CS6109 – COMPILER DESIGN

Module – 4

Presented By
Dr. S. Muthurajkumar,
Assistant Professor,
Dept. of CT, MIT Campus,
Anna University, Chennai.

MODULE - 4

- Context-Free Grammar (CFG)
- Derivation Trees
- Ambiguity in Grammars and Languages
- Need and Role of the parser

CONTEXT-FREE GRAMMAR (CFG)

- The Formal Definition of a Context-Free Grammar
- Notational Conventions
- Derivations
- Parse Trees and Derivations
- Ambiguity
- Verifying the Language Generated by a Grammar
- Context-Free Grammars Versus Regular Expressions

CONTEXT-FREE GRAMMAR (CFG)

- Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.
- Context free grammar G can be defined by four tuples as:
- G= (V, T, P, S)
- Where,
- G describes the grammar
- T describes a finite set of terminal symbols.
- V describes a finite set of non-terminal symbols
- P describes a set of production rules
- S is the start symbol.
- In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

CONTEXT-FREE GRAMMAR (CFG)

- In CFG, the start symbol is used to derive the string.
- You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.
- Example:
- L= {wcw^R | w € (a, b)*}

CONTEXT-FREE GRAMMAR (CFG)

- Production rules:

  $S \rightarrow aSa$

  $S \rightarrow bSb$

  $S \rightarrow c$

- Now check that **abbcbba** string can be derived from the given CFG.

  $S \Rightarrow aSa$

  $S \Rightarrow abSba$

  $S \Rightarrow abbSbba$

  $S \Rightarrow abbcbba$

- By applying the production $S \rightarrow aSa$, $S \rightarrow bSb$ recursively and finally applying the production $S \rightarrow c$, we get the string abbcbba.

CFG – NOTATIONAL CONVENTIONS

- These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, c.

(b) Operator symbols such as +, *, and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1, …,9.

(e) Boldface strings such as id or if, each of which represents a single terminal symbol.

CFG – NOTATIONAL CONVENTIONS

- These symbols are non-terminals:

(a) Uppercase letters early in the alphabet, such as A, B, C.

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as expr or stmt.

(d) When discussing programming constructs, uppercase letters may be used to represent non-terminals for the constructs.

For example, non-terminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

CFG – NOTATIONAL CONVENTIONS

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either non-terminals or terminals.

4. Lowercase letters late in the alphabet, chiefly u, v, …, z, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α the body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …, $A \rightarrow \alpha_k$ with a common head A (call them A-productions), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$. Call α_1 ,α_2 ,…, α_k the alternatives for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

CFG – NOTATIONAL CONVENTIONS

Example:

E → E + T | E → E - T | T

T → T * F | T = F | F

F → (E) | id

- The notational conventions tell us that E, T, and F are non-terminals, with E the start symbol.
- The remaining symbols are terminals.

CFG – DERIVATIONS

- The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules.
- Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.
- Bottom-up parsing is related to a class of derivations known as “rightmost” derivations, in which the rightmost nonterminal is rewritten at each step.
- E → E + E | E * E | (E) | id
- Adds a production E → -E
- E → E + E | E * E | -E | (E) | id
- E => -E

CFG – DERIVATIONS

- E → E + E | E * E | -E | (E) | id
- E => -E
- which is read, “E derives –E”. The production E → (E) can be applied to replace any instance of E in any string of grammar symbols by (E), e.g., E * E => (E) * E => E * E => E * (E). We can take a single E and repeatedly apply productions in any order to get a sequence of replacements.
- For example, E => -E => (E) => (id)
- **Types of Derivations**
  - Leftmost Derivations
  - Rightmost Derivations

CFG – LEFTMOST DERIVATIONS

- In leftmost derivations, the leftmost nonterminal in each sentential is always chosen.
- If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in is α replaced,

  we write $\alpha \underset{lm}{\Rightarrow} \beta$.

$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$

- $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E + E) \underset{lm}{\Rightarrow} -(id + E) \underset{lm}{\Rightarrow} -(id + id)$

CFG – LEFTMOST DERIVATIONS

- In rightmost derivations, the rightmost nonterminal is always chosen;

  we write $\alpha \underset{rm}{\Longrightarrow} \beta$.

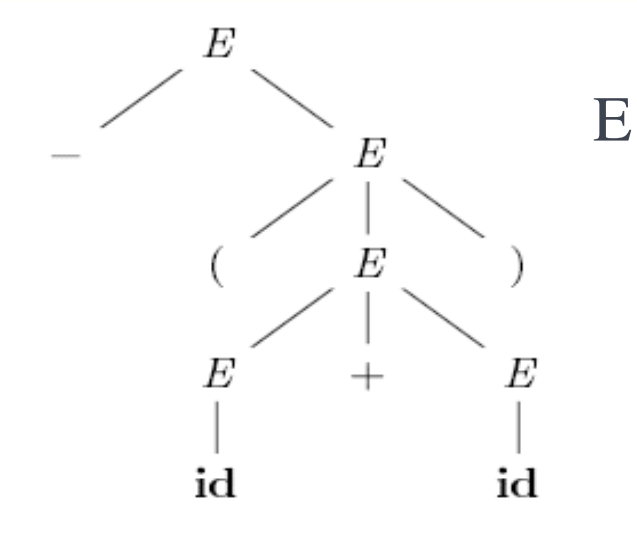
  $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$

- $E \underset{rm}{\Longrightarrow} -E \underset{rm}{\Longrightarrow} -(E) \underset{rm}{\Longrightarrow} -(E + E) \underset{rm}{\Longrightarrow} -(E + id) \underset{rm}{\Longrightarrow} -(id + id)$

CFG – PARSE TREES AND DERIVATIONS

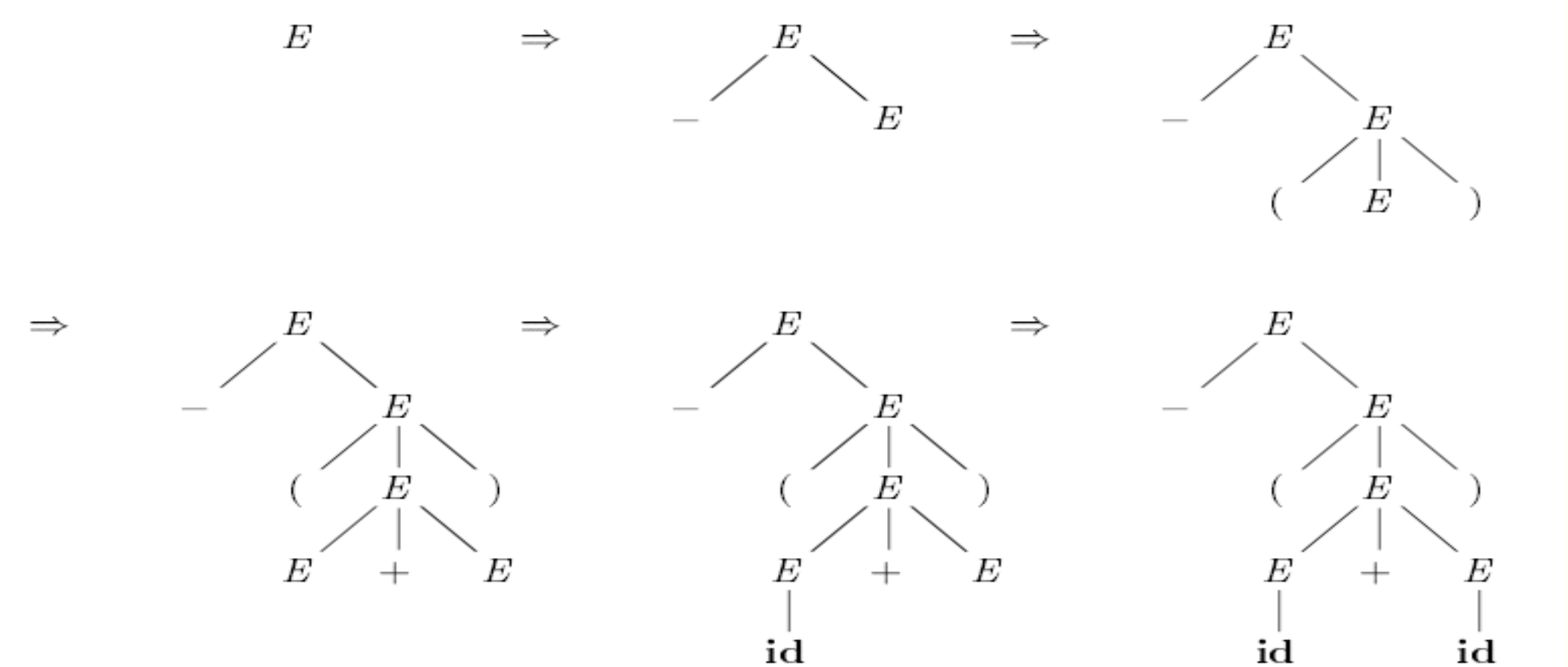
- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals.
- Each interior node of a parse tree represents the application of a production.
- The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.
- -(id + id)

E → E + E | E * E | -E | (E) | id



- Parse tree for -(id + id)

CFG – PARSE TREES AND DERIVATIONS



Sequence of parse trees for derivation

CFG – AMBIGUITY

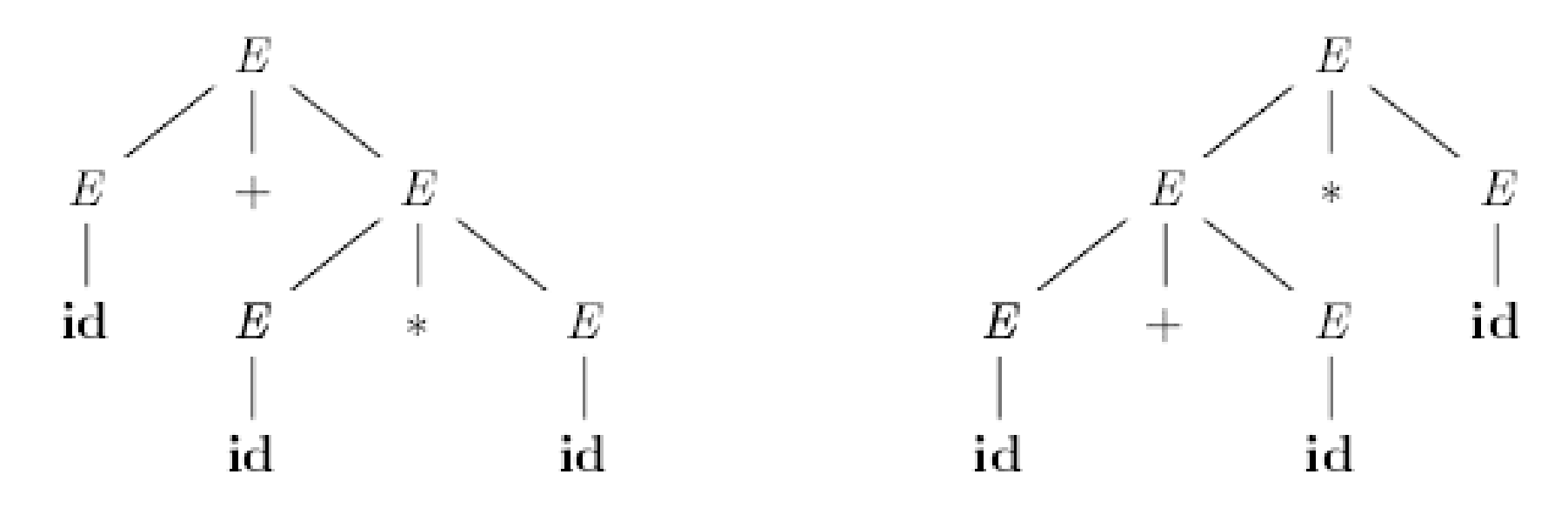
- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.
- Example: E → E + E | E * E | (E) | id
- permits two distinct leftmost derivations for the sentence id + id * id:

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

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

CFG – AMBIGUITY

- Example: E → E + E | E * E | (E) | id



Two parse trees for id + id * id

CFG – Verifying the Language Generated by a Grammar

- A proof that a grammar G generates a language L has two parts: show that every string generated by G is in L, and conversely that every string in L can indeed be generated by G.

CFG – Context-Free Grammars Versus Regular Expressions

- A proof that a grammar G generates a language L has two parts: show that every string generated by G is in L, and conversely that every string in L can indeed be generated by G.
- Example: $S \rightarrow (S) S \mid \varepsilon$
- $S \underset{lm}{\Rightarrow} (S)\, S \overset{*}{\underset{lm}{\Rightarrow}} ((x)S \overset{*}{\underset{lm}{\Rightarrow}} (x)y$
- $S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)\, y$
- Proving that w = (x)y is also derivable from S.

CFG – Context-Free Grammars Versus Regular Expressions

- Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.
- For example, the regular expression (a|b)*abb and the grammar
- $A_0 \rightarrow aA_0 \mid bA_0 \mid aA1$
- $A_1 \rightarrow bA_2$
- $A_2 \rightarrow bA_3$
- $A_3 \rightarrow \varepsilon$
- describe the same language, the set of strings of a's and b's ending in abb.

DERIVATION TREES

- Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG.
- It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules.
- The derivation tree is also called a parse tree.
- Parse tree follows the precedence of operators.
- The deepest sub-tree traversed first.
- So, the operator in the parent node has less precedence over the operator in the sub-tree.

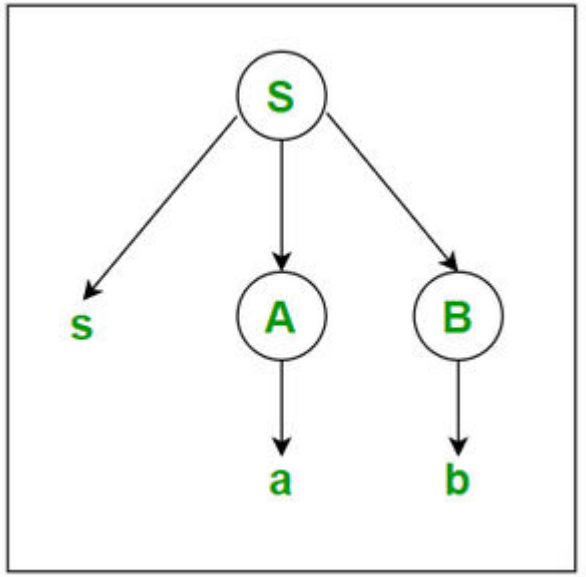
DERIVATION TREES

- **Rules to Draw a Parse Tree :**
  1. The root node is always a node indicating start symbols.
  2. The derivation is read from left to right.
  3. The leaf node is always terminal nodes.
  4. The interior nodes are always the non-terminal nodes.
- Example-1:
- Let us take an example of a Grammar (Production Rules).

  S -> sAB

  A -> a

  B -> b
- The input string is “sab”, then the Parse Tree is :



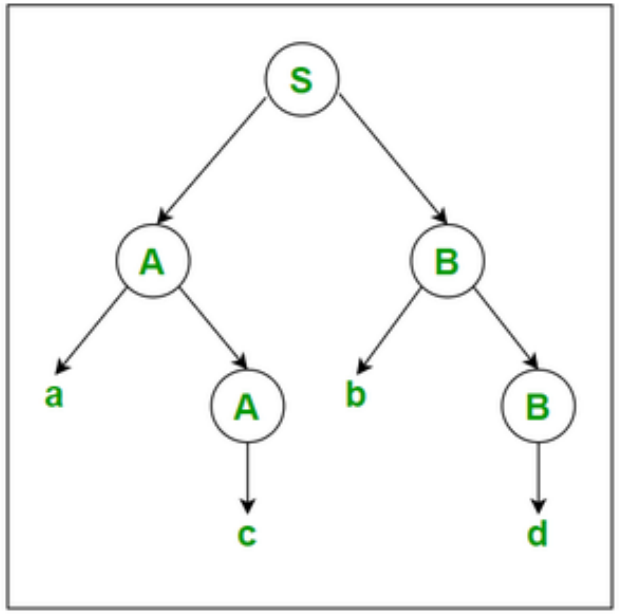
DERIVATION TREES

- Example-2:
- Let us take another example of a Grammar (Production Rules).

  S -> AB

  A -> c/aA

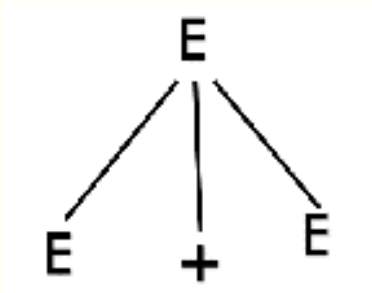
  B -> d/bB
- The input string is “acbd”, then the Parse Tree is :



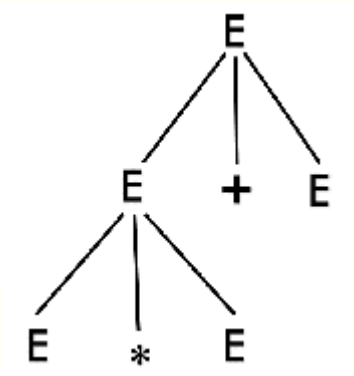
DERIVATION TREES

- **Production rules:**
- E = E + E
- E = E * E
- E = a | b | c
- Input: a * b + c

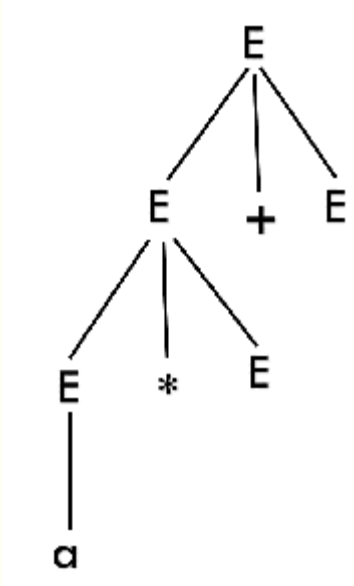
Step 1:



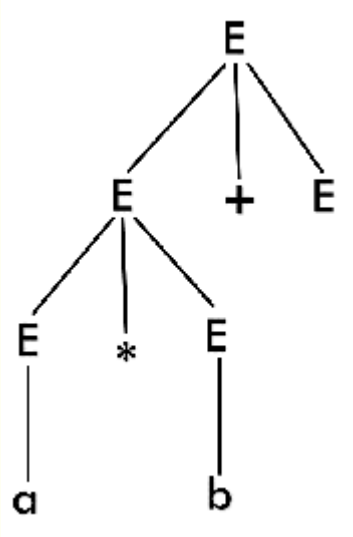
Step 2:



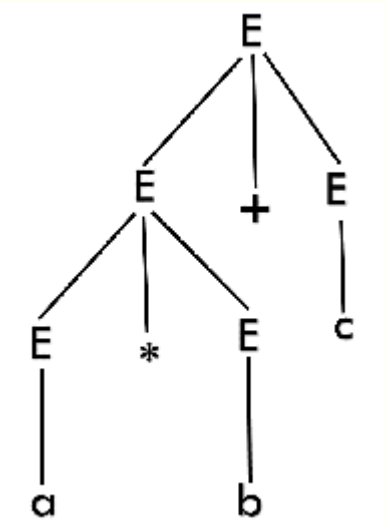
Step 3:



Step 4:



Step 5:



DERIVATION TREES

- Example 1

  S → bSb | a | b

  input→ bbabb

- Example 2

  S → aB | bA

  A → a | aS | bAA

  B → b | bS | aBB

  Input → aabbabba

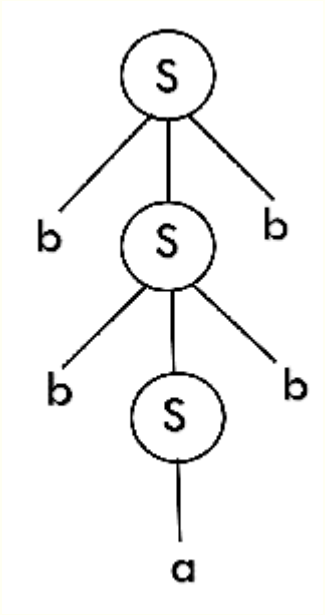
- Example 3

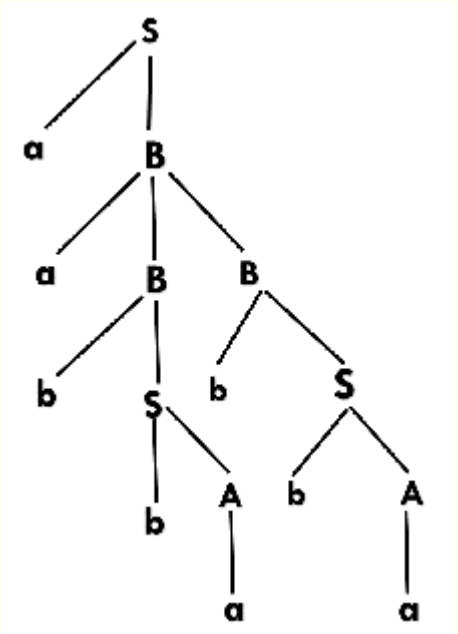
  S → AB | ε

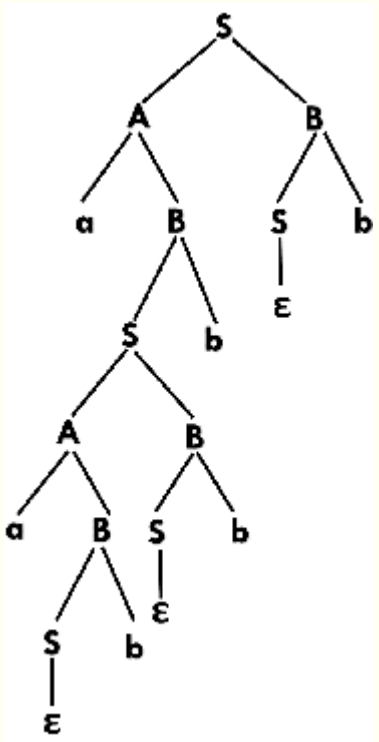
  A → aB

  B → Sb

  Input → aabbbb







AMBIGUITY IN GRAMMARS AND LANGUAGES

- A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.
- If the grammar is not ambiguous, then it is called unambiguous.
- If the grammar has ambiguity, then it is not good for compiler construction.
- No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

$E \rightarrow I$

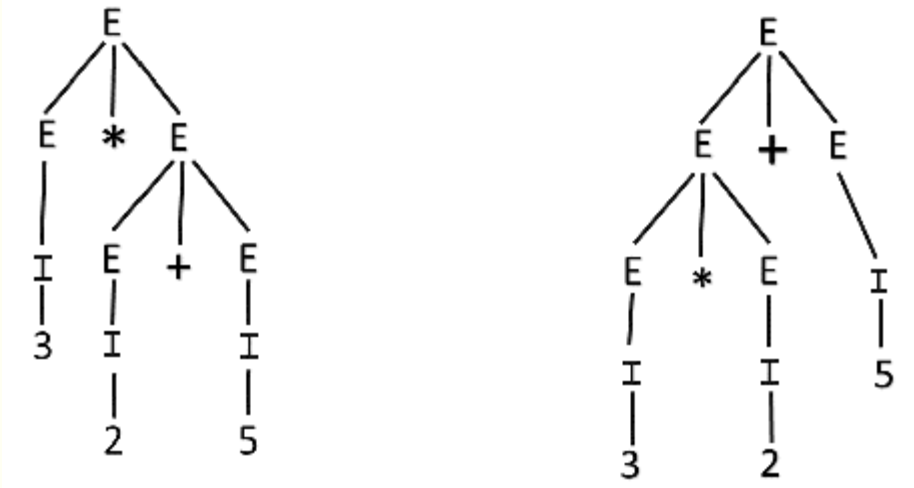
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid ... \mid 9$

string "3 * 2 + 5“



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

AMBIGUITY IN GRAMMARS AND LANGUAGES

- E → E + E
- E → E - E
- E → id
- String "id + id - id"
- First Leftmost derivation

  E → E + E
  → id + E
  → id + E - E
  → id + id - E
  → id + id- id

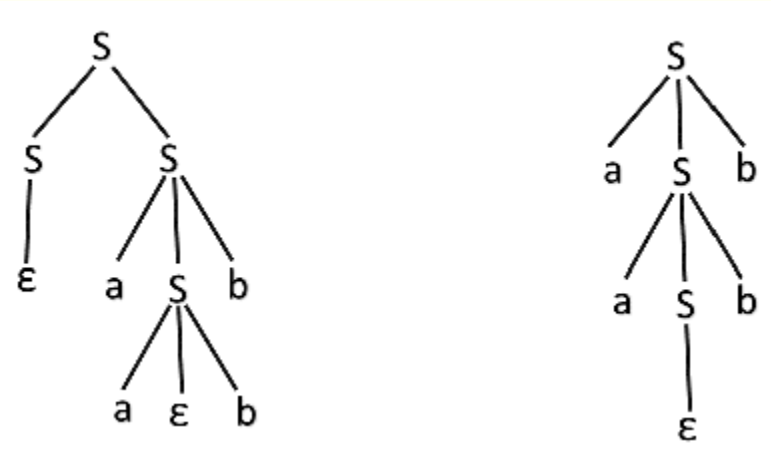
  Second Leftmost derivation

  E → E - E
  → E + E - E
  → id + E - E
  → id + id - E
  → id + id - id

- Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

AMBIGUITY IN GRAMMARS AND LANGUAGES

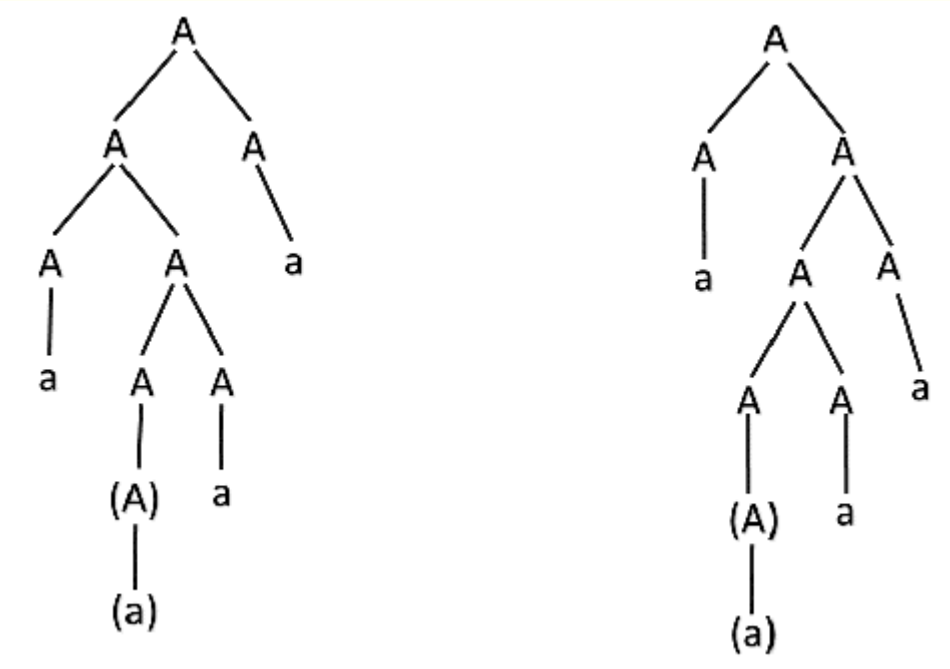
- S → aSb | SS
- S → ε
- tring "aabb“



- Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

AMBIGUITY IN GRAMMARS AND LANGUAGES

- $A \rightarrow AA$
- $A \rightarrow (A)$
- $A \rightarrow a$
- string "a(a)aa"



- Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

UNAMBIGUITY IN GRAMMARS AND LANGUAGES

- To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

   $$X \rightarrow Xa$$

2. If the right associative operates(^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

   $$X \rightarrow aX$$

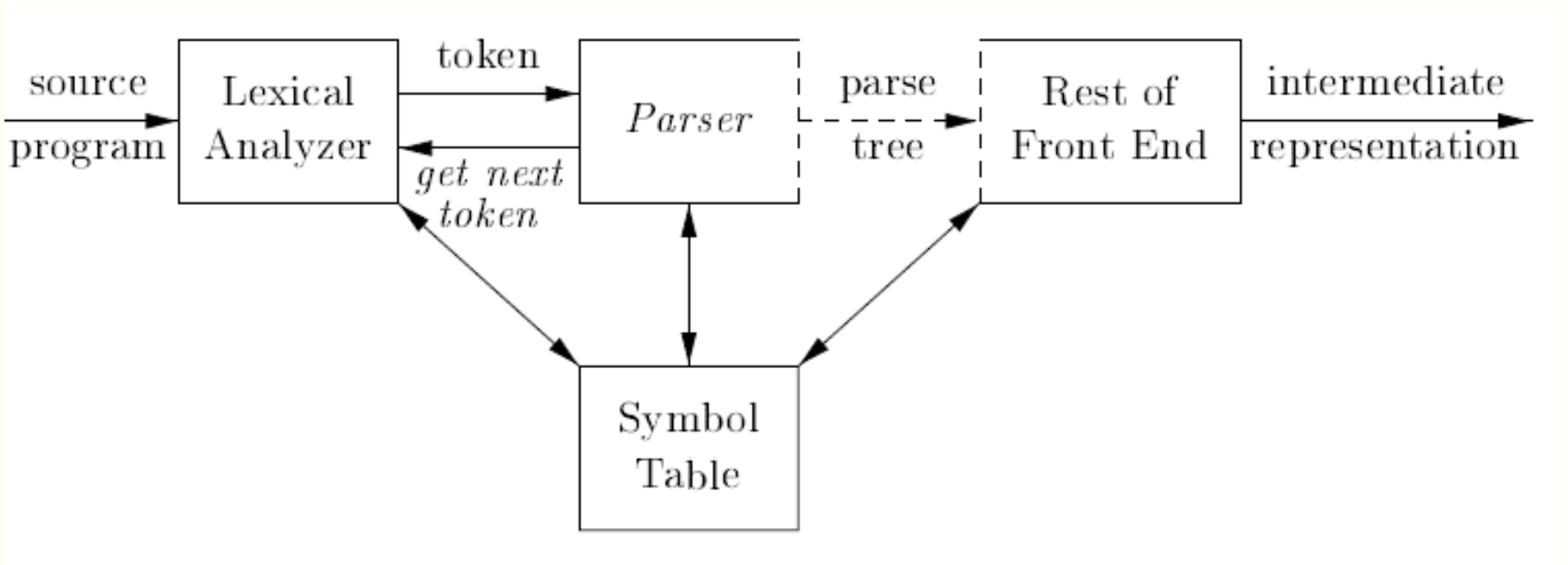
UNAMBIGUITY IN GRAMMARS AND LANGUAGES

- S → AB | aaB
- A → a | Aa
- B → b derive the string "aab"
- Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.
- S → ABA
- A → aA | ε
- B → bB | ε

NEED AND ROLE OF THE PARSER

- The string of token names can be generated by the grammar for the source language.
- We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.
- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see.
- Thus, the parser and the rest of the front end could well be implemented by a single module

NEED AND ROLE OF THE PARSER



Position of parser in compiler model

REFERENCE

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Second Edition, Pearson Education Limited, 2014.
2. Randy Allen, Ken Kennedy, “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”, Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint, 2003.
4. Keith D Cooper and Linda Torczon, “Engineering a Compiler”, Morgan Kaufmann Publishers, Elsevier Science, 2004.
5. V. Raghavan, “Principles of Compiler Design”, Tata McGraw Hill Education Publishers, 2010.
6. Allen I. Holub, “Compiler Design in C”, Prentice-Hall Software Series, 1993.


Thank you