Syntactic analysis or parsing

Stream of tokens → Parser → Parse tree/syntax error

Q: Will regular grammar/expression work?

Mohammad T. Irfan
RD parser for assignment stmt

Assignment → Id = Expr;
Expr → Term {AddOp Term}
AddOp → + | -
Term → Factor {MulOp Factor}
MulOp → * | /
Factor → [UnaryOp] Primary
UnaryOp → -
Primary → Id | IntLiteral | FloatLiteral | (Expr)
... <Lexical syntax for Id, IntLiteral, FloatLiteral> ...

Python code for smaller version

Expr → Term {(+|-) Term}
Term → Factor {(*|/) Factor}
Factor → IntLiteral
... <Lexical syntax for IntLiteral> ...
Announcement

- Exam 1 on Wednesday, 2/26
  - Scope: up to CYK
  - 1:15 to 1:55 in class (40 minutes), followed by lecture
- Accommodation:
  - 12:55 to 1:55 in VAC 309
  - 10:05 to 11:05 in VAC 309

Requirements for RD parser

1. Remove left recursions (why?)
2. Do "left factoring"
Removing left recursion

Example

Algorithm (assume no cycle; i.e., no $A \Rightarrow A$)

Nonterminals: $A_1, A_2, \ldots, A_n$ (ordered arbitrarily)

For $i = 1$ to $n$

For each $j < i$

Let $A_j \Rightarrow \delta_1 | \delta_2 | \ldots | \delta_k$

Replace each $A_i \Rightarrow A_j \gamma$ by $A_i \Rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma$

Eliminate left recursion from all $A_i$ products

---

Left factoring

- IfStmt $\Rightarrow$ if Expr then Stmt
- IfStmt $\Rightarrow$ if Expr then Stmt else Stmt
- Why can’t RD parser deal with it?
- Solution
  - Find the largest prefix $\alpha$ and factor it out
    $A \Rightarrow \alpha \beta_1 | \alpha \beta_2$
    
    $A \Rightarrow \alpha A'$
    $A' \Rightarrow \beta_1 | \beta_2$
Literature review

- NP-hard: Given a CFG, is there an LL(1) parser?
- Impossibility example:
  \( L_G = \{a^n b^n | n \geq 1\} \cup \{a^n 1 b^{2n} | n \geq 1\} \)
- Why is an LL(1) impossible?

Literature review

- Is there always a parser (not necessarily LL(1)) for any CFG?
- **CYK algorithm**: Cocke & Younger (1967) and Kasami (1965)
  - First parser for any CFG
  - Bottom-up parser
- **Frost (2007)**: First top-down parser for any CFG; improved by Ridge (2014)
CYK Parsing Algorithm


What it does

- Given (1) a CFG and (2) a string, verifies whether the string can be derived by this grammar
- Example
  - Detects syntactic errors in a given C program
Requirements

- CFG must be in **Chomsky Normal Form (CNF)**
  - $A \rightarrow BC$
  - $A \rightarrow a$
- No $\varepsilon$ in any product
- OK to have left recursion!
- Left factoring is out of question (why?)
- Example

---

CYK Idea

- Bottom-up approach + dynamic programming
- Start with individual symbols of input string
- Combine multiple symbols into nonterminal
  - Any 2 consecutive symbols
  - Any 3 consecutive symbols
  - ...
- Climb up the hierarchy toward start symbol
- Yes answer to parsing

we can get to the start symbol
CYK example (cont...)

Input string: 2 - 3 * 4

Length

Start index

j

1

2

3

4

5

Expr

Term

Factor

AddOp

MultOp

Expr → Expr X
X → Expr Y
AddOp → + | -
Expr → Term Y
Term → Term Y
Y → MultOp Factor
MultOp → * | /
Factor → 0 | 1 | ... | 9
Term → 0 | 1 | ... | 9
Expr → 0 | 1 | ... | 9

CYK Algorithm

Inputs: CNF grammar and n tokens
Fill in the row for length 1
For each length i from 2 to n:
  For each index j from 1 to n-i+1:
    A → BC?
    For k = length of B from 1 to i-1:
      If there's a product A → BC s.t.
      B is in cell (j,k) and
      C is in cell (j+k, i-k):
        Add A to cell (j,i)

Return True iff cell (1,n) contains the start symbol.
Negative example

- Input string:
  \[2 + 3 * /\]

Practice problem

- CNF grammar
  \[S \rightarrow AX \mid AB\]
  \[X \rightarrow SB\]
  \[A \rightarrow 0\]
  \[B \rightarrow 1\]

- Parse the following strings using CYK
  - 0011
  - 01010 \(\times\)

- Work in groups