Parsing Overview

Last time: How lexer generators work

Spec = R1 { Action1 }
| R2 { Action2 }
| R3 { Action 3 }
... // in decreasing priority

$\longrightarrow$ NFA $\longrightarrow$ DFA $\longrightarrow$ code

RE Derivatives

Sometimes precomputing NFA/DFA is not worth the trouble or cost => RE libraries

One elegant idea to implement such a RE library: interpreting RE directly   Brzozowski

$$RE_0 \xrightarrow{x_1} RE_1 \xrightarrow{x_2} RE_2 \longrightarrow \cdots \longrightarrow \varepsilon$$

$$\partial_x R = R'$$

$$\partial_a\, a = \varepsilon \qquad \partial_a (R_1 + R_2) = \partial_a R_1 + \partial_a R_2$$

$$\partial_a\, \varepsilon = \emptyset \qquad \partial_a (R^*) = (\partial_a R)\, R^*$$

$$\partial_a\, b = \emptyset \quad b \neq a \qquad \partial_a (R_1 R_2) = (\partial_a R_1) R_2 + \nu(R_1)(\partial_a R_2)$$

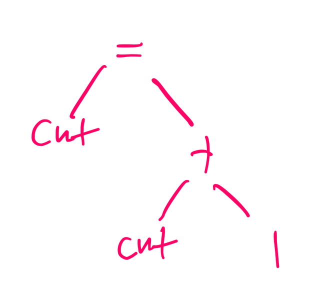
$$\nu(R_1) = \begin{cases} \varepsilon & \text{if } R_1 \text{ accepts empty str} \\ \emptyset & \text{otherwise} \end{cases}$$

Today: An Overview of Parsing

Where we are:

characters $\xrightarrow{\text{lexing}}$ tokens $\xrightarrow{\text{parsing}}$ AST

cnt = cnt + 1



Syntax specification: context-free grammar (CFG)

$S \to S + S$
$S \to S * S$
$S \to E$
$E \to n$
$E \to (S)$

- productions
- nonterminal   S, E
- terminal symbols   + * n ( )
- symbols: terminals or nonterminals
- one of the symbols is the start symbol

expressive power comes from recursion (compare with regular expressions)

Some definitions:
- L(G): the language of grammar G
- token stream ∈ L(G): there is a **derivation** using G

Derivation

Derivation: a sequence of rewrite steps from start symbol to token stream

$$\alpha X \beta \longrightarrow \alpha \gamma \beta \quad \text{where } X \to \gamma \text{ is a production}$$

An example derivation:

1 + 2 * 3   1 + 2 + 3

Grammar:

$\boxed{\begin{array}{l} S \to S + S \\ S \to S * S \\ S \to E \\ E \to n \\ E \to (S) \end{array}}$

Token stream: (1 + 4) + 2

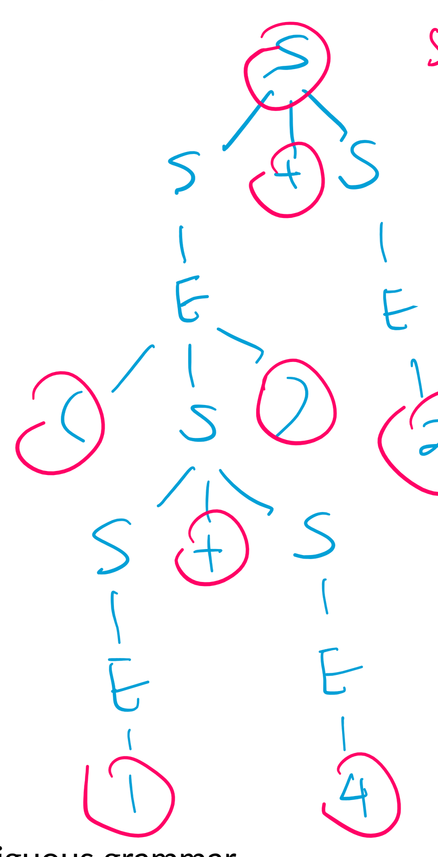
Q: How would we derive the token stream using the grammar?

$$S \to S + S \to E + S \to (S) + S \to (S + S) + S \to \cdots \to (1+4) + S \to (1+4) + E \to (1+4) + 2$$

Leftmost derivation: expand leftmost nonterminal
turns out to correspond to top-down parsing
Rightmost derivation: expand rightmost nonterminal
turns out to correspond to bottom-up parsing

Parse tree

Derivation generates a parse tree.



start symbol

tokens

Ambiguous grammar

∃ stream of tokens in L(G) s.t. > 1 parse trees for that stream

Fixing ambiguity

Specifying precedence and associativity using additional nonterminals

$S \longrightarrow S + T \mid T$
term $T \to T * F \mid F$
factor $F \to n \mid (S)$

Some parser generators allow precedence/associativity declarations:

cup
```
precedence left PLUS;
precedence left TIMES;
```

Dangling-else problem

$S \to$ if E then S | if E then S else S | ...

if E1 then (if E2 then S1 else S2)

cup
```
precedence nonassoc IF;
precedence nonassoc ELSE;
```

$S \to U \mid M$
$M \to$ if E then M else M
| ...
$U \to$ if E then S
| if E then M else U

- Statements ending w/ unmatched THEN
- Otherwise

Needing context

Example in C++:
HashTable<K,V> x;

int x, K, V, HashTable;

Significant whitespace

Top-Down Parsing.   LL parser

Example   S-expressions (McCarthy. Lisp)

(let ((x z) (+ x 1)))   let x = z in x + 1

$S \to (L) \mid x$
$L \to \varepsilon \mid S\ L$

parse (a b)

recursive descent

```
parseS () {
  switch (peek()) {
  case "(":      // S -> (L)
    consume "(";
    parseL ();
    consume ")";
    return;
  case ID:       // S -> x
    consume ID;
    return
  default:
    throw SyntaxError;
}
```

```
parseL () {
  switch (peek()) {
  case ID:       // L -> SL
  case "("       // L -> SL
    parse S()
    parse L(); return
  case ")"       // L -> ε
    return
  default:
    throw SyntaxError
}
```