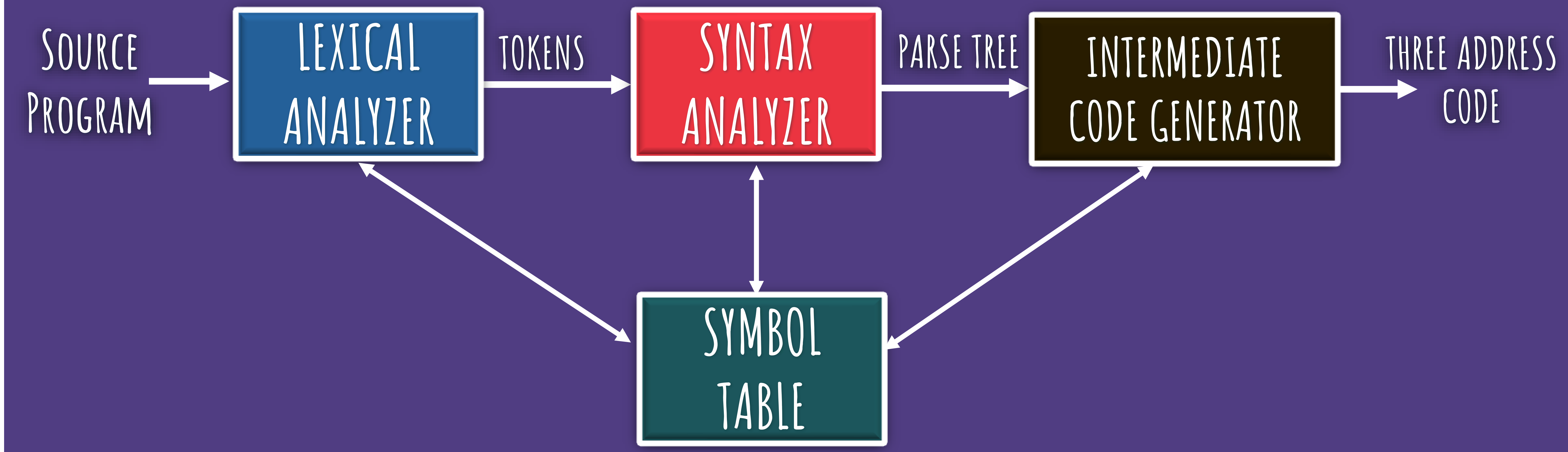
COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

SYNTAX ANALYSIS

- Syntax analysis or parsing is the second phase of a compiler.
- A lexical analyzer can identify tokens with the help of regular expressions and pattern rules.
- A lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.
- Regular expressions cannot check balancing tokens, such as parenthesis.
- Syntax analysis phase uses **Context-Free Grammar (CFG)**, which is recognized by push-down automata.

MODEL OF A COMPILER FRONT END



REVIEW OF CFG

- A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol and productions.
  - **Terminals** are the basic symbols from which strings are formed.
  - The term "token name" is a synonym for "terminal.

REVIEW OF CFG

- A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol and productions.
  - **Non-terminals** are syntactic variables that denote sets of strings.
  - Non-terminals define sets of strings that help define the language generated by the grammar.
  - They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.

REVIEW OF CFG

- In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar.
- Conventionally, the productions for the start symbol are listed first.

REVIEW OF CFG

- The **productions** of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of
  - A set of **production rules** which are the rules for replacing nonterminal symbols.
  - Production rules have the following form: variable→ string of variables and terminals.

REVIEW OF CFG

- The grammar with the following productions defines simple arithmetic expression

$$expr \rightarrow expr + term$$
$$expr \rightarrow expr - term$$
$$expr \rightarrow term$$
$$term \rightarrow term * factor$$
$$term \rightarrow term/factor$$
$$term \rightarrow factor$$
$$factor \rightarrow (expr)$$
$$factor \rightarrow \mathbf{id}$$

In this grammar, the **terminal symbols** are : id + - * / ( )
The **nonterminal symbols** are : expr, term, factor
Start symbol : expr

REVIEW OF 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.

REVIEW OF 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.

REVIEW OF CFG

Notational Conventions

- Uppercase letters, such as X, Y, Z, represent grammar symbols; that is, either non-terminals or terminals.
- Lowercase letters late in the alphabet , chiefly u, v, ... ,z, represent (possibly empty) strings of terminals.
- Lowercase Greek letters α, β, γ for example, represent (possibly empty) strings of grammar symbols.

REVIEW OF CFG

Notational Conventions

- 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 $A \rightarrow \alpha 1 \mid \alpha 2 \mid \ldots\ldots \mid \alpha k$
- $\alpha 1$, $\alpha 2$,.., αk are called the alternatives for A.
- Unless stated otherwise, the head of the first production is the start symbol.

REVIEW OF CFG

Notational Conventions

- Using these conventions, the grammar for arithmetic expression can be rewritten as

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

DERIVATION

- 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.

DERIVATION

- E.g. consider the following grammar , with a single non-terminal E

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

The production E → - E signifies that if E denotes an expression, then – E must also denote an expression.

The replacement of a single E by – E will be described by writing E => -E which is read, "E derives - E."

DERIVATION

- 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 or E * E ⇒ E * (E)**
- We can take a single E and repeatedly apply productions in any order to get a sequence of replacements.

  e.g., **E ⇒ - E ⇒ - (E) ⇒ - (id)**
- We call such a sequence of replacements a derivation of - (id) from E.
- This derivation provides a proof that the string - (id) is one particular instance of an expression.

Leftmost Derivation

- A leftmost derivation is obtained by applying production to the leftmost variable in each step.

$$s \rightarrow AB$$
$$A \rightarrow aaA \mid \varepsilon$$
$$B \rightarrow Bb \mid \varepsilon$$

- Leftmost Derivation

$$\begin{aligned} s &\Rightarrow AB \\ &\Rightarrow aaAB \\ &\Rightarrow aaB \\ &\Rightarrow aaBb \\ &\Rightarrow aab \end{aligned}$$

Rightmost Derivation

- A leftmost derivation is obtained by applying production to the rightmost variable in each step.

$$s \rightarrow AB$$
$$A \rightarrow aaA \mid \varepsilon$$
$$B \rightarrow Bb \mid \varepsilon$$

- Rightmost Derivation

$$\begin{aligned} s &\Rightarrow AB \\ &\Rightarrow ABb \\ &\Rightarrow Ab \\ &\Rightarrow aaAb \\ &\Rightarrow aab \end{aligned}$$

LEFTMOST & RIGHTMOST DERIVATION

- Let any set of production rules in a CFG be

  $\mathbf{X \rightarrow X{+}X \mid X{*}X \mid X \mid a}$ over an alphabet {a}

- The leftmost derivation for the string "**a+a*a**" may be

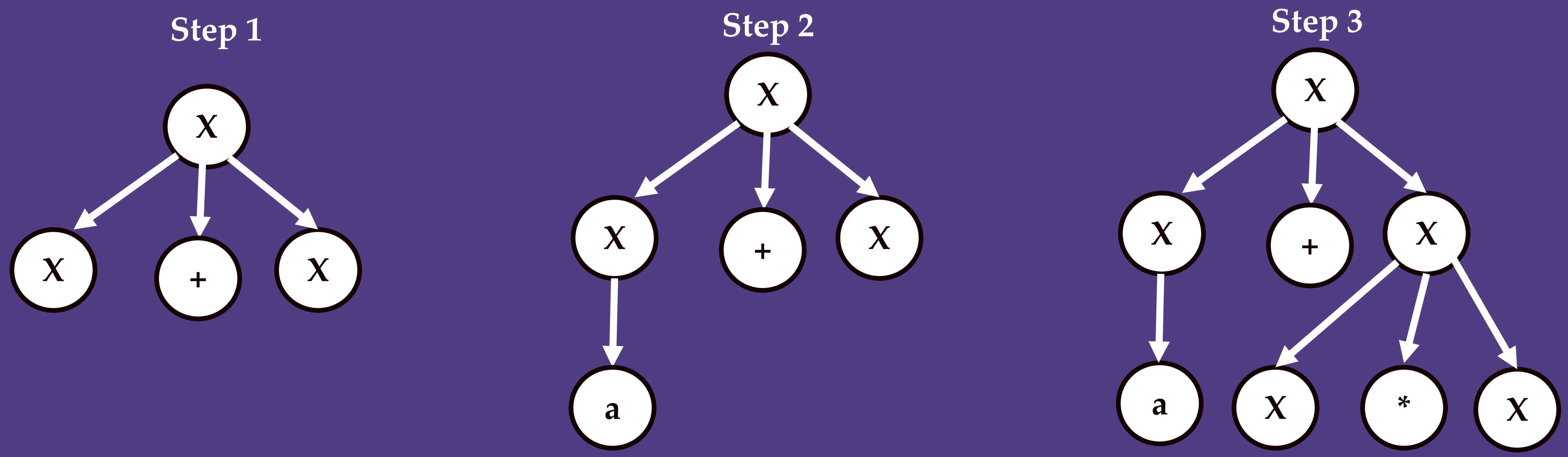
  $X \Rightarrow X+X \Rightarrow a+X \Rightarrow a + X*X \Rightarrow a+a*X \Rightarrow a+a*a$

- The rightmost derivation for the string may be

  $X \Rightarrow X*X \Rightarrow X*a \Rightarrow X+X*a \Rightarrow X+a*a \Rightarrow a+a*a$

LEFTMOST DERIVATION

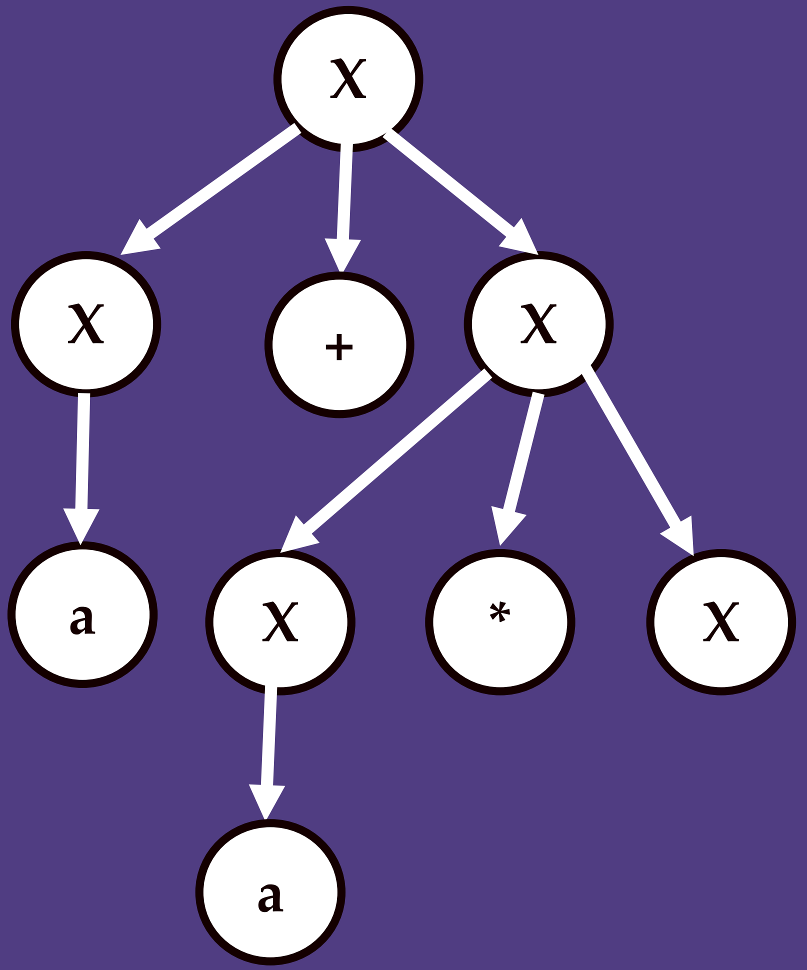
- Step-wise derivation of the string is



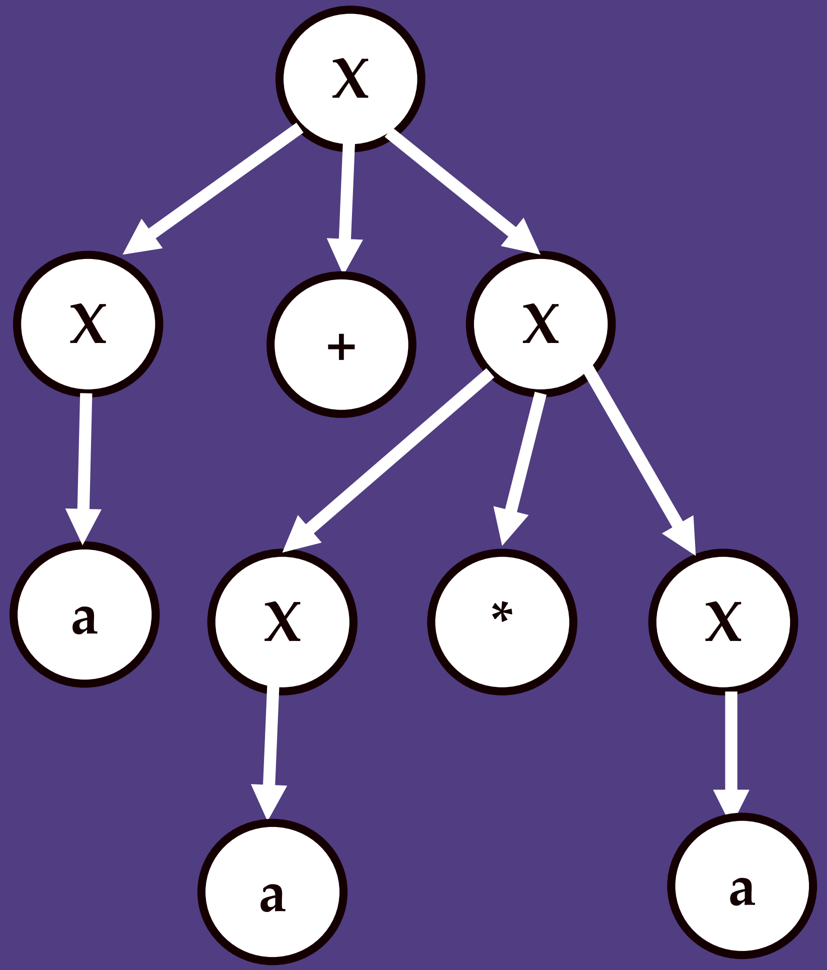
Divys-Compiler Design PPT

LEFTMOST DERIVATION

Step 4



Step 5

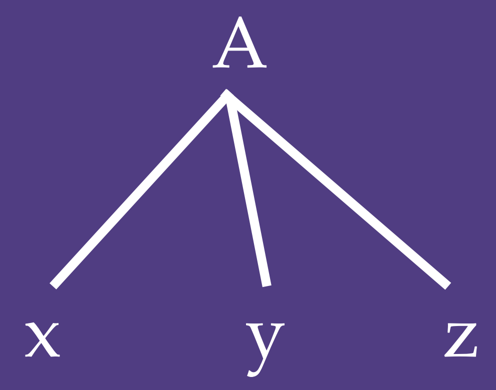


PARSE TREES

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Simply it is the graphical representation of derivations.
- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree are labeled by non-terminals or terminals.
- Each interior node is labeled by some non terminals.

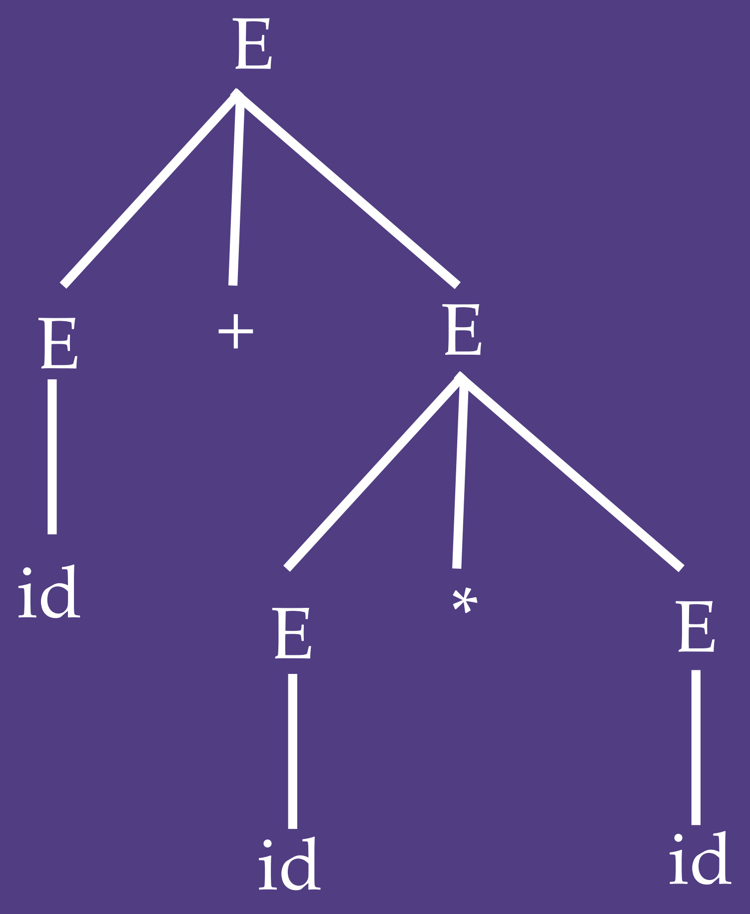
PARSE TREES

- If **A→xyz** is a production, then the parse tree will have **A** as interior node whose children are **x, y** and **z** from its left to right.



PARSE TREES

- Construct the parse tree for E→E+E | E*E | E | id



YIELD OF A PARSE TREE

- The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right, they constitute a sentential form, called the yield or frontier of the tree.
- The string **id + id * id**, is the yield of the parse tree.

AMBIGUITY

- An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.
- For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.
- In other cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that "throw away" undesirable parse trees, leaving only one tree for each sentence.

AMBIGUITY

- Consider the input string id+id*id

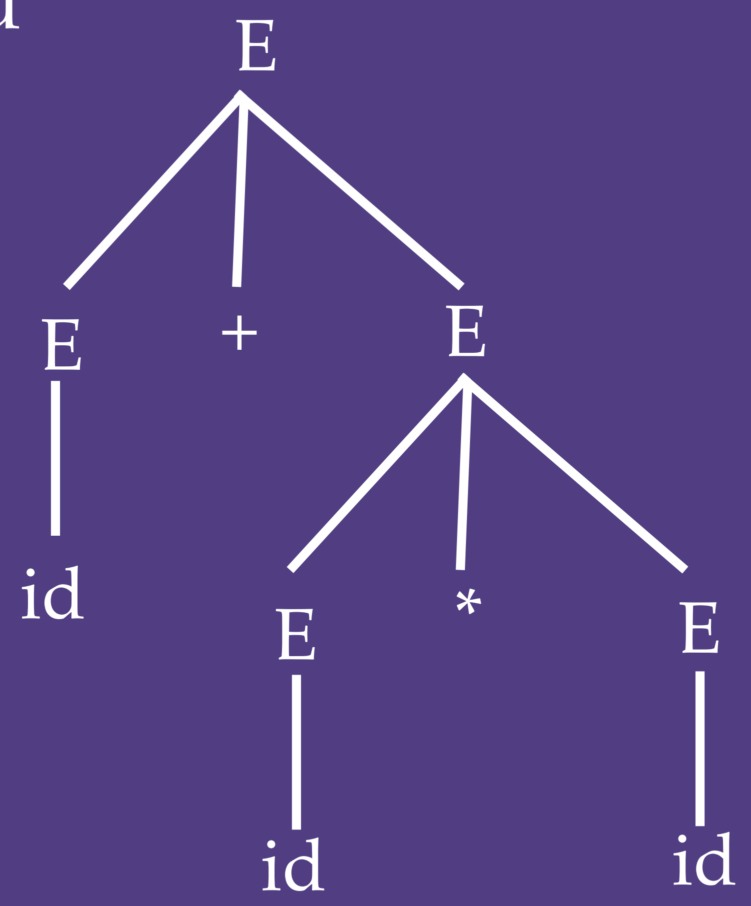
E⇒E+E

⇒id+E

⇒id+E*E

⇒id+id*E

⇒id+id*id



AMBIGUITY

- Consider the input string id+id*id

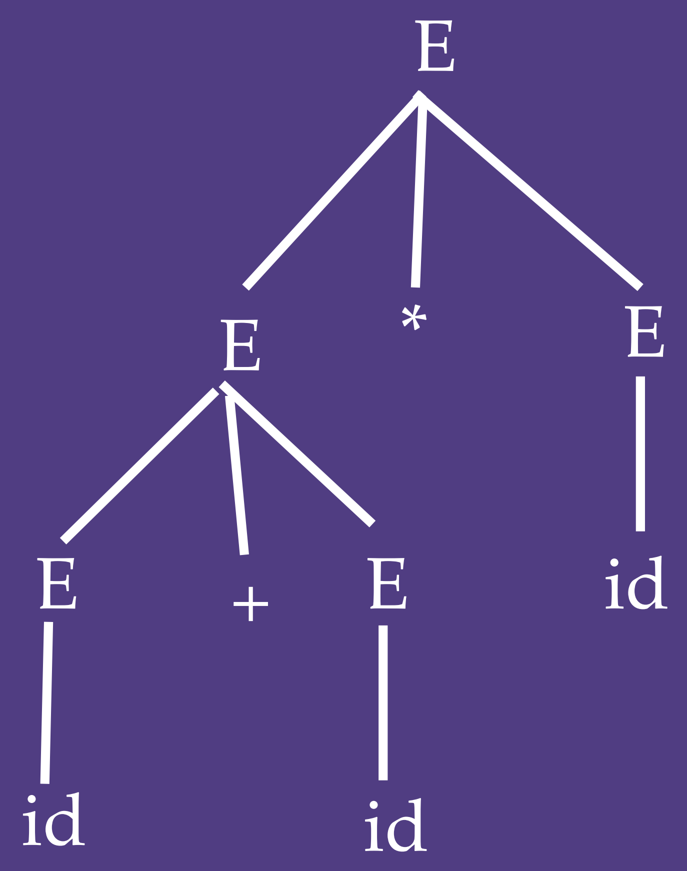
$E \Rightarrow E*E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$



WRITING A GRAMMAR

- Grammars are describing most, but not all, of the syntax of the programming languages.
- E.g. Identifiers need to be declared before they are used cannot be described by a context-free grammar.
- The sequences of tokens accepted by a parser form a superset of the programming language.
- Subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

WRITING A GRAMMAR

Lexical Versus Syntactic Analysis

- Everything that can be described by regular expression can also be described by a grammar.
- Then, why regular expression is used to describe the lexical syntax of a language?

WRITING A GRAMMAR

Lexical Versus Syntactic Analysis

1. Separating the syntactic structure of a language into lexical and non-lexical parts provide a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.

WRITING A GRAMMAR

Lexical Versus Syntactic Analysis

3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More difficult lexical analyzers can be constructed automatically from regular expressions than arbitrary grammars.

WRITING A GRAMMAR

Lexical Versus Syntactic Analysis

- Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords and white space.
- Grammars are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's and so on. These cannot be described by regular expressions.

WRITING A GRAMMAR

Eliminating Ambiguity

- Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.
- Consider the dangling else grammar

$$
\begin{aligned}
stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\
|\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\
|\ & \textbf{other}
\end{aligned}
$$

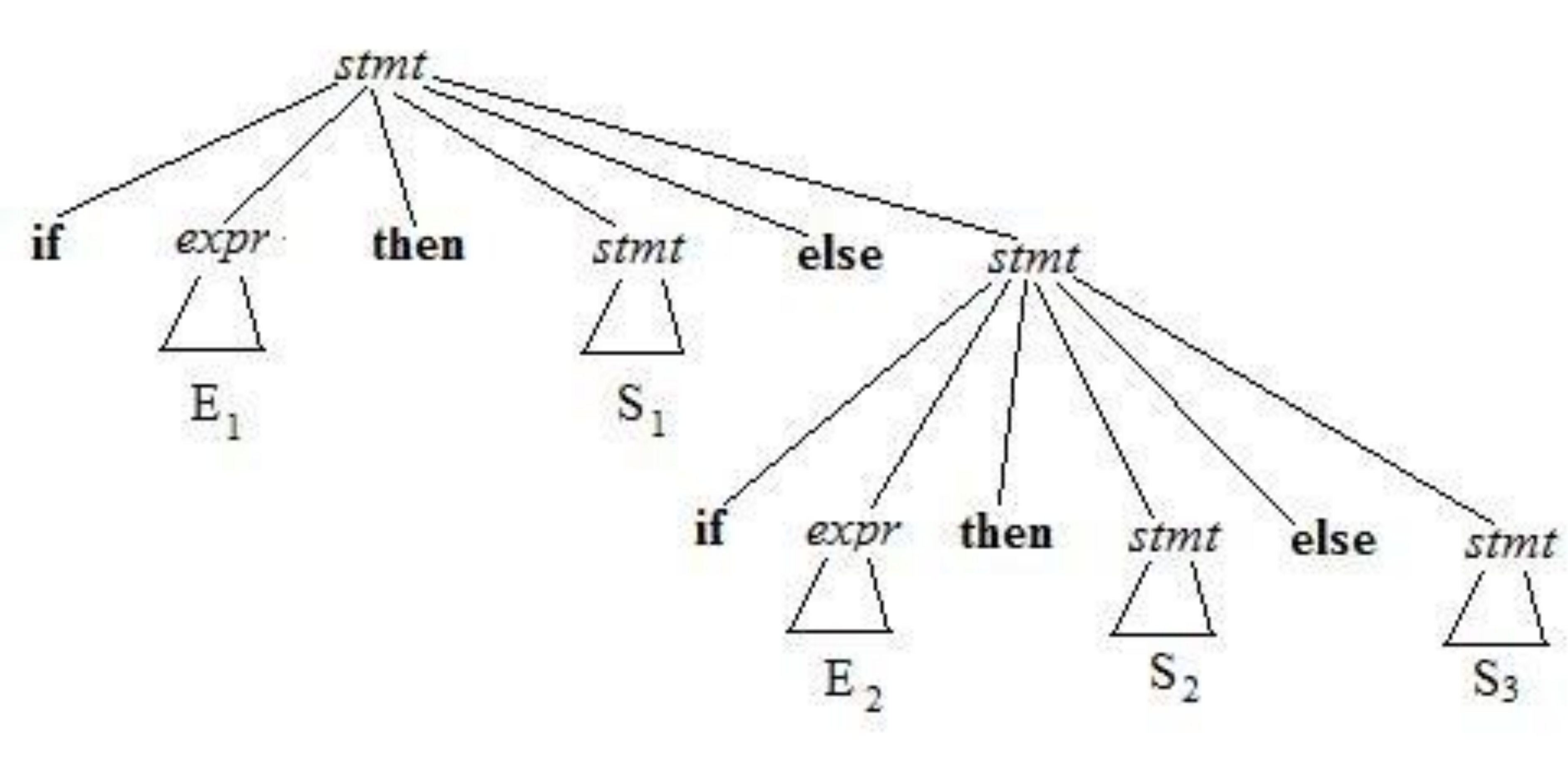
Here, **other** stands for any other statement

Eliminating Ambiguity

The compound conditional statement,

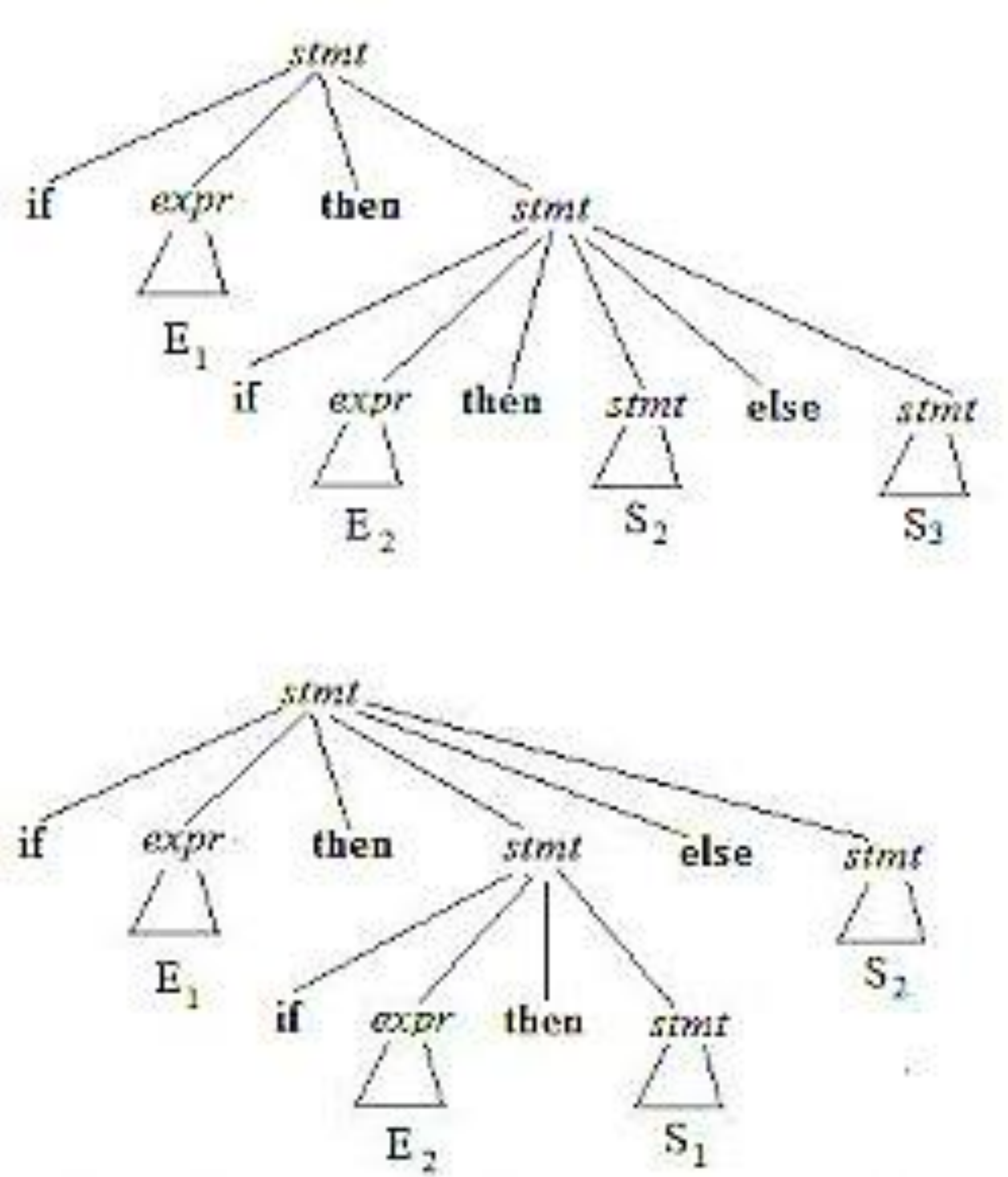
if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

has the parse tree



Eliminating Ambiguity

The grammar is ambiguous since the string
if E_1 **then if** E_2 **then** S_1 **else** S_2
has two parse trees



WRITING A GRAMMAR

Eliminating Ambiguity

- In all programming languages with conditional statements of this form, the first parse tree is preferred.
- The general rule is, “Match each else with the closest unmatched then”.

WRITING A GRAMMAR

Eliminating Ambiguity

- Unambiguous grammar for if-then-else statements,

$$
\begin{aligned}
stmt &\rightarrow matched\_stmt \\
&\mid\ open\_stmt \\
matched\_stmt &\rightarrow \textbf{\textit{if}}\ expr\ \textbf{\textit{then}}\ matched\_stmt\ \textbf{else}\ matched\_stmt \\
&\mid\ other \\
open\_stmt &\rightarrow \textbf{if}\ \mathrm{expr}\ \textbf{then}\ \mathrm{stmt} \\
&\mid\ \textbf{if}\ \mathrm{expr}\ \textbf{then}\ matched\_stmt\ \ \textbf{else}\ open\_stmt
\end{aligned}
$$

WRITING A GRAMMAR

Elimination of Left Recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{*}{\Rightarrow} A\alpha$ for some string α.
- Top-down parsing methods cannot handle left recursive grammars, so a transformation is needed to eliminate left recursion.

WRITING A GRAMMAR

Elimination of Left Recursion

- $A \rightarrow A\alpha \mid \beta$ is left recursive
- This can be made non-left recursive by

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

WRITING A GRAMMAR

Elimination of Left Recursion

Eliminate left recursion from the grammar

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

After eliminating left recursion,

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Elimination of Left Recursion

Eliminate left recursion from the grammar

$S \rightarrow ABC$
$A \rightarrow Aa \mid Ad \mid b$
$B \rightarrow Bb \mid e$
$C \rightarrow Cc \mid g$

After eliminating left recursion,
$S \rightarrow ABC$
$A \rightarrow bA'$
$A' \rightarrow aA' \mid \varepsilon \mid dA'$
$B \rightarrow eB'$
$B' \rightarrow bB' \mid \varepsilon$
$C \rightarrow gC'$
$C' \rightarrow cC' \mid \varepsilon$

WRITING A GRAMMAR

Elimination of Left Recursion

- Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions.
- First group the productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A.

WRITING A GRAMMAR

Elimination of Left Recursion

- Then replace the A-productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

- The non terminal A generates the same strings as before but it is no longer left recursive.

WRITING A GRAMMAR

Elimination of Left Recursion

- The procedure eliminates all left recursion from the A and A′ productions (provided no αi is ε), but it does not eliminate left recursion involving derivations of two or more steps.
- E.g.

$$S \rightarrow A\,a \mid b$$
$$A \rightarrow A\,c \mid S\,d \mid \epsilon$$

Elimination of Left Recursion

INPUT : Grammar G with no cycles or ε-productions.

OUTPUT: An equivalent grammar with no left recursion.

```
arrange the nonterminals in some order A1, A2, ..., An.
for ( each i from 1 to n ) {
        for ( each j from 1 to i - 1 ) {
                replace each production of the form Ai → Ajγ by the
                    productions Ai → δ1γ | δ2γ | ··· | δkγ, where
                    Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
        }
        eliminate the immediate left recursion among the Ai-productions
}
```

Elimination of Left Recursion

Eliminate left recursion from the grammar

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

Substitute S in $A \rightarrow Sd$ to obtain the following A-productions,
$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Eliminating left recursion yields the following grammar
$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

WRITING A GRAMMAR

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.

WRITING A GRAMMAR

Left Factoring

- For e.g., if we have two productions

$$\begin{aligned} stmt \rightarrow\ & \textbf{\textit{if}}\ expr\ \textbf{\textit{then}}\ stmt\ \textbf{\textit{else}}\ stmt \\ |\ \ & \textbf{\textit{if}}\ expr\ \textbf{\textit{then}}\ stmt \end{aligned}$$

- On seeing the input if, we cannot immediately tell which production to choose to expand stmt.

WRITING A GRAMMAR

Left Factoring

- In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two productions, and the input begins with a non empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
- We can defer the decision by expanding A to $\alpha A'$
- After seeing the input derived from α we can expand A' to β_1 or β_2

WRITING A GRAMMAR

Left Factoring

- Left factored the original productions become,

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Left Factoring

INPUT : Grammar G.

OUTPUT : An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

WRITING A GRAMMAR

Left Factoring

- Apply left factoring to the "dangling-else" problem

$$
\begin{aligned}
S &\rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a \\
E &\rightarrow b
\end{aligned}
$$

$$
\begin{aligned}
S &\rightarrow i\,E\,t\,S\,S' \mid a \\
S' &\rightarrow e\,S \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$