Context-Free Grammars
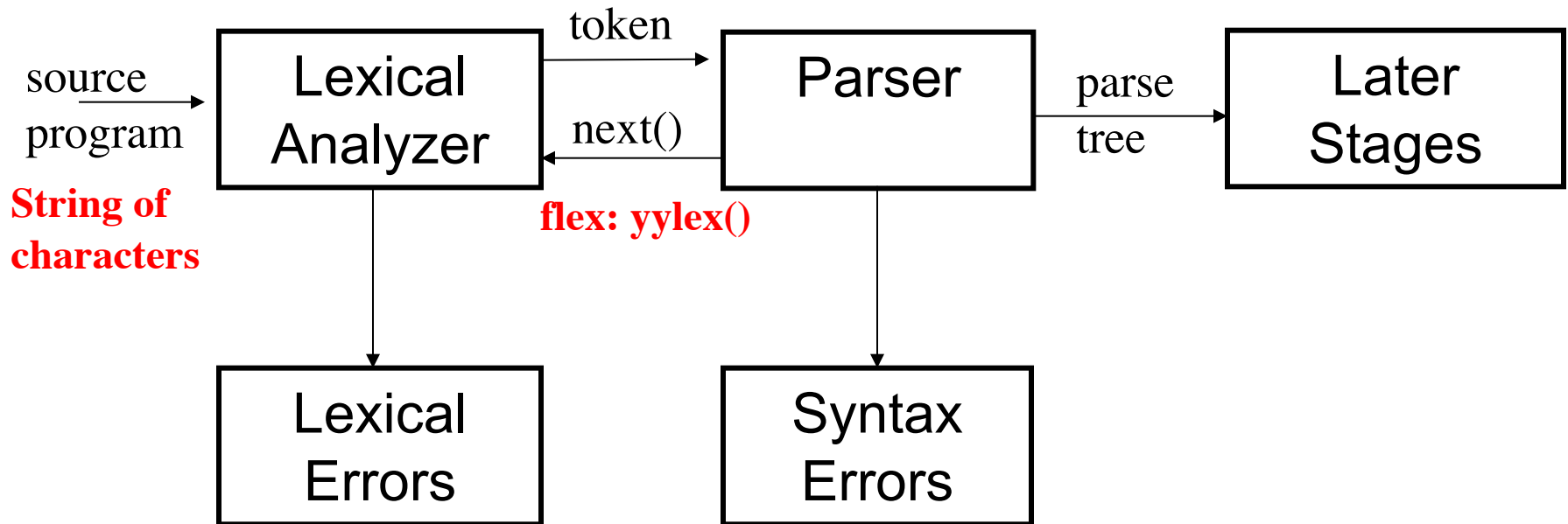
CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Parsing

source program → **Lexical Analyzer** — token → **Parser** — parse tree → **Later Stages**

next()

String of characters

flex: yylex()

Lexical Errors

Syntax Errors

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 $X \rightarrow Y_1 ... Y_n$

    $$X \in N$$

    $$Y_i \in N \cup T \cup \{\varepsilon\}$$

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 $X \rightarrow Y_1 ... Y_n$

3. Repeat (2) until there is no non-terminals

$$r1: S \rightarrow (S)$$
$$r2: S \rightarrow \varepsilon$$

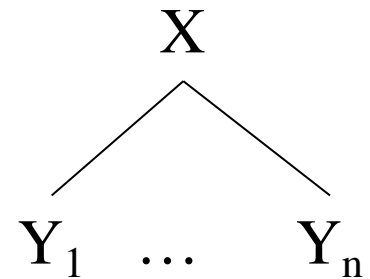$$S \Rightarrow^{r1} (S) \Rightarrow^{r1} ((S)) \Rightarrow^{r2} ((\ ))$$

Derivation and Parse Tree

- A derivation is a sequence of productions

$$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \ldots$$

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

Language of CFGs

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

$$\{\alpha_1 \ldots \alpha_n \mid \forall_i \; \alpha_i \in T \; and \; S \Rightarrow^* \alpha_1 \ldots \alpha_n\}$$

$$\{\varepsilon, (), (()), ((())), \ldots\}$$

Arithmetic Expressions

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

Derivation for
id + id * id

Leaves nodes:
terminals
Interior nodes:
non-terminals

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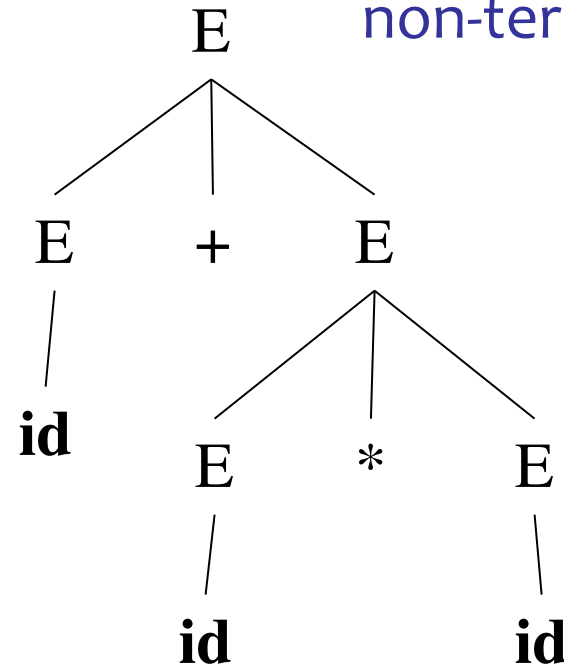$E \rightarrow \mathbf{id}$

$E \Rightarrow E + E$

$\Rightarrow \mathbf{id} + E$

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

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

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

Notation:
$E \Rightarrow^* \mathrm{id} + \mathrm{id} * \mathrm{id}$

Leftmost derivation for id + id * id

E → E + E
E → E * E
E → (E)
E → - E
E → **id**

E ⟹ E * E

⟹ E + E * E

⟹ **id** + E * E

⟹ **id** + **id** * E

⟹ **id** + **id** * **id**

Parse tree gives a meaning to the string

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

12

Rightmost derivation for id + id * id

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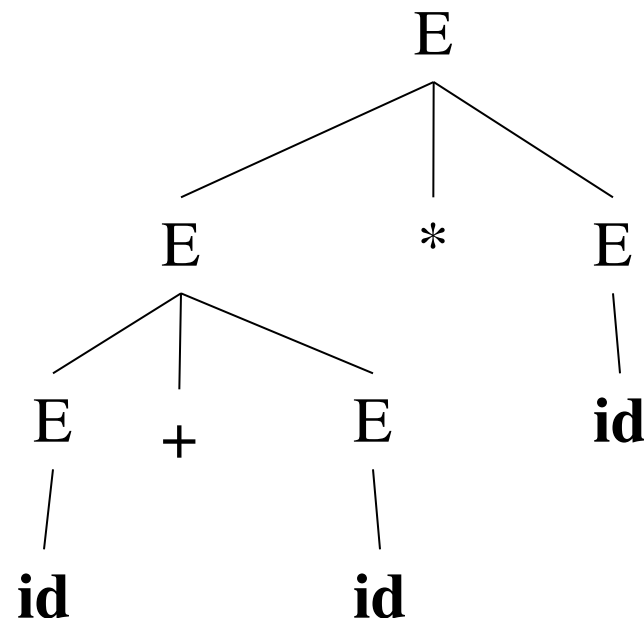$E \rightarrow (E)$
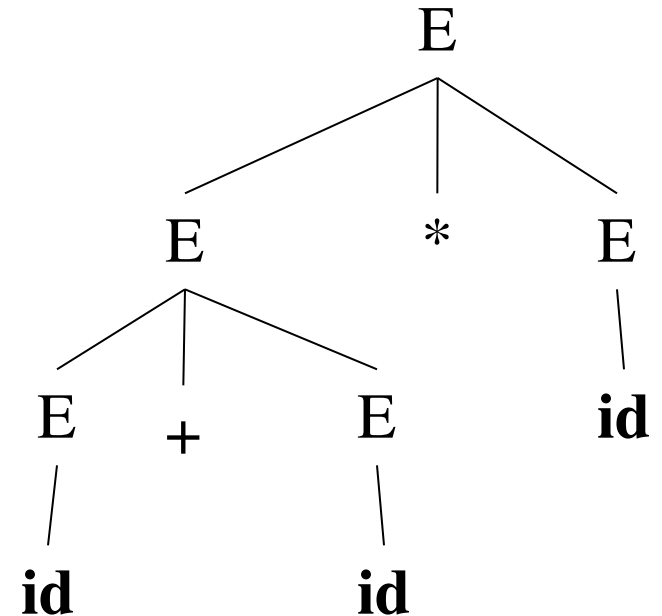$E \rightarrow - E$
$E \rightarrow \textbf{id}$

$E \Rightarrow E * E$

$\Rightarrow E * \textbf{id}$

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

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

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

# 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

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

Arithmetic Expressions

- E → E + E { $$ = $1 + $3 }
- E → E * E { $$ = $1 * $3 }
- E → (E) { $$ = $2 }
- E → - E { $$ = -1 * $2 }
- E → **id** { $$ = $1 }
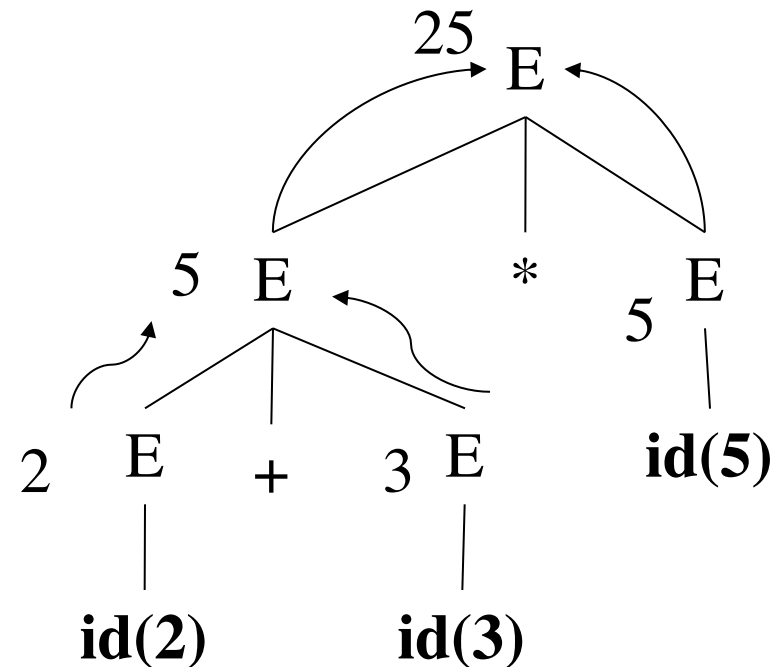
CFG Notation

- Normal CFG notation

  E $\rightarrow$ E * E

  E $\rightarrow$ E + E

- Backus Naur notation

  E ::= E * E | E + E

  (an or-list of right hand sides)

  Also:

  E = E "*" E | E "+" E