



Bringing the simplicity of Ruby
to syntactic analysis.

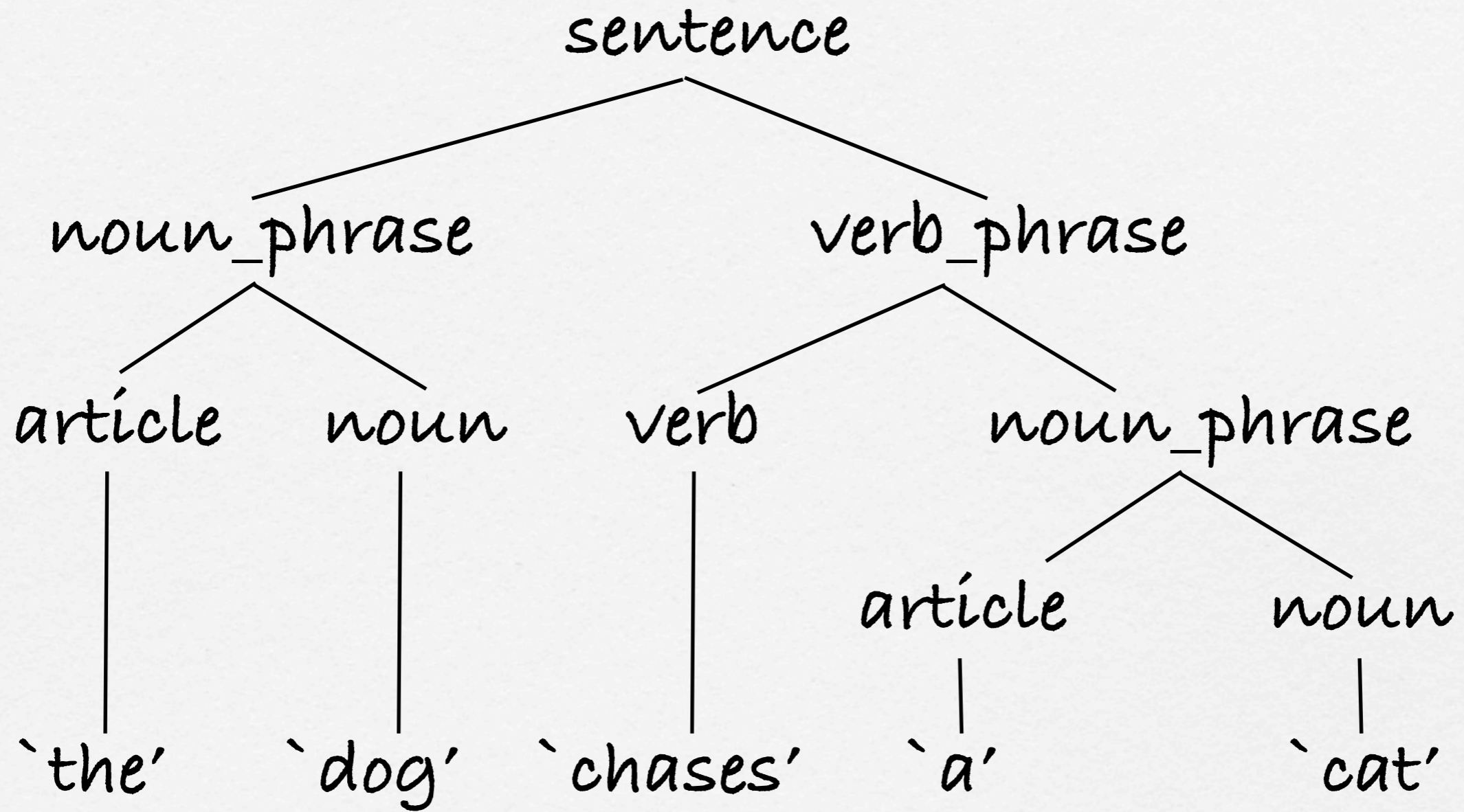
**Presented by Clifford Heath,
Data Constellation.**

Parsing = syntax recognition

- Regular Expressions
 - Can't handle recursive definitions
- Regex + ad-hoc code
- Context-free grammars
 - CFG generates all legal sentences

Context-free grammar

- $\text{sentence} \rightarrow \text{noun_phrase verb_phrase '.'}$
- $\text{noun_phrase} \rightarrow \text{article noun}$
- $\text{verb_phrase} \rightarrow \text{verb} \mid \text{verb noun_phrase}$
- $\text{article} \rightarrow \text{'a'} \mid \text{'the'}$
- $\text{noun} \rightarrow \text{'dog'} \mid \text{'cat'}$
- $\text{verb} \rightarrow \text{'bites'} \mid \text{'sleeps'} \mid \text{'chases'}$



Parsing techniques

- Left-Left (Top down) parsing - LL
 - implemented by recursive descent
- Left-Right (bottom up) parsing - LR
 - needs a parser generator
- Left and Left-Right parsing - LALR
 - e.g. yacc
- All these rely on limited lookahead, e.g. LALR(2)
- Parsing Expression Grammar (PEG) - new technique!

but where is the ambiguity?

- if expr stmt
- if expr stmt else stmt
- if exp stmt if e2 stmt else stmt...?
- Parsing techniques differ in their handling
- CFGs encode no rule priorities
 - Requires unlimited backtracking
 - $O(n^m)$ time

PEG (packrat) parsing

- 'if' expr stmt 'else' stmt / 'if' expr stmt
- / encodes alternatives with preference
- PEG parsers are greedy
 - take first alternative if possible
 - Backtrack if not
- Results are memoized
 - $O(n)$ space, $O(n)$ time

Greed is good...?

- Care is needed:
 - `word* 'end'`
 - word eats up too much input
 - then 'end' fails
- `(!'end' word)* 'end'`
 - Much better!
- Must avoid left-recursion!

Basic Treetop constructs

- Terminals:
 - string literals, e.g. "foo", 'bar'
 - character classes, e.g. [a-z]
 - anything: . (match any single char)
- non-terminals, defined by 'rule'
- alternatives, separated by /
- sequence, separated by space (maybe with parentheses)
- zero-or-more (*), one-or more (+), optional (?)
- lookahead: negative (!), positive (&)

Arithmetic Example

```
rule expr
  term ( add_op term )*
end
rule term
  factor ( mul_op term )?
end
rule factor
  variable / number / `( ` expr ` )`
end
rule variable
  [a-zA-Z_] [a-zA-Z0-9]*
end
rule number
  [1-9] [0-9]* / `0`
end
rule add_op `+` / `-` end
rule mul_op `*` / `/` / `%` end
```

*Note right-recursion!
Makes it right-associative.*

*This example
doesn't account
for white space!*

Lex Rex?

- Packrat parsers typically don't LEX
- Every character must be accounted for, including white space...

```
rule white
  [ \r\t\n]
end
rule expr
  white* expr ( add_op term )*
end
... etc.
```

Comments etc

```
rule comment_c_style
```

```
  '/'* (!'*/' . )* '*/'
```

```
end
```

```
rule white
```

```
  [ \t\n\r ]+
```

```
end
```

```
rule S
```

```
  ( white / comment_c_style )+
```

```
end
```

```
rule s
```

```
  S?
```

```
end
```



illustrates
negative
Look-ahead



Mandatory space



Optional space

Modules, grammars

```
module MyLang
  grammar Arithmetic
    rule ...
  end
end
```

Module is optional!

```
$ tt arith.treetop
$ cat arith.rb
module MyLang
  module Arithmetic
    ...
  end
  ...
  class ArithmeticParser < Treetop::Runtime::CompiledParser
    include Arithmetic
    ...
  end
end
```

Syntax Tree

```
$ ruby -e '  
require "treetop"  
require "arith"  
p MyLang::ArithmeticParser.new.parse("x+2")  
'
```

```
SyntaxNode+Expr1 offset=0, "x+2" (term):  
  SyntaxNode+Term1 offset=0, "x" (factor):  
    SyntaxNode+Factor1 offset=0, "x" (s):  
      SyntaxNode+Variable0 offset=0, "x":  
        SyntaxNode offset=0, "x"  
      SyntaxNode offset=1, ""  
    SyntaxNode offset=1, "+2":  
      SyntaxNode+Expr0 offset=1, "+2" (s,add_op,expr):  
        SyntaxNode offset=1, "+"  
        SyntaxNode+Expr1 offset=2, "2" (term):  
          SyntaxNode+Term1 offset=2, "2" (factor):  
            SyntaxNode+Factor1 offset=2, "2" (s):  
              SyntaxNode+Number0 offset=2, "2":  
                SyntaxNode offset=2, "2"
```

```
$
```

*Nodes for
white-space
have been
stripped out!*

Actions

□ Add methods to `SyntaxNodes`:

```
rule number
  ([1-9] [0-9]* / `0`)
  { def value
    text_value.to_i
  end
end
```

Note the parentheses.
Action blocks bind
tighter than /

Every `SyntaxNode`
has `text_value` method

Actions II

```
rule expr
  s term seq:( s add_op s term )* s
  { def value
    seq.elements.inject(term.value) { | a, t |
      t.add_op.operate(a, t.term.value)
    }
  }
end
end
```

Note the seq: label

*Every SyntaxNode
has elements method*

```
rule add_op
  '+' { def operate(a, b); a+b end }
  / '-' { def operate(a, b); a-b end }
end
```

Custom SyntaxNodes

rule variable

[a-zA-Z]+ <VarNode>

end

*instantiates subclass VarNode
instead of SyntaxNode*

- *As you'd expect, the VarNode class can be re-opened elsewhere in your code.*

Demonstration

Resources

- ❑ `sudo gem install treetop`
- ❑ <http://treetop.rubyforge.org>
- ❑ http://rubyconf2007.confreaks.com/d1t1p5_treetop.html
- ❑ <http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf>

Questions