

# **CS 2201: Context Free Grammar**

# Context-Free Grammars (CFG)

- Inherently recursive structures of a programming language are defined by a CFG
- In a CFG, we have:
  - A finite set of terminals (in our case, this will be the set of tokens)
  - A finite set of non-terminals (syntactic-variables)
  - A finite set of productions rules in the following form
$$A \rightarrow \alpha$$
where  $A$  is a non-terminal and  $\alpha$  is a string of terminals and non-terminals (including the empty string);  $|A| \leq |\alpha|$
  - A start symbol: one of the non-terminal symbols
- Example:
$$E \rightarrow E + E \quad | \quad E - E \quad | \quad E * E \quad | \quad E / E \quad | \quad - E$$
$$E \rightarrow ( E )$$
$$E \rightarrow \text{id}$$

# Derivations

$E \Rightarrow E+E$

- $E+E$  derives from  $E$ 
  - we can replace  $E$  by  $E+E$

$E \Rightarrow E+E \Rightarrow \text{id} + E \Rightarrow \text{id} + \text{id}$

- A sequence of replacements of non-terminal symbols is called a **derivation** of  $\text{id} + \text{id}$  from  $E$
- In general a derivation step is  
 $\alpha A \beta \Rightarrow \alpha \gamma \beta$  if there is a production rule  $A \rightarrow \gamma$  in our grammar  
where  $\alpha$  and  $\beta$  are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$  ( $\alpha_n$  derives from  $\alpha_1$  or  $\alpha_1$  derives  $\alpha_n$ )

$\Rightarrow$  : derives in one step

$\Rightarrow$  : derives in zero or more steps

$\Rightarrow$  : derives in one or more steps

\*

+

# CFG - Terminology

- $L(G)$  is *the language of G* (the language generated by G) which is a set of sentences
- *A sentence of  $L(G)$*  is a string of terminal symbols of G
- If S is the start symbol of G then  
 $\omega$  is a sentence of  $L(G)$  iff  $S \xRightarrow{+} \omega$  where  $\omega$  is a string of terminals of G
- If G is a context-free grammar,  $L(G)$  is a *context-free language*
- Two grammars are *equivalent* if they produce the same language

\*  
 $S \Rightarrow \alpha$       - If  $\alpha$  contains non-terminals, it is called as a *sentential form* of G  
                         - If  $\alpha$  does not contain non-terminals, it is called as a *sentence* of G

# Derivation: Example

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

OR

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

- At each derivation step, we can choose any of the non-terminals in the sentential form of G for the replacement
- **left-most derivation:** always chooses the left-most non-terminal in each derivation step
- **right-most derivation:** always chooses the right-most non-terminal in each derivation step

# Left-Most and Right-Most Derivations

## Left-Most Derivation

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

## Right-Most Derivation

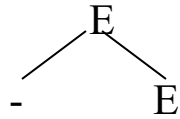
$$E \underset{\text{rm}}{\Rightarrow} -E \underset{\text{rm}}{\Rightarrow} -(E) \underset{\text{rm}}{\Rightarrow} -(E+E) \underset{\text{rm}}{\Rightarrow} -(E+id) \underset{\text{rm}}{\Rightarrow} -(id+id)$$

- **top-down parsers**: finds the left-most derivation of the given source program
- **bottom-up parsers**: finds the right-most derivation of the given source program in the reverse order

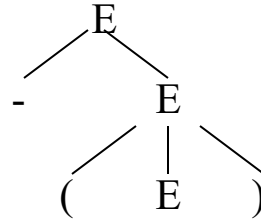
# Parse Tree

- Intermediate nodes: Inner nodes of a parse tree
- Leaves: Terminal symbols
- A parse tree can be seen as a graphical representation of a derivation

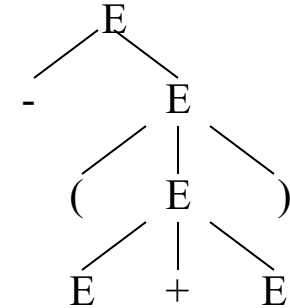
$E \Rightarrow -E$



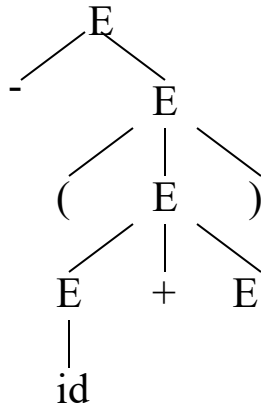
$\Rightarrow -(E)$



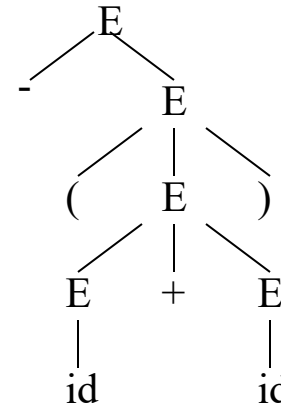
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



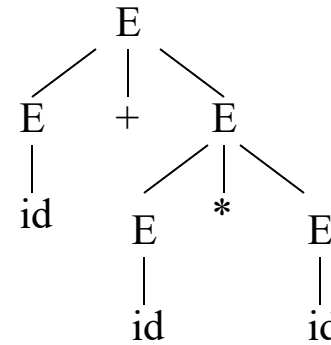
$\Rightarrow -(id+id)$



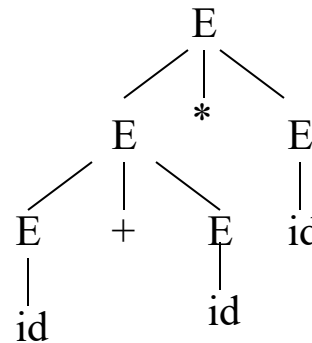
# Ambiguity

- A grammar that produces more than one parse tree for a sentence is called as an *ambiguous* grammar

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$   
 $\Rightarrow id+id*E \Rightarrow id+id*id$



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$   
 $\Rightarrow id+id*E \Rightarrow id+id*id$



# Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous
- Unambiguous grammar
  - ➔ unique selection of the parse tree for a sentence
- Disambiguation
  - Necessary to eliminate the ambiguity in the grammar
  - Design unambiguous grammar
  - Choose one of the parse trees of a sentence to restrict to this choice

# Ambiguity (cont.)

stmt  $\rightarrow$  if expr then stmt |  
if expr then stmt else stmt | otherstmts

if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$

*Interpretation-1*:  $S_2$  being executed when  $E_1$  is false (thus attaching the else to the first if)

if  $E_1$  then (if  $E_2$  then  $S_1$ ) else  $S_2$

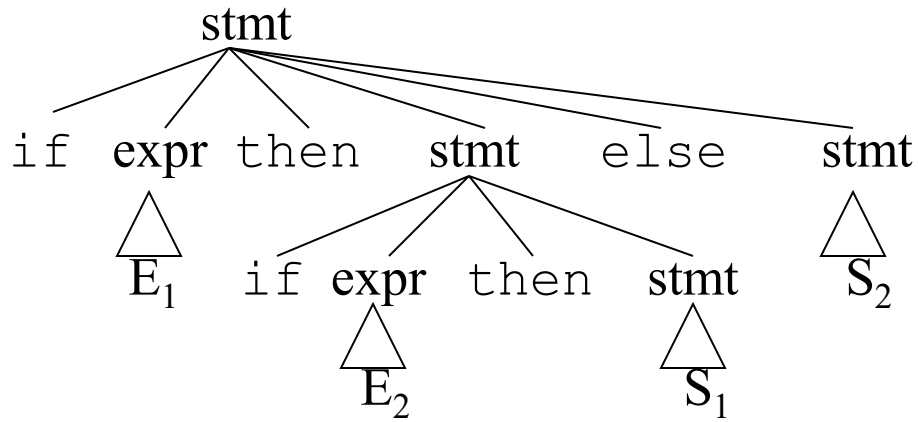
*Interpretation-11*:  $E_1$  is true and  $E_2$  is false (thus attaching the else to the second if)

if  $E_1$  then (if  $E_2$  then  $S_1$  else  $S_2$ )

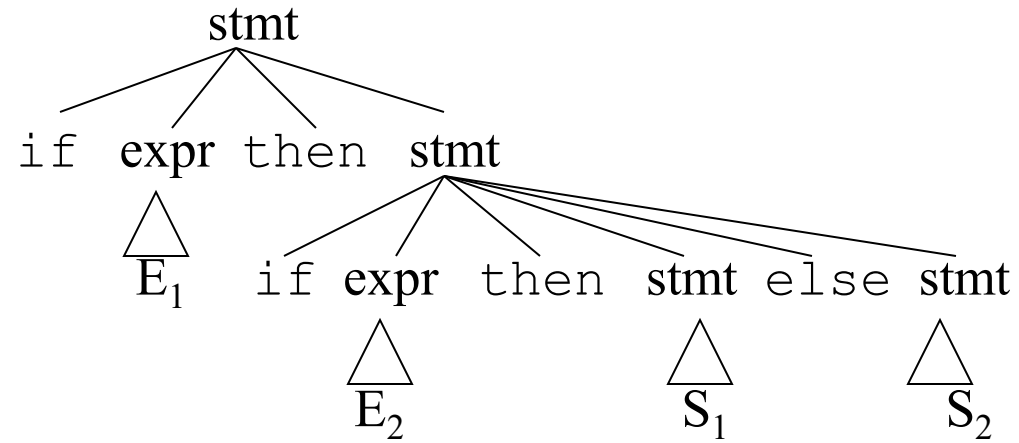
# Ambiguity (cont.)

stmt  $\rightarrow$  if expr then stmt |  
if expr then stmt else stmt | otherstmts

if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$



**1**



**2**

# Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest if)

*So, we have to disambiguate our grammar to reflect this choice*

- Unambiguous grammar:

stmt  $\rightarrow$  matchedstmt | unmatchedstmt

matchedstmt  $\rightarrow$  if expr then matchedstmt else matchedstmt |  
otherstmts

unmatchedstmt  $\rightarrow$  if expr then stmt |  
if expr then matchedstmt else unmatchedstmt

# Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity rules**

$$E \rightarrow E + E \mid E * E \mid E \wedge E \mid \text{id} \mid (E)$$

disambiguate the grammar  
 $\Downarrow$   
precedence:       $\wedge$  (right to left)  
                          $*$  (left to right)  
                          $+$  (left to right)

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow G \wedge F \mid G$$
$$G \rightarrow \text{id} \mid (E)$$

# Left Recursion

- A grammar is *left recursive* if it has a non-terminal A such that there is a derivation

$$A \xRightarrow{+} A\alpha \quad \text{for some string } \alpha$$

- Top-down parsing techniques **cannot** handle left-recursive grammars
  - Conversion of left-recursive grammar into an equivalent non-recursive grammar is essential
- Possible ways of left-recursion
  - may appear in a single step of the derivation (*immediate left-recursion*) or
  - may appear in more than one step of the derivation

# Immediate Left-Recursion

$A \rightarrow A \alpha \mid \beta$  where  $\beta$  does not start with  $A$

$\Downarrow$

eliminate immediate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$  an equivalent grammar

In general,

$A \rightarrow A \alpha_1 \mid \dots \mid A \alpha_m \mid \beta_1 \mid \dots \mid \beta_n$  where  $\beta_1 \dots \beta_n$  do not start with  $A$

$\Downarrow$

eliminate immediate left recursion

$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$  an equivalent grammar

# Immediate Left-Recursion -- Example

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow \text{id} \mid (E)$$



eliminate immediate left recursion

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow \text{id} \mid (E)$$

# Left-Recursion -- Problem

- A grammar cannot be *immediately left-recursive*, but it still can be *left-recursive*
- Just elimination of the immediate left-recursion does not guarantee a grammar which is not left-recursive

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Sc \mid d$$

This grammar is not immediately left-recursive,  
but it is still left-recursive

$$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca \quad \text{or}$$

$$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac \text{ causes to a left-recursion}$$

- Solution: *eliminate all left-recursions from the grammar*

# Eliminate Left-Recursion – Algorithm

- Arrange non-terminals in some order:  $A_1 \dots A_n$
- **for**  $i$  **from** 1 **to**  $n$  **do** {
  - **for**  $j$  **from** 1 **to**  $i-1$  **do** {
    - replace each production
$$A_i \rightarrow A_j \gamma$$
by
$$A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$$
where  $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$
- eliminate immediate left-recursions among  $A_i$  productions

# Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace  $A \rightarrow Sd$  with  $A \rightarrow Aad \mid bd$

So, we will have  $A \rightarrow Ac \mid Aad \mid bd \mid f$

- Eliminate the immediate left-recursion in A

$$\begin{aligned} A &\rightarrow bdA' \mid fA' \\ A' &\rightarrow cA' \mid adA' \mid \varepsilon \end{aligned}$$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

# Eliminate Left-Recursion – Example2

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid f \end{aligned}$$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop
- Eliminate the immediate left-recursion in A

$$\begin{aligned} A &\rightarrow SdA' \mid fA \\ A &\rightarrow cA \mid \varepsilon \end{aligned}$$

for S:

- Replace  $S \rightarrow Aa$  with  $S \rightarrow SdA'a \mid fA'a$
- So, we will have  $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

$$\begin{aligned} S &\rightarrow fA'aS' \mid bS' \\ S &\rightarrow dA'aS' \mid \varepsilon \end{aligned}$$

So, the resulting equivalent grammar which is not left-recursive is:

$$\begin{aligned} S &\rightarrow fA'aS' \mid bS' \\ S &\rightarrow dA'aS' \mid \varepsilon \\ A &\rightarrow SdA' \mid fA \\ A &\rightarrow cA \mid \varepsilon \end{aligned}$$

# Left-Factoring

- Top-down *parser without backtracking* (**predictive parser**) insists that the grammar must be *left-factored*

grammar  $\rightarrow$  a new equivalent grammar suitable for predictive parsing

$$\begin{aligned} \text{stmt} &\rightarrow \text{if expr then stmt else stmt} \quad | \\ &\text{if expr then stmt} \end{aligned}$$

After seeing `if`, we cannot decide which production rule to choose to re-write *stmt* in the derivation

# Left-Factoring (cont.)

- In general,

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$       where  $\alpha$  is non-empty and the first symbols of  $\beta_1$  and  $\beta_2$  (if they have one) are different

- Choice involved when processing  $\alpha$

$A$  to  $\alpha\beta_1$     or

$A$  to  $\alpha\beta_2$

- Re-write the grammar as follows:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$       so, we can immediately expand  $A$  to  $\alpha A'$

# Left-Factoring – Algorithm

- For each non-terminal  $A$  with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \dots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \dots \mid \beta_n$$

# Left-Factoring – Example 1

$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$

$$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cde}B \mid \underline{cdf}B$$
$$A' \rightarrow bB \mid B$$

$$A \rightarrow aA' \mid cdA''$$
$$A' \rightarrow bB \mid B$$
$$A'' \rightarrow g \mid eB \mid fB$$

# Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$



$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$



$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

# Non-Context Free Language Constructs

- Some language constructions in the programming languages are not context-free

**Example-1:**  $L1 = \{ \omega c \omega \mid \omega \text{ is in } (a|b)^* \}$

➔ declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free)

**Example-2:**  $L2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$

➔ declaring two functions (one with  $n$  parameters, the other one with  $m$  parameters), and then calling them with actual parameters