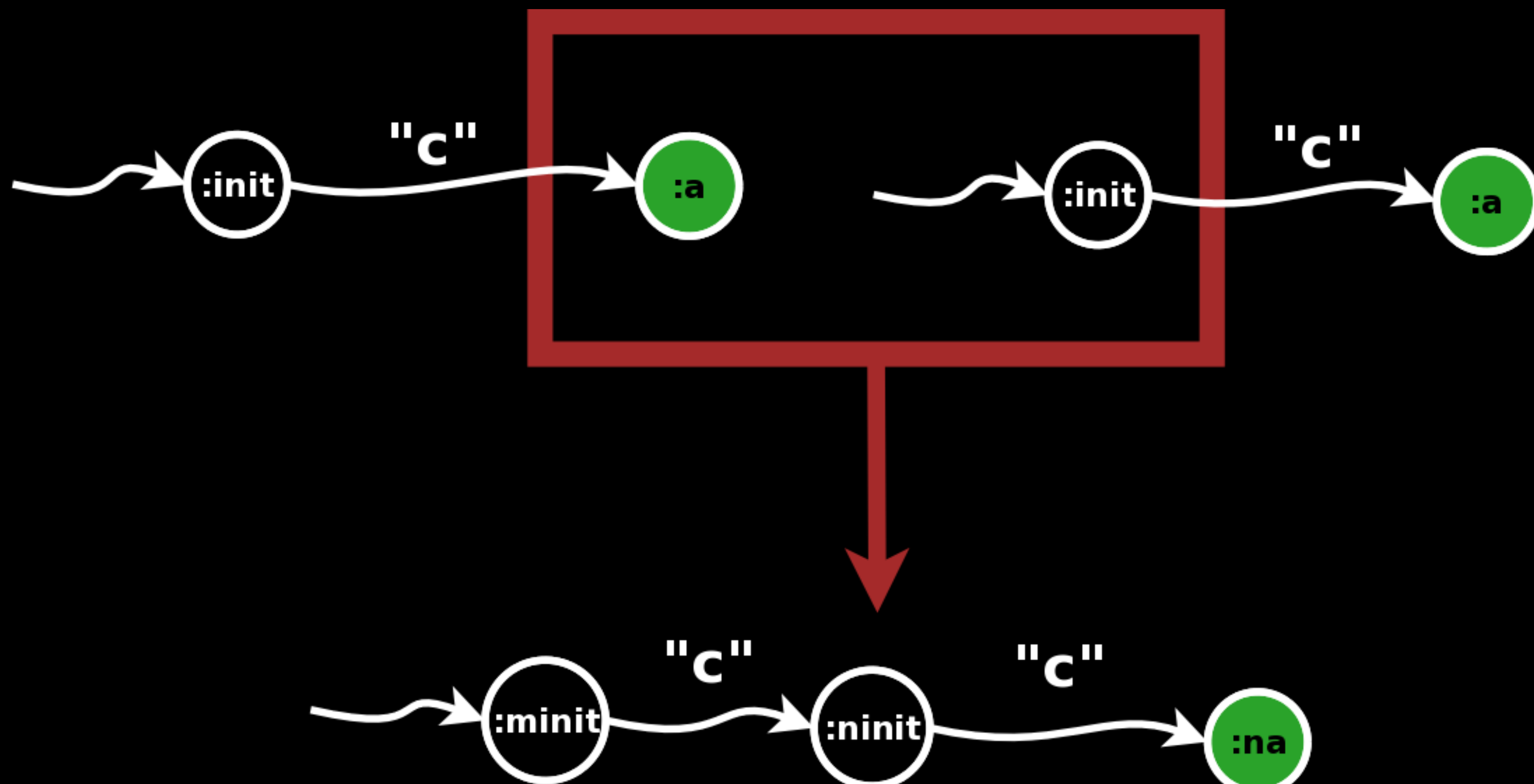
```
def star
  a = self.wrap(“m”)
  states = a.states + [:init]
  transition = lambda do |st,sym|
    if(state==:init)
      if(symbol.nil?)
        return [a.start]+a.accepting
    else
      ret = a.transition.call(st,sym)
      if(a.accepting.any?{|s|ret.include?(s)})
        ret << a.start
      end
      return ret
    end
  end
```

<cont> -- NFA::star

```
  NFA.new(states,
          @alphabet,
          transition,
          :init,
          a.accepting)
end
```

NFA::concatenate

```
my_simple.concatenate(my_simple)
```



NFA::concatenate(other)

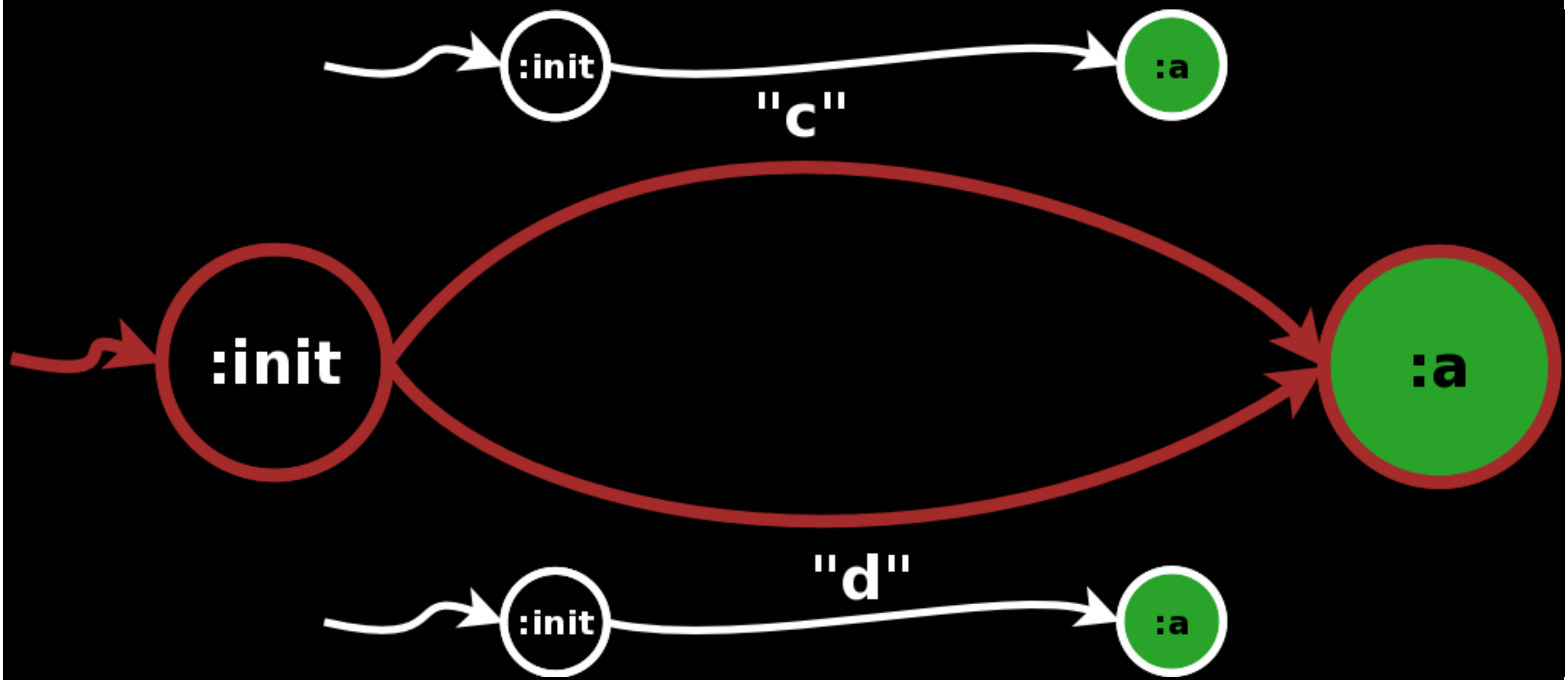
```
def concatenate(other)
  a=self.wrap(“m”)
  b=other.wrap(“n”)
  states = a.states-a.accepting+b.states
  transition = lambda do |st, sym|
    if(a.states.include?(state))
      a.transition.call(state,symbol).
        map{|s|a.accepting.include?(s) ?
          b.start : s}
    else
      b.transition.call(state,symbol)
    end
  end   # continuing…
```

<cont> -- NFA::concatenate(other)

```
  NFA.new(states,
          @alphabet,
          transition,
          a.start,
          b.accepting)
end
```

NFA::or

```
NFA.simple("c").or(NFA.simple("d"))
```



NFA::or(other)

```
def or(other)
  a = self.wrap(“m”)
  b = self.wrap(“n”)
  states = a.states +
        b.states +
        [:init, :accept] –
        [a.start, b.start,
         a.accepting, b.accepting]
```

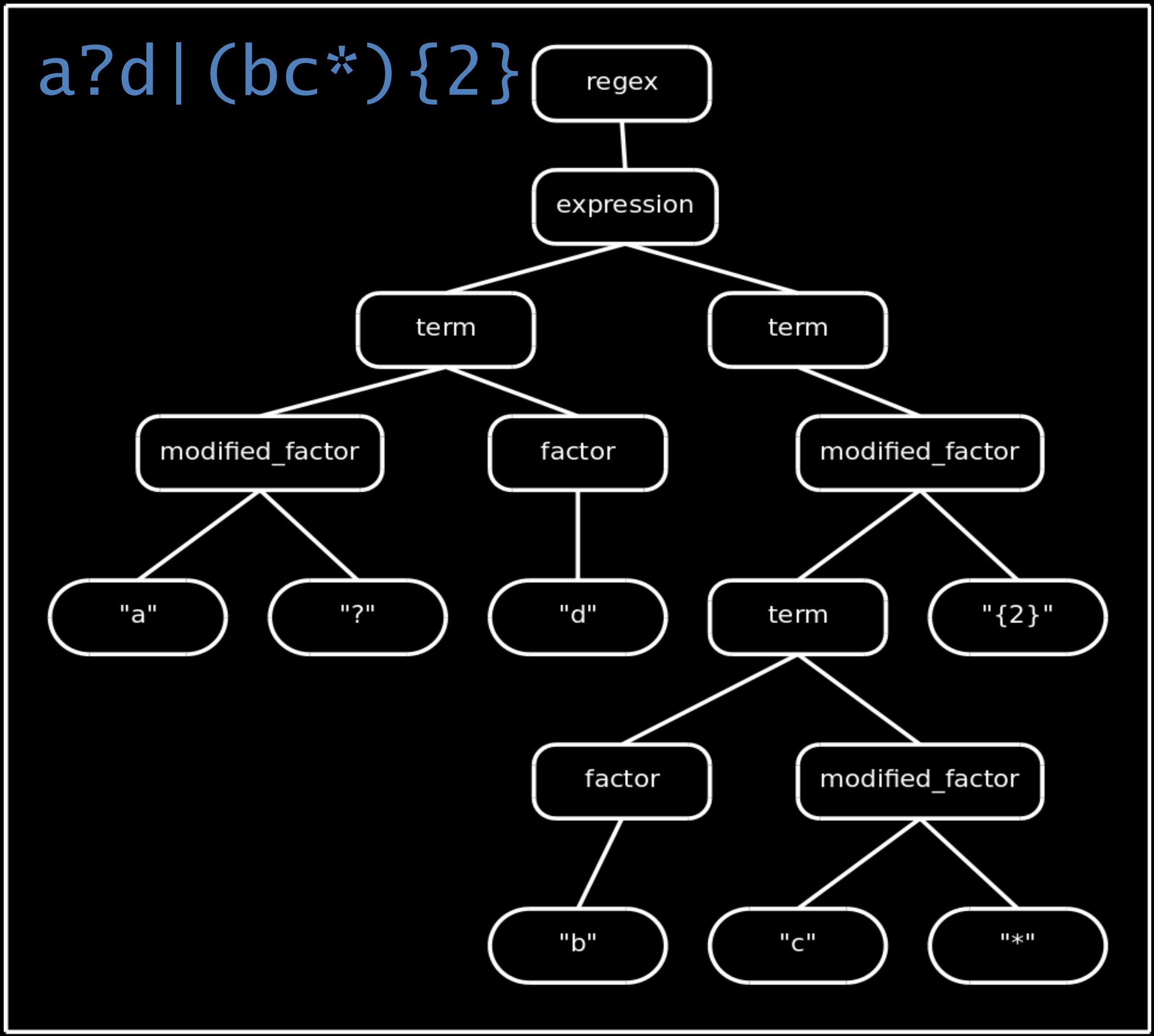
<cont> -- NFA::or(other)

```
transition = lambda do |st, sym|
  ret=
    if(st==:init)
      a.transition.call(a.start,sym)+
        b.transition.call(b.start,sym)
    elsif(a.states.include?(st))
      a.transition.call(st,sym)
    else
      b.transition.call(st,sym)
    end
  return ret.map do |s|
    [a.accepting+b.accepting].
      include?(s) ? :accept : s
  end
end
```

<cont> -- NFA::or(other)

```
    NFA.new(states,
            @alphabet,
            transition,
            :init,
            [:accept])
end
```

Syntax Tree Translation



Conclusion

I enjoyed making this.

Questions?

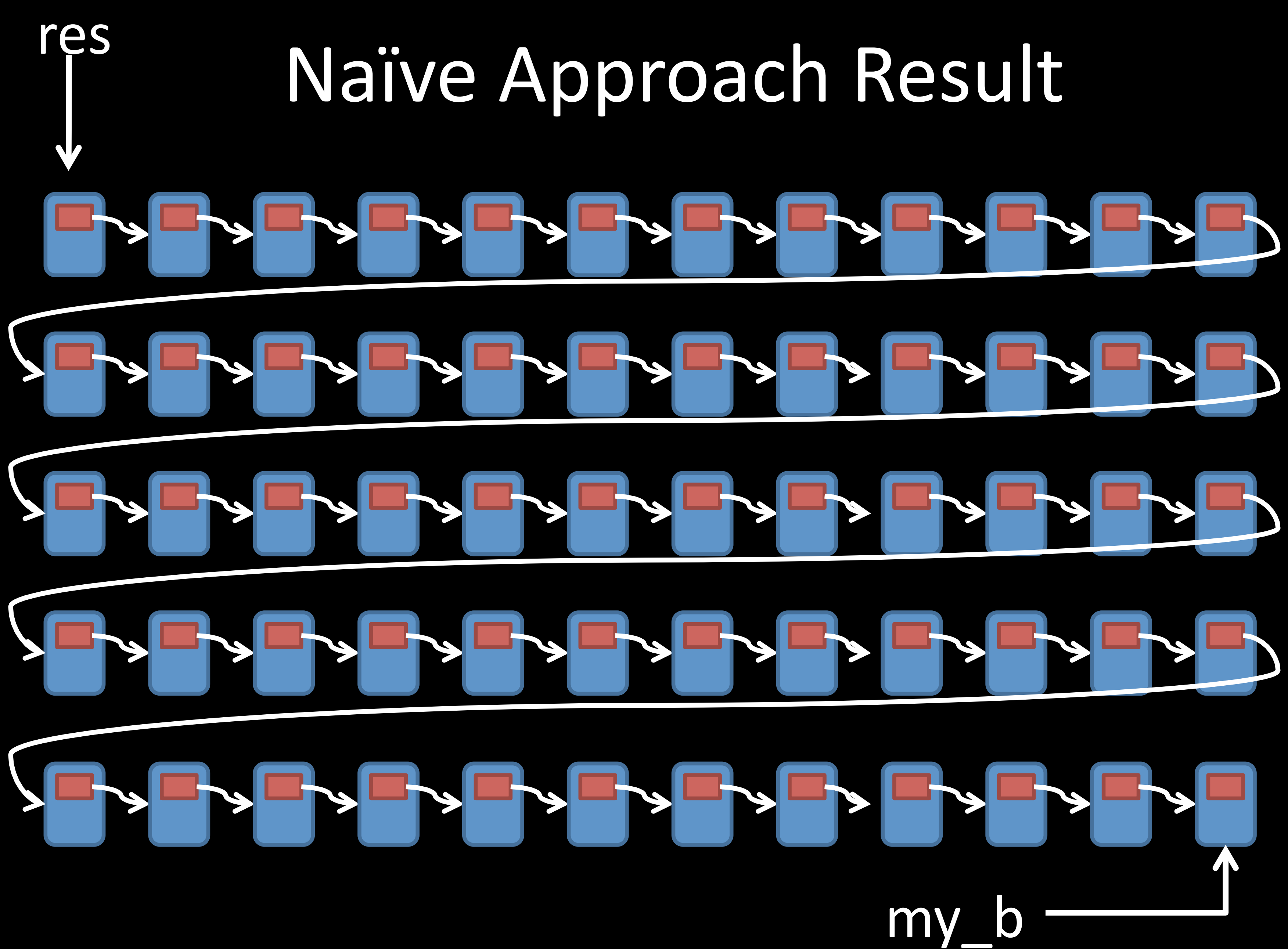
Optimization: Repetition

What do we do for the regular expression b {200}?

Naïve:

```
my_b = NFA.simple("b")
res=my_b
199.times do
  res=res.concatenate(my_b)
end
```

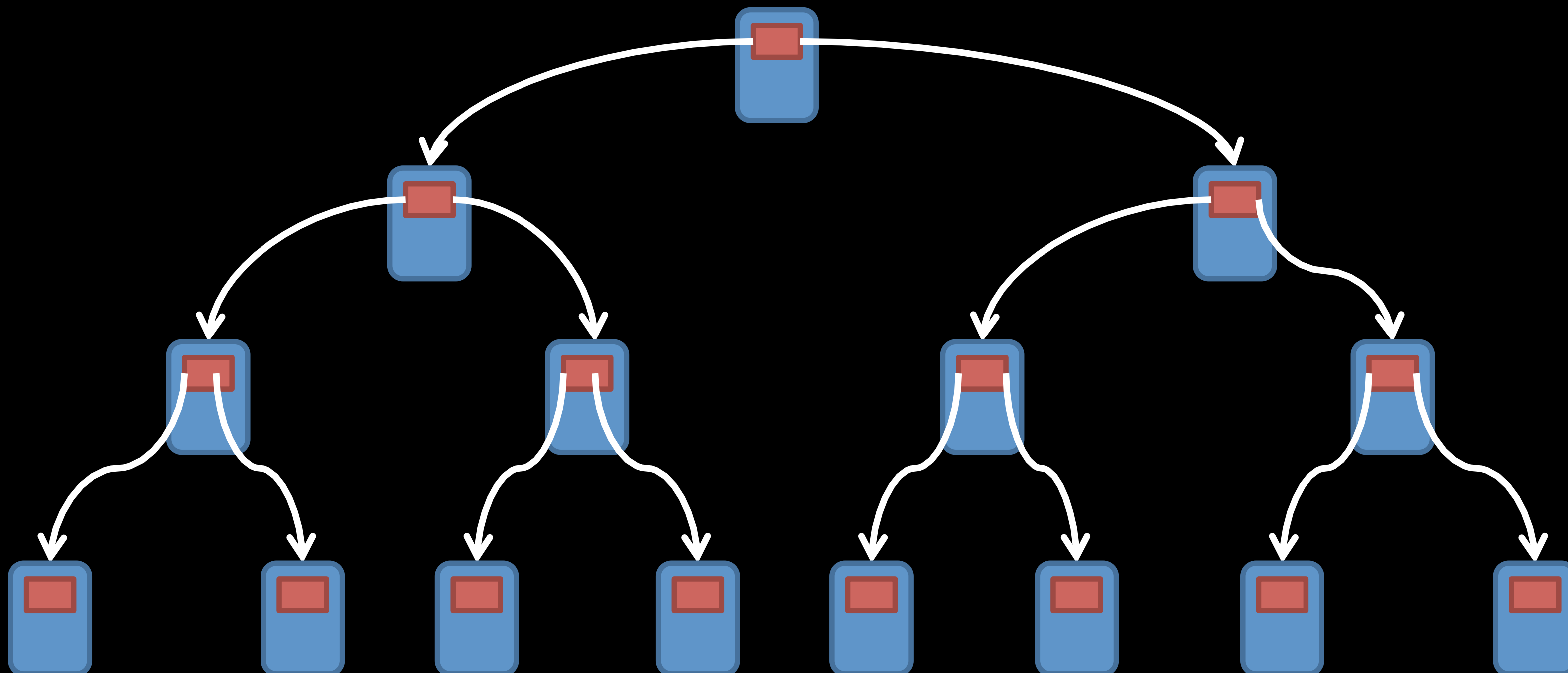
Naïve Approach Result



Better Idea – Divide and Conquer

```
def times(n)
  return self if(n==1)
  a = self.times(n/2)
  b = self.times(n-n/2)
  return a.concatenate(b)
end
```

Divide and Conquer Result



Conclusion

I enjoyed making this.

Questions?