Recursive Descent Parsing

with ruby

Who am I?

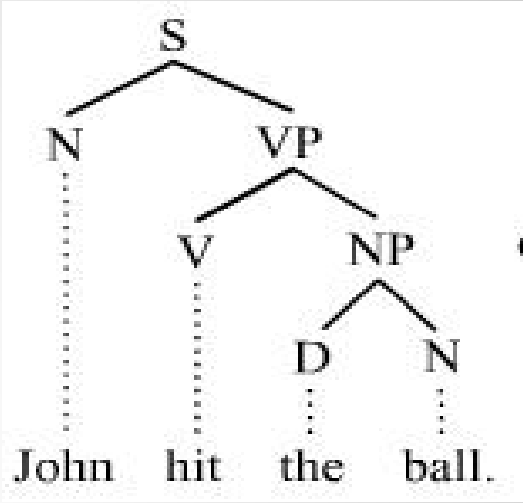
Nick Evans

410 Labs

Mailstrom.co

Working on several patches for Net::IMAP::ResponseParser.

What is parsing?



Analyzing an input stream
according to the rules of a grammar,
(usually) resulting in a parse tree.

Why would we want to parse?

- Natural Language Processing
- Network protocols
  - RFC 5234: Augmented BNF for Syntax Specifications
- Compiler design
- Tools: Syntax highlighting, refactoring, etc
- Domain Specific Language

Goals

- *Very basic* understanding of:
  - parsing terminology
  - parsing techniques

- Comfort with:
  - reading hand-written RD parser
  - writing hand-written RD parser

Why not just use RegExp?

Because regular expressions are not the simple answer when the problem is hard.

Glossary

- lexer
- tokens
- ambiguous grammar
- backtracking
- lookahead
- associativity, precedence
- left recursion

Lexer and Tokens

Lexer: parses the input stream into tokens.

Tokens: meaningful groupings of characters.

Usually implemented with regular expressions.

Ambiguous Grammar

more than one possible parse tree for
a given expression

Police help dog bite victim.

Ambiguous Grammar

more than one possible parse tree for a given expression

Police help dog bite victim.

“One morning I shot an elephant in my pajamas. How he got in my pajamas, I don't know.”

-- Groucho Marx

Ambiguous Grammar

more than one possible parse tree for a given expression

A = A + A | A * A | [0-9]

- 4 + 2 * 5 = (4+2) * 5 = 6 * 5 = 30
- 4 + 2 * 5 = 4 + (2*5) = 4 + 10 = 14

Police help dog bite victim.

“One morning I shot an elephant in my pajamas. How he got in my pajamas, I don't know.”

-- Groucho Marx

Time flies like an arrow; fruit flies like a banana

- flies: verb or noun?
- like: preposition or verb?

Backtracking

The horse raced past the barn fell.

Initial parse: The horse raced past the barn... ?!?

The horse -- (that was) raced past the barn -- fell.

Backtracking

The old man the boat.

Initial parse: The man who is old... ?!?

The elderly operate the boat.

Lookahead

parser uses one or more incoming tokens to decide which rule it should use

Enough definitions

Let’s look at some code!

The following concepts will be discussed when the code forces us to:

- associativity
- precedence
- left-recursion

Code: Lexing with RegExp

1. lexer; parsing(str), next_token, parse_error
    - Token struct (:type, :value)
2. arithmetic tokens lexer (with tests)
    - INTEGER
    - FLOAT
    - RPAR
    - LPAR
    - PLUS
    - MINUS
    - TIMES
    - DIV

Code: Lookahead

1. lexer; parsing(str), next_token, parse_error
2. arithmetic tokens lexer (with tests)
3. lookahead: lookahead, lookahead(k), shift_token
4. lookahead convenience helpers: match, accept, lookahead?(t1, t2, t3)

Analyzing the grammar

1. lexer; parsing(str), next_token, parse_error
2. arithmetic tokens lexer (with tests)
3. lookahead: lookahead, lookahead(k), shift_token
4. lookahead convenience helpers: match, accept, lookahead?(t1, t2, t3)
5. arithmetic expression grammar: parse(str)
   - `expression = term | expression “+” term | expression “-” term`
   - `term = factor | term “*” factor | term “/” factor`
   - `factor = number | “(“ expression “)”`
   - `number = integer | float`
   - `integer = nonzero digit* | “0”`
   - `float = integer “.” digit digit*`
   - `nonzero = [1-9]`
   - `digit = [0-9]`

Analyzing the grammar

1. lexer; parsing(str), next_token, parse_error
2. arithmetic tokens lexer (with tests)
3. lookahead: lookahead, lookahead(k), shift_token
4. lookahead convenience helpers: match, accept, lookahead?(t1, t2, t3)
5. analyze the grammar: left-recursion and lexer handled

```
expression = term   | expression "+" term   | expression "-" term
term       = factor | term       "*" factor | term       "/" factor
factor     = number | "(" expression ")"
number     = integer | float
integer    = nonzero digit* | "0"        /* lexer handles this */
float      = integer "." digit+          /* lexer handles this */
nonzero    = [1-9]                       /* lexer handles this */
digit      = [0-9]                       /* lexer handles this */
```

Code: implement grammar rules

1. lexer; parsing(str), next_token, parse_error
2. arithmetic tokens lexer (with tests)
3. lookahead: lookahead, lookahead(k), shift_token
4. lookahead convenience helpers: match, accept, lookahead?(t1, t2, t3)
5. implement new grammar (with no left recursion, but retaining left-associativity)
    - `expression = term   (“+” term   | “-” term)*`
    - `term       = factor (“*” factor | “/” factor)*`
    - `factor     = number  | “(“ expression “)”`
    - `number     = integer | float`