Syntax Analysis:
Context-free Grammars, Pushdown Automata and Parsing Part - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis?
- Specification of programming languages: context-free grammars
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Grammars

- Every programming language has precise grammar rules that describe the syntactic structure of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.
- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars
- Parsers or syntax analyzers are generated *for* a particular grammar
- Context-free grammars are usually used for syntax specification of programming languages

What is Parsing or Syntax Analysis?

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on *context-free* grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Context-free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)	(2)	(3)	(4)
$E \rightarrow E + E$	$S \rightarrow 0S0$	$S \rightarrow aSb$	$S \rightarrow aB \mid bA$
$E \rightarrow E * E$	$S \rightarrow 1S1$	$S \rightarrow \epsilon$	$A \rightarrow a \mid aS \mid bAA$
$E \rightarrow (E)$	$S \rightarrow 0$		$B \rightarrow b \mid bS \mid aBB$
$E \rightarrow id$	$S \rightarrow 1$		
	$S \rightarrow \epsilon$		

Derivations

- $E \Rightarrow^{E \rightarrow E+E} E + E \Rightarrow^{E \rightarrow id} id + E \Rightarrow^{E \rightarrow id} id + id$
  is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \rightarrow E + E$, $E \rightarrow id$, and $E \rightarrow id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as,
  $E \Rightarrow^* id + id$, and is read as S **derives** $id + id$

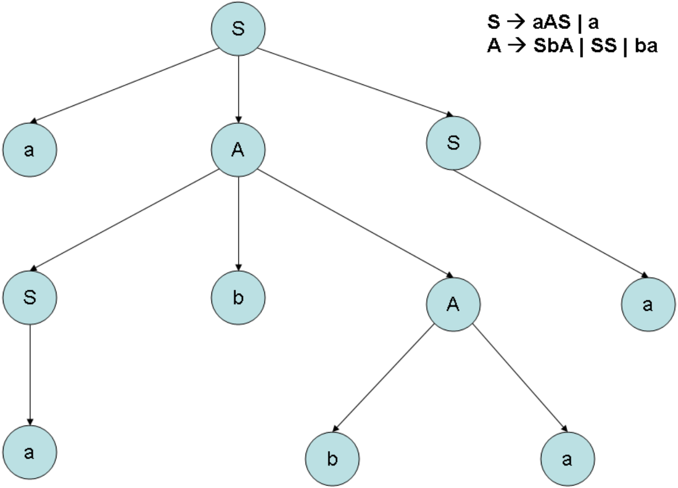
Context-free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by G*, denoted $L(G)$, is
  $L(G) = \{w \mid w \ \in T^*, \ and \ S \Rightarrow^* \ w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S
- Examples
  1. $L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'
  2. $L(G_2)$ = Set of palindromes over 0 and 1
  3. $L(G_3) = \{a^n b^n \mid n \geq 0\}$
  4. $L(G_4) = \{x \mid x \ has \ equal \ no. \ of \ a's \ and \ b's\}$
- A string $\alpha \ \in \ (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \ \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in$ P, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Derivation Tree Example



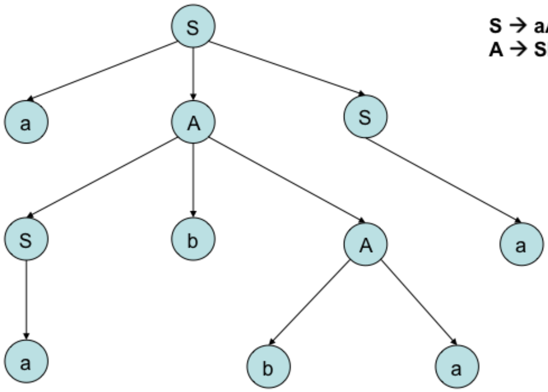
S → aAS | a
A → SbA | SS | ba

S => aAS => aSbAS => aabAS => aabbaS => aabbaa

Leftmost and Rightmost Derivations

- If at each step in a derivation, a production is applied to the leftmost nonterminal, then the derivation is said to be **leftmost.** Similarly **rightmost derivation.**
- If $w \in L(G)$ for some G, then w has at least one parse tree and corresponding to a parse tree, w has unique leftmost and rightmost derivations
- If some word w in $L(G)$ has two or more parse trees, then G is said to be **ambiguous**
- A CFL for which every G is ambiguous, is said to be an **inherently ambiguous** CFL

Leftmost and Rightmost Derivations: An Example



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S => aAS => aSbAS => aabAS => aabbaS => aabbaa**

Rightmost derivation: **S => aAS => aAa => aSbAa => aSbbaa => aabbaa**

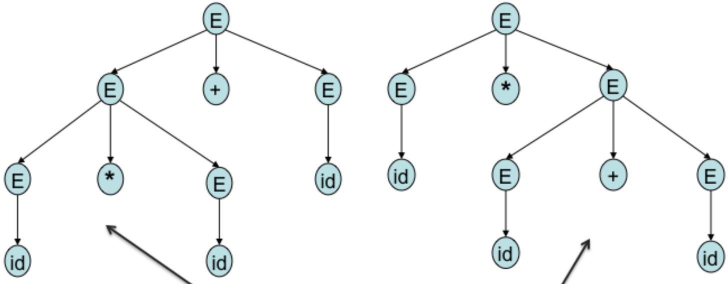
Ambiguous Grammar Examples

- The grammar, $E \rightarrow E + E | E * E | (E) | id$
  is ambiguous, but the following grammar for the same language is unambiguous
  $E \rightarrow E + T | T,\ T \rightarrow T * F | F,\ F \rightarrow (E) | id$
- The grammar,
  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$

  is ambiguous, but the following equivalent grammar is not

  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ matched\_stmt\ ELSE\ stmt$
  $matched\_stmt \rightarrow$
  $IF\ expr\ matched\_stmt\ ELSE\ matched\_stmt | other\_stmt$
- The language,
  $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$,
  is inherently ambiguous

Ambiguity Example 1

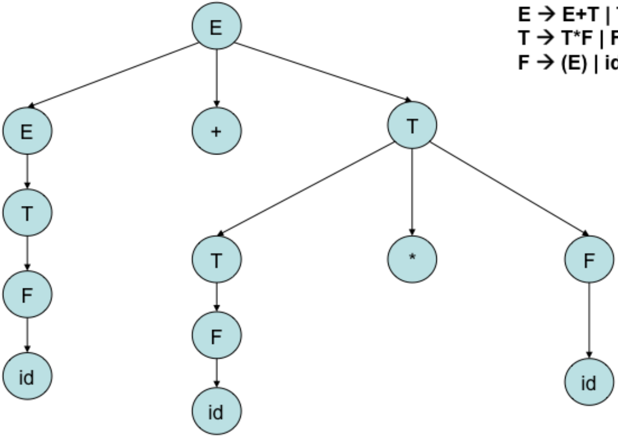


E => E+E => E*E+E => id*E+E => id*id+E => id*id+id

E => E*E => id*E => id*E+E => id*id+E => id*id+id

E → E+E | E*E | (E) | id

Equivalent Unambiguous Grammar

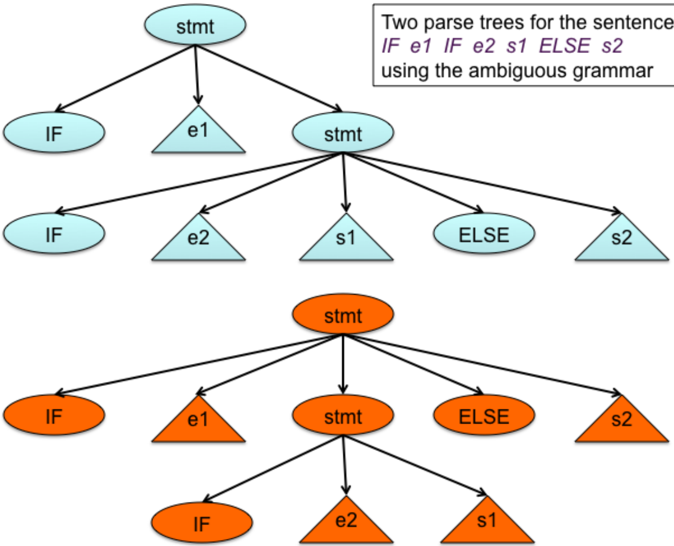


E → E+T | T
T → T*F | F
F → (E) | id

E => E+T => T+T => F+T => id+T => id+T*F => id+F*F => id+id*F => id+id*id

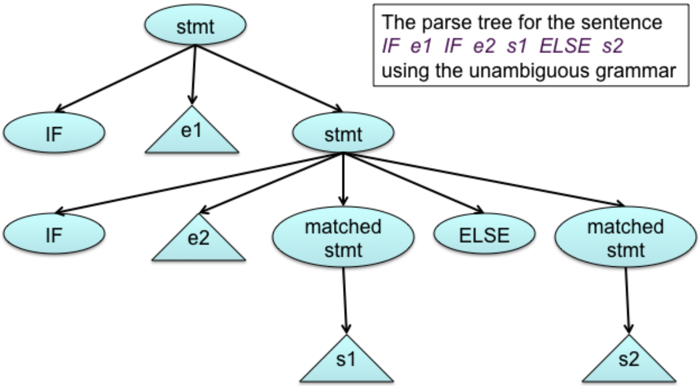
E => T*F => F*F => (E)*F => (E+T)*F => (T+T)*F => (F+T)*F => (id+T)*F
=> (id+F)*id => (id+id)*F => (id+id)*id

Ambiguity Example 2



Two parse trees for the sentence
IF e1 IF e2 s1 ELSE s2
using the ambiguous grammar

Ambiguity Example 2 (contd.)



s→ IF e s | IF e ms ELSE s
ms → IF e ms ELSE ms | other_s

Fragment of C-Grammar (Statements)

```
program --> VOID MAIN '(' ')' compound_stmt
compound_stmt --> '{' '}' | '{' stmt_list '}'
          | '{' declaration_list stmt_list '}'
stmt_list --> stmt | stmt_list stmt
stmt --> compound_stmt| expression_stmt
            | if_stmt | while_stmt
expression_stmt --> ';'| expression ';'
if_stmt --> IF '(' expression ')' stmt
   | IF '(' expression ')' stmt ELSE stmt
while_stmt --> WHILE '(' expression ')' stmt
expression --> assignment_expr
             | expression ',' assignment_expr
```

Fragment of C-Grammar (Expressions)

```
assignment_expr --> logical_or_expr
        | unary_expr  assign_op  assignment_expr
assign_op --> '='| MUL_ASSIGN| DIV_ASSIGN
                 | ADD_ASSIGN| SUB_ASSIGN
                 | AND_ASSIGN| OR_ASSIGN
unary_expr --> primary_expr
        | unary_operator   unary_expr
unary_operator --> '+'| '-'| '!'
primary_expr --> ID| NUM| '(' expression ')'
logical_or_expr --> logical_and_expr
        | logical_or_expr   OR_OP logical_and_expr
logical_and_expr --> equality_expr
        | logical_and_expr   AND_OP  equality_expr
equality_expr --> relational_expr
             | equality_expr EQ_OP relational_expr
             | equality_expr NE_OP relational_expr
```

Fragment of C-Grammar (Expressions and Declarations)

```
relational_expr --> add_expr
                  | relational_expr '<' add_expr
                  | relational_expr '>' add_expr
                  | relational_expr LE_OP add_expr
                  | relational_expr GE_OP add_expr
add_expr --> mult_expr| add_expr '+' mult_expr
                      | add_expr '-' mult_expr
mult_expr --> unary_expr| mult_expr '*' unary_expr
                        | mult_expr '/' unary_expr
declarationlist --> declaration
                  | declarationlist  declaration
declaration --> type  idlist ';'
idlist --> idlist ',' ID | ID
type --> INT_TYPE | FLOAT_TYPE | CHAR_TYPE
```

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Pushdown Automata (contd.)

- The leftmost symbol of γ_i will be the new top of stack
- a in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\},\ \delta(q_1, 0, 0) = \{(q_1, 00)\},$
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\},\ \delta(q_2, 1, 0) = \{(q_2, \epsilon)\},$
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*