

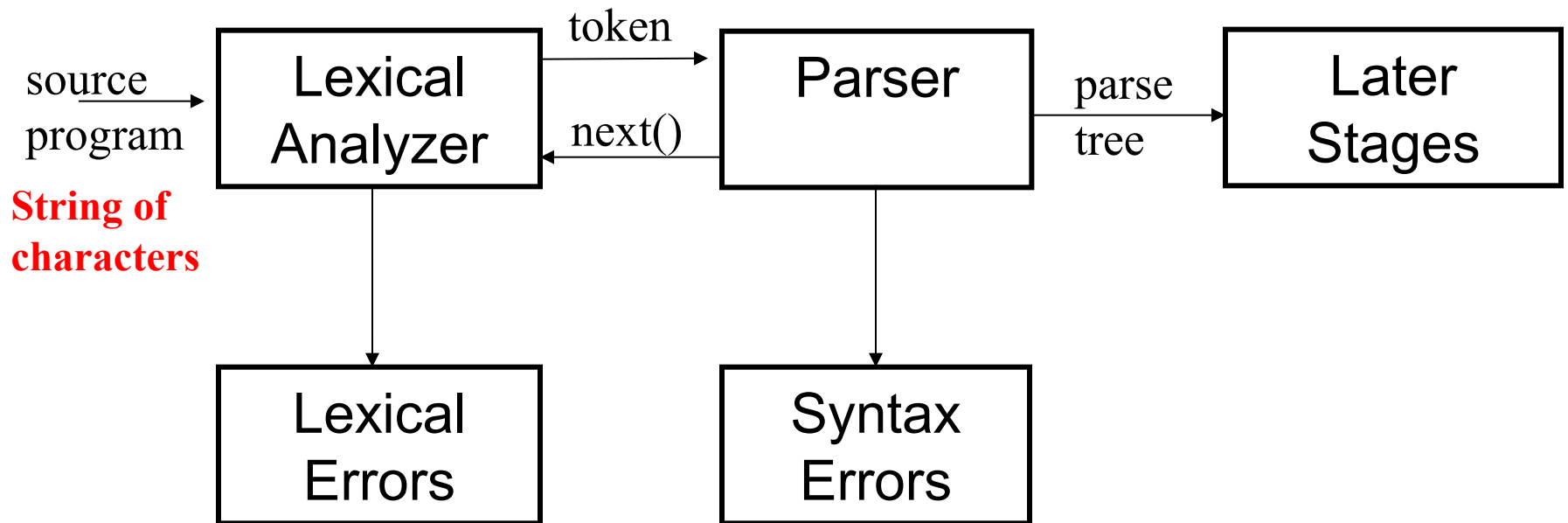
# Context-Free Grammars

CMPT 379: Compilers

Instructor: Anoop Sarkar

[anoopsarkar.github.io/compiler-class](https://anoopsarkar.github.io/compiler-class)

# Parsing



# Parsing

- Not all string of tokens are valid programs
- Parser distinguishes between valid and invalid programs
- We need
  - A language for describing valid string of tokens
  - A method for distinguishing valid from invalid programs

# Context-free Grammars (CFGs)

- Programming languages have recursive structure

- An EXP is ...

if EXP then	while EXP do
EXP	EXP
else	end
EXP	

- Context Free Grammars are natural notation for the recursive structures

# Context-free Grammars (CFGs)

- A CFG consists of
  - A set of terminals  $T$
  - A set on non-terminals  $N$
  - A start symbol  $S \in N$
  - A set of productions  $X \rightarrow Y_1 \dots Y_n$   
 $X \in N$   
 $Y_i \in N \cup T \cup \{\epsilon\}$

# Context-free Grammars (CFGs)

- $\{( ^i )^i \mid i \geq 0 \}$

Productions:

$$S \rightarrow ( S )$$

$$S \rightarrow \varepsilon$$

$$N = \{S\}$$

$$T = \{ ( , ) \}$$

# Context-free Grammars (CFGs)

1. Begin with string that has only start symbol  $S$
2. Replace any non-terminal  $X$  in the string by the right-hand side of some production  $X \rightarrow Y_1 \dots Y_n$
3. Repeat (2) until there is no non-terminals

$$\begin{array}{l} S \rightarrow ( S ) \\ S \rightarrow \varepsilon \end{array} \quad \begin{array}{c} S \\ ( S ) \\ (( S )) \\ ((( )) ) \end{array}$$

# Language of CFGs

- Let  $G$  be a context free grammar with start symbol  $S$ , and terminals  $T$ 
  - The language  $L(G)$  of  $G$  is:

$$\{\alpha_1 \dots \alpha_n \mid \forall_i \alpha_i \in T \wedge S \rightarrow^* \alpha_1 \dots \alpha_n\}$$
$$\{\epsilon, (), (()), ((())), \dots\}$$

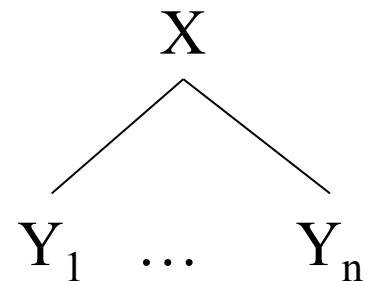


# Derivation and Parse Tree

- A derivation is a sequence of productions

$$S \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots$$

- A derivation can be drawn as a **parse tree**
  - Start symbol is the tree's root
  - For a production  $X \rightarrow Y_1 \dots Y_n$  add children  $Y_1 \dots Y_n$  to node  $X$



# Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow ( E )$
- $E \rightarrow - E$
- $E \rightarrow \mathbf{id}$

# Derivation for **id + id \* id**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow ( E )$

$E \rightarrow - E$

$E \rightarrow \text{id}$

•  $E \Rightarrow E + E$

$\Rightarrow \text{id} + E$

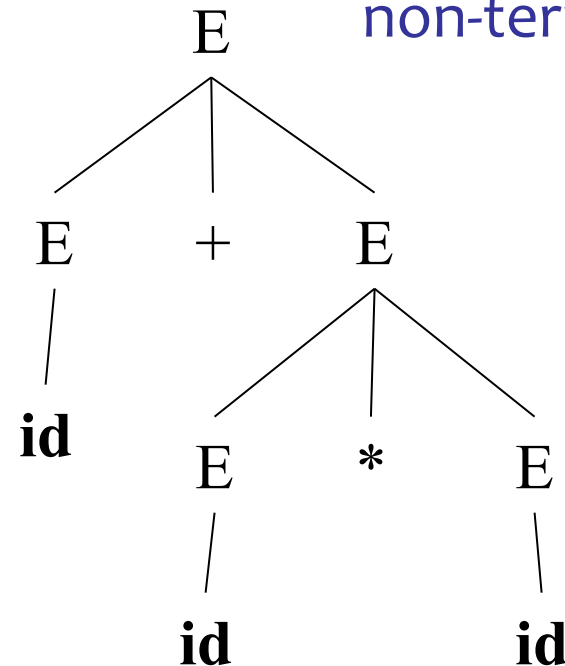
$\Rightarrow \text{id} + E * E$

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

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

Leaves nodes:  
terminals

Interior nodes:  
non-terminals



# Leftmost derivation for **id + id \* id**

**$E \rightarrow E + E$**

**$E \rightarrow E * E$**

**$E \rightarrow ( E )$**

**$E \rightarrow - E$**

**$E \rightarrow \text{id}$**

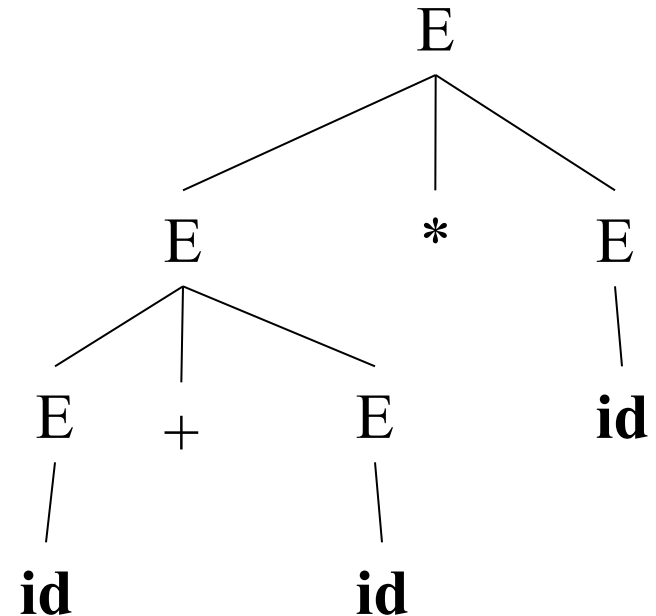
•  $E \Rightarrow E * E$

$\Rightarrow E + E * E$

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

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

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



Parse tree gives a  
meaning to the string

**(id+id)\*id vs id+(id\*id)**

# Rightmost derivation for **id + id \* id**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow -E$

$E \rightarrow \text{id}$

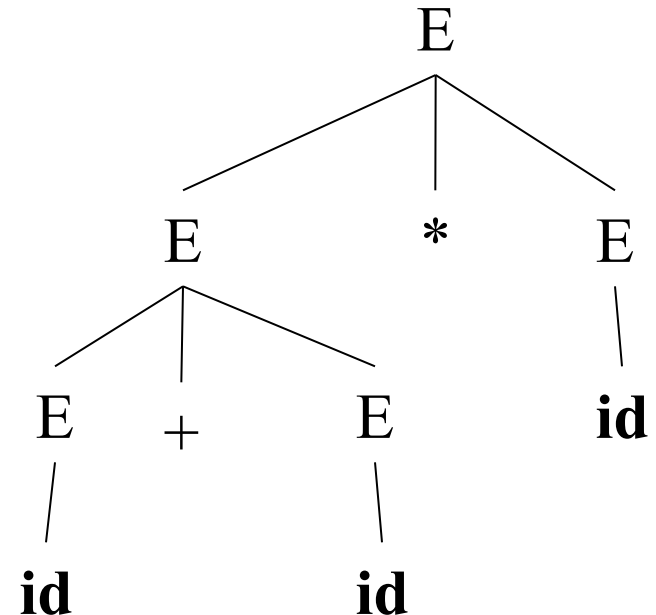
$E \Rightarrow E * E$

$\Rightarrow E * \text{id}$

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

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

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



# Rightmost vs. Leftmost Derivation

- Note that rightmost and leftmost derivations have the same parse tree
  - Every parse tree has a rightmost and a leftmost derivation
  - Important in resolving ambiguity

# Extra Slides

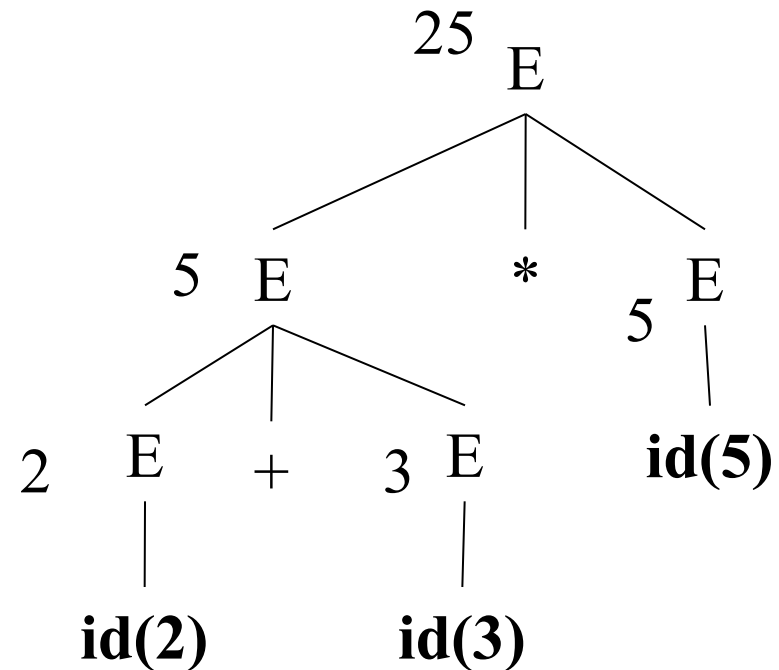
# Writing a CFG for a PL

- First write (or read) a reference grammar of what you want to be valid programs
- For now, we only worry about the structure, so the reference grammar might choose to over-generate in certain cases (e.g. `bool x = 20;` )
- Convert the reference grammar to a CFG
- Certain CFGs might be easier to work with than others (this is the **essence** of the study of CFGs and their parsing algorithms for compilers)



# Arithmetic Expressions

- $E \rightarrow E + E \{ \$\$ = \$1 + \$3 \}$
- $E \rightarrow E * E \{ \$\$ = \$1 * \$3 \}$
- $E \rightarrow ( E ) \{ \$\$ = \$2 \}$
- $E \rightarrow - E \{ \$\$ = -1 * \$2 \}$
- $E \rightarrow \text{id} \{ \$\$ = \$1 \}$



# CFG Notation

- Normal CFG notation

$$E \rightarrow E * E$$

$$E \rightarrow E + E$$

- Backus Naur notation

$$E ::= E * E \mid E + E$$

(an or-list of right hand sides)