

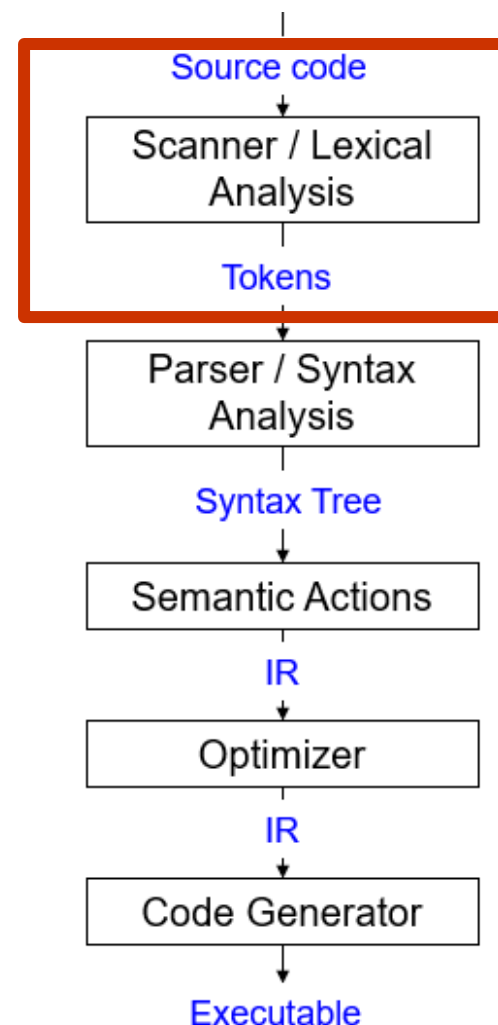
# CS323: Compilers

Spring 2023

## Week 3: Scanners (conclusion), Parsers

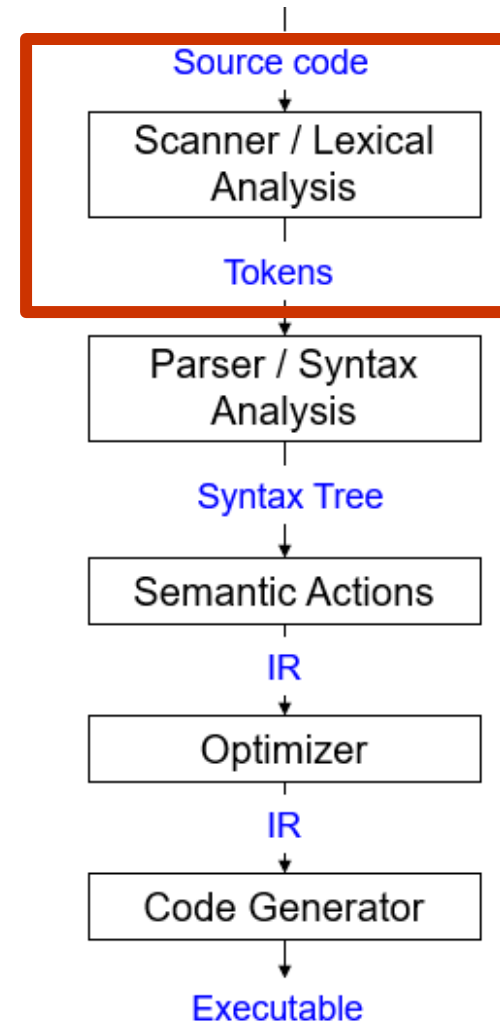
# Scanners (Summary)

- Also called *Lexers / Lexical Analyzers*
- Input: stream of letters (program text / source code), Output: sequence / list of *tokens*
- Token: a pair <category/class, value>
  - Category defines a string *pattern*
  - Value also called *lexeme*
  - Value is a *prefix* (and hence, is a substring)
  - Value matches on of the patterns that category defines
- Scan *left-to-right* in program text, *look-ahead* to identify tokens.
  - Look-ahead buffer size determined by language design



# Scanners (Summary)

- *Regular expressions* are used to formally define the patterns specified by token classes.
  - Some customization done while defining regular expressions: 1) Match the longest substring possible 2) Handle errors
- Tools such as Flex and ANTLR convert regular expressions to code. The code is your scanner implementation
  - The implementation typically converts regular expressions to *Finite Automata* (special kind of state diagram)
    - Automata are coded using efficient algorithms (E.g. Table-lookup method)
  - Efficient algorithms exist for substring matching (requiring single-pass over input program text)
    - Aho-Corasic, Knuth-Morris-Pratt (KMP)



# Parsers - Overview

- Also called syntax analyzers
- Determine two things:
  1. Is a program syntactically valid?  
(Analogy) is an English language sentence grammatically correct?
  2. What is the structure of programming language constructs? E.g. does the sequence\*

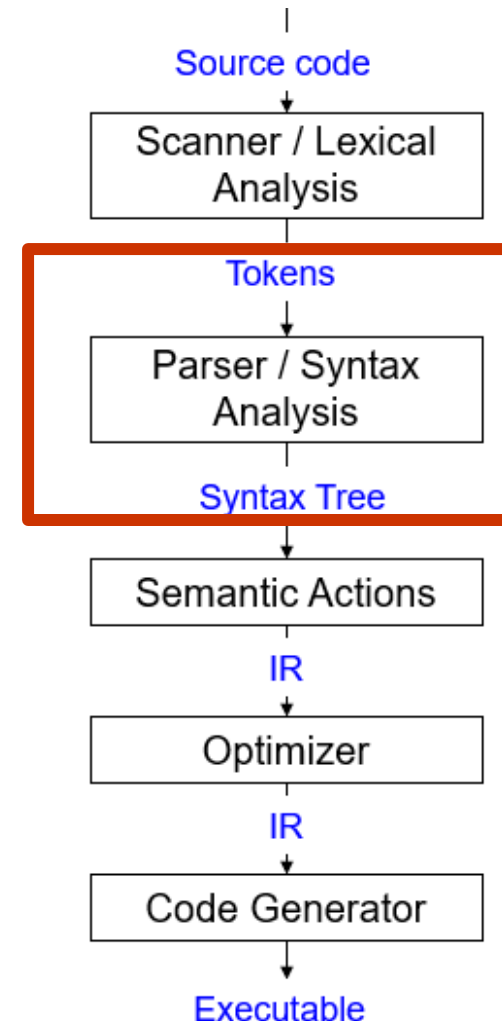
IF, ID(a), OP(<), ID(b), {, ID(a),  
ASSIGN, LIT(5), }}

refer to an `if` statement?

(Analogy) diagramming English sentences

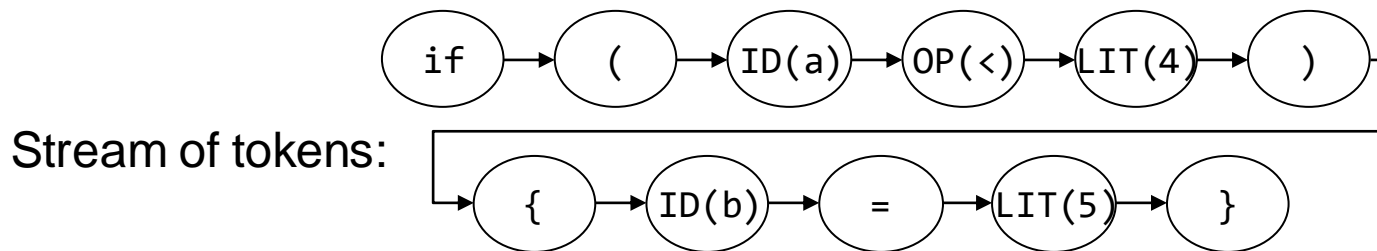
\* Corresponding program text:

```
if (a < 4) {  
    b = 5  
}
```

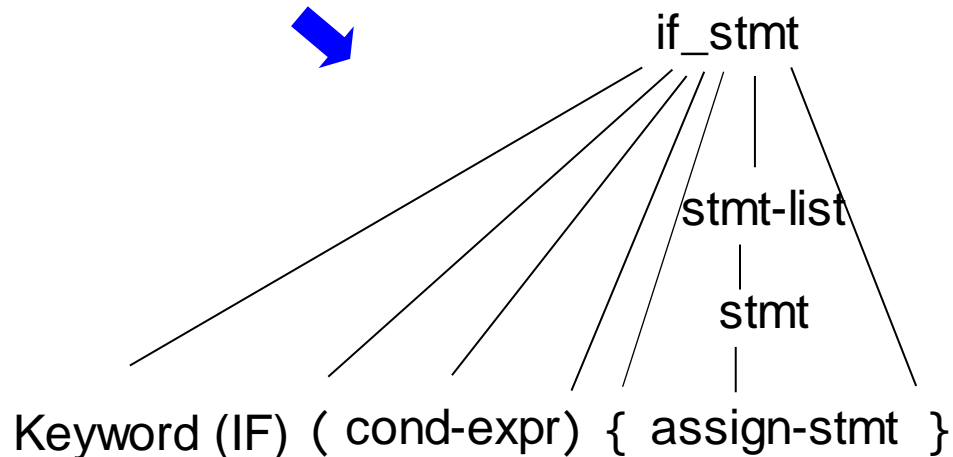


# Parsers - Overview

- Input: stream of tokens
- Output: Parse tree
  - sometimes implicit



Parse tree:



# Parsers – what do we need to know?

1. How do we define language constructs?
  - Context-free grammars
2. How do we determine: 1) valid strings in the language? 2) structure of program?
  - LL Parsers, LR Parsers
3. How do we write Parsers?
  - E.g. use a parser generator tool such as Bison

# Languages

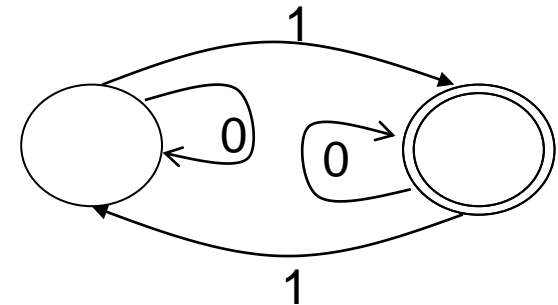
- A language is (possibly infinite) set of strings
- Regular expressions specify *regular languages*. However, regular languages are *weak formal languages* to describe the features of a practical programming language.

What set of strings does this FA accept?

The FA shown accepts all string with *odd number of 1s*.

What is the regular expression for the FA?

$(0^*10^*)(10^*10^*)^*$



Regular expressions can describe strings specifying *parity*:

$\{ \text{mod } k \mid k = \# \text{ states in FA} \}$

**weakness:** regular expressions can't describe a string of the form:  $\{ ({}^i) {}^i \mid i \geq 1 \}$

# Regular Languages

- Regular expressions can't describe a string of the form:

$$\{ ({}^i) {}^i \mid i \geq 1 \}$$

E.g. Parenthesized expressions

`((2+3)*5)`

*Programming language syntax is i.e. recursive*

`(( ( int x; ) ))`

Nested structures:

IF  
  IF  
    IF  
      FI  
    FI  
  FI  
FI



# Context Free Grammar (CFG)

- Natural notation for describing recursive structure definitions. Hence, suitable for specifying language constructs.
- Consist of:
  - A set of *Terminals* (T)
  - A set of *Non-terminals* (N)
  - A *Start Symbol* ( $S \in N$ )
  - A *set of Productions* ( $X \rightarrow Y_1 \dots Y_N$ ) ( aka. rules)

$$P: X \longrightarrow Y_1 Y_2 Y_3 \dots Y_N \quad X \in N, \quad Y_i \in N \cup T \cup \epsilon/\lambda$$

# Context Free Grammar (CFG)

- Grammar  $G = (T, N, S, P)$

E.g.  $G = (\{a, b\}, \{S, A, B\}, S, \{S \rightarrow AB, A \rightarrow Aa, A \rightarrow a, B \rightarrow Bb, B \rightarrow b\})$

- Implicit meanings
  - First rule listed in the set of productions contains start symbol (on the left-hand side)
  - In the set of productions, you can replace the symbol X (appearing on the right-hand side only) with the string of symbols that are on the right-hand side of a rule, which has X (on the left-hand side)

# Context Free Grammar (CFG)

1. Begin with only S as the initial string

2. Replace S

- S replaced with AB

3. Repeat 2 until the string contains only terminals

i. AB replaced with aB

ii. aB replaced with ab

$G = (T, N, S, P)$   
 $P: \{ S \rightarrow AB, \\ A \rightarrow Aa, \\ A \rightarrow a, \\ B \rightarrow Bb, \\ B \rightarrow b \}$

**Summary:** we move from S to a string of terminals through a series of transformations:

$\alpha_0 \rightarrow \dots \rightarrow \alpha_n$  where  $\alpha_1 \dots \alpha_n$  are strings

Shorthand notation:  $\alpha_0 \xrightarrow{*} \alpha_n$

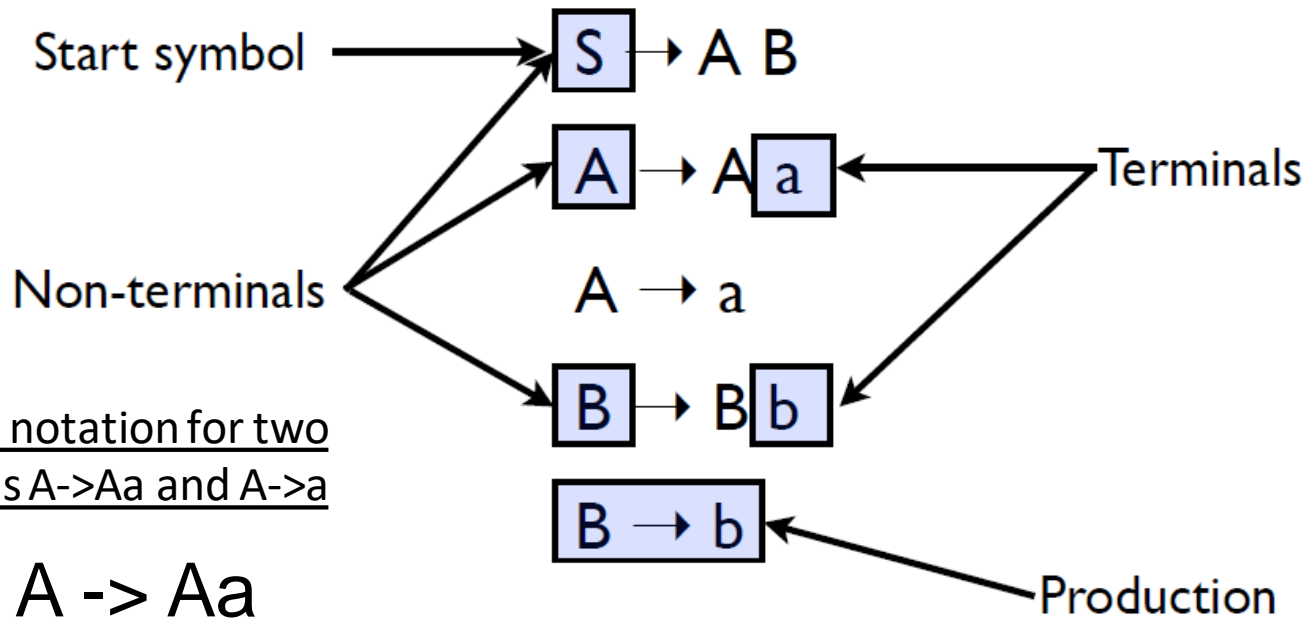
# Detour: Context-Sensitive Grammar

- Can have context-sensitive grammar and languages (think:  $aB \rightarrow ab$ )
  - Cannot replace right-hand side with left-hand side irrespective of the context.
  - E.g.  $aB \rightarrow ab$  lays down a context: 'a' must be a prefix in order to transform the string "aB" to a string of terminals "ab"
    - ccaBb can be replaced by ccabb

*Is grammar G context-free?*

$$\begin{aligned} G &= (T, N, S, P) \\ P: \{ & S \rightarrow AB, \\ & A \rightarrow Aa, \\ & A \rightarrow a, \\ & B \rightarrow Bb, \\ & B \rightarrow b \} \end{aligned}$$

# Simple grammar (Summary)



Alternative notation for two productions  $A \rightarrow Aa$  and  $A \rightarrow a$

$A \rightarrow Aa$   
 $| a$

*Backus Naur Form (BNF)*

# Programming language syntax

- Programming language syntax is defined with CFGs
- Constructs in language become non-terminals
- May use auxiliary non-terminals to make it easier to define constructs

`if_stmt`  $\rightarrow$  if ( `cond_expr` ) then `statement` `else_part`

`else_part`  $\rightarrow$  else `statement`

`else_part`  $\rightarrow \lambda$

- Tokens in language become terminals

# Language of the Grammar

- Language  $L(G)$  of the context-free grammar  $G$ 
  - Set of strings that can be derived from  $S$
  - $\{a_1a_2a_3 \dots a_N \mid a_i \in T \forall i \text{ and } S \xrightarrow{*} a_1a_2a_3 \dots a_N\}$
  - Is called context-free language
    - All regular languages are context-free but not vice-versa.
    - Can have many grammars generating same language.

# String Derivations: Does a string belong to the Language?

- How do we apply the grammar rules to determine the acceptability of a string? (i.e. the string belongs to the language,  $L(G)$ , specified by the CFG  $G$ )
  - Begin with  $S$
  - Replace  $S$
  - Repeat till string contains terminals only. Why terminals only?  
 *$L(G)$  must contain strings of terminals only*
- Notation:
  - We will use Greek letters to denote strings containing non-terminals and terminals
- Derivations: sequence of rules applied to produce the string of terminals



# Generating strings (Example)

$S \rightarrow A B$

$A \rightarrow A a$

$A \rightarrow a$

$B \rightarrow B b$

$B \rightarrow b$

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- Some productions may rewrite to  $\lambda$ . That just removes the non-terminal

To derive the string “a a b b b” we can do the following rewrites:

$S \Rightarrow A B \Rightarrow A a B \Rightarrow a a B \Rightarrow a a B b \Rightarrow$   
 $a a B b b \Rightarrow a a b b b$

# CFG and Parsers

- Is it enough if parsers answer “yes” or “no” to check if a string belongs to context-free language?
  - Also need a parse tree
- What if the answer is a “no”?
  - Handle errors
- How do we implement CFGs?
  - E.g. Bison

# Exercise

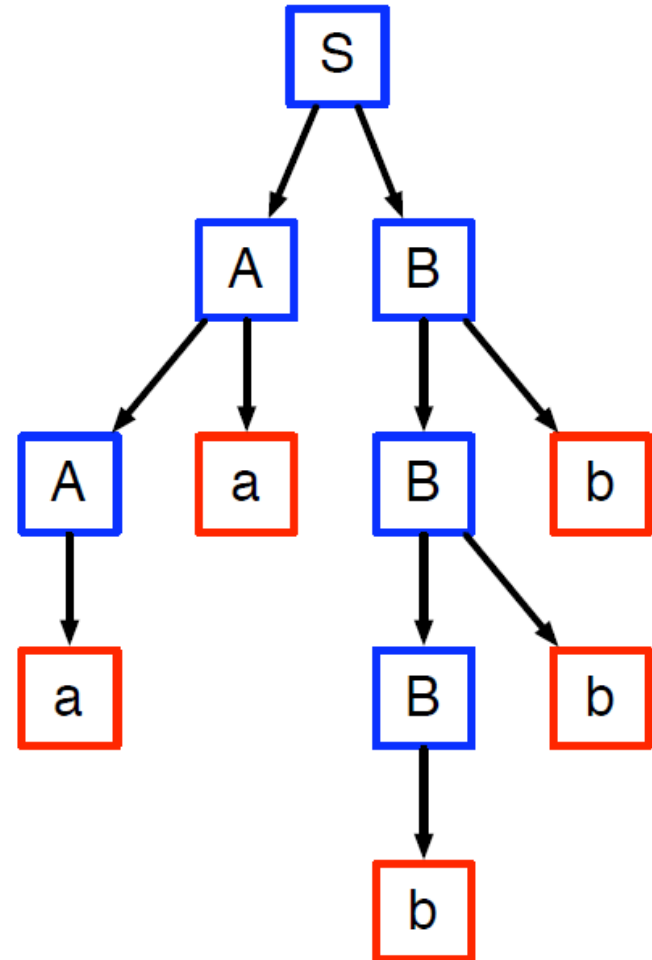
Which of the below strings are accepted by the grammar:

- 1:  $A \rightarrow aAa$
- 2:  $A \rightarrow bBb$
- 3:  $A \rightarrow \lambda$
- 4:  $B \rightarrow cA$
- 5:  $B \rightarrow \lambda$

- 1. abcba      1- $\rightarrow$ 2- $\rightarrow$ 4- $\rightarrow$ 3
- 2. abcbca
- 3. abba      1- $\rightarrow$ 2- $\rightarrow$ 5
- 4. abca

# Parse trees

- Tree which shows how a string was produced by a language
- Interior nodes of tree: non-terminals
  - Children: the terminals and non-terminals generated by applying a production rule
- Leaf nodes: terminals



# Derivations and Parse Trees

- Recall: Derivation is a sequence of rules applied to produce a string
  - $S \rightarrow \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_n$
- A derivation defines a parse tree
  - Parse tree is an alternative way to gather information on how the string was derived
  - A parse tree may have many derivations (think: different permutations of  $\alpha$  )

# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $\quad \mid E * E$

3:  $\quad \mid id$

- Produce derivations for the string: **id\*id+id**

Apply 1: **Start with E**, the start symbol      Parse Tree

**E**



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

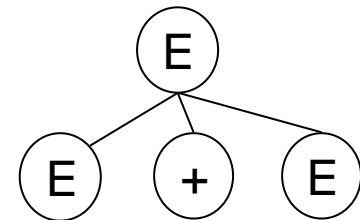
- Produce derivations for the string: **id\*id+id**

Apply 1: Replace E with E + E

E

E+E

Parse Tree





# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id * id + id$**

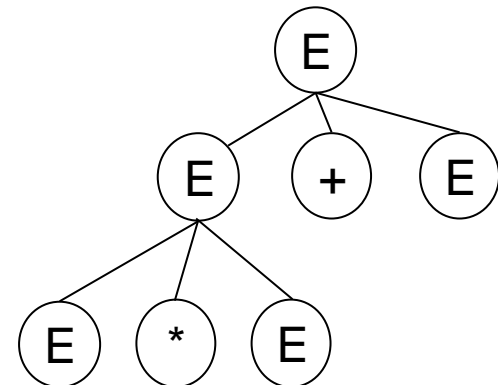
Apply 2: Replace  $E$  with  $E * E$

$E$

$E + E$

$E * E + E$

Parse Tree



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

Apply 3: Replace E with id

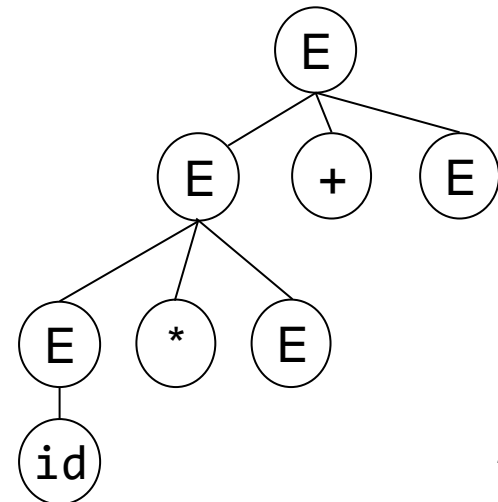
E

E+E

E\*E+E

id\*E+E

Parse Tree



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

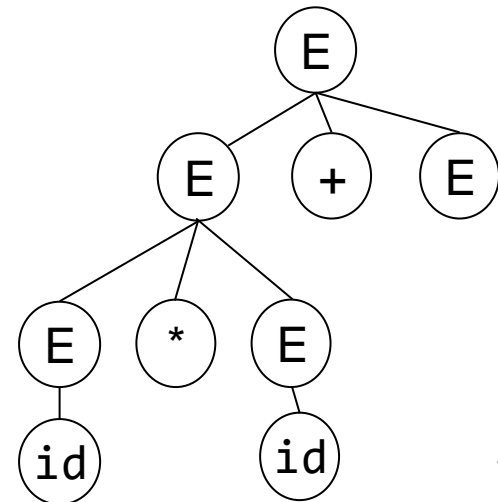
3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

Apply 3: Replace E with id

E  
E+E  
E\*E+E  
id\*E+E  
id\*id+E

Parse Tree



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

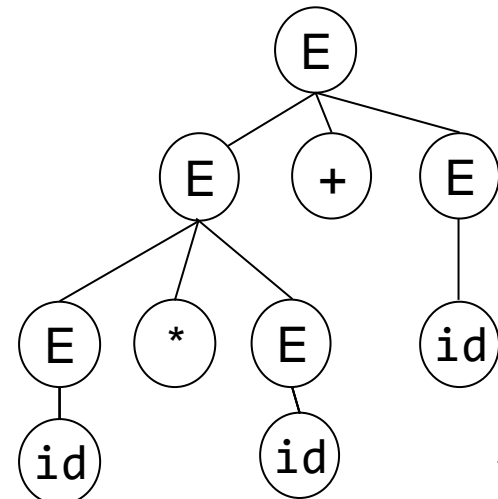
3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

Apply 3: Replace E with id

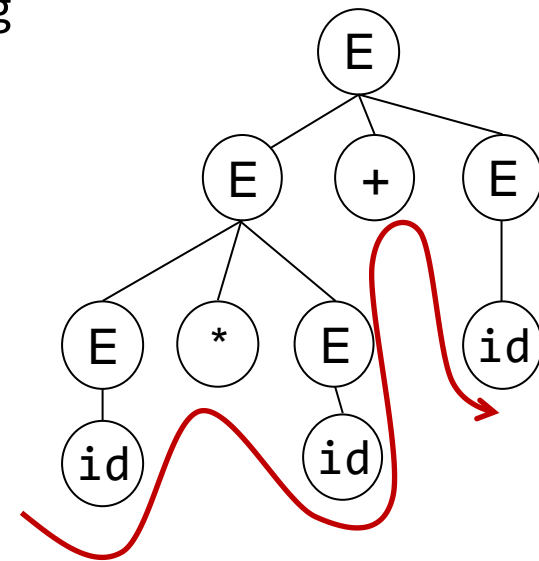
E  
E+E  
E\*E+E  
id\*E+E  
id\*id+E  
id\*id+id

Parse Tree



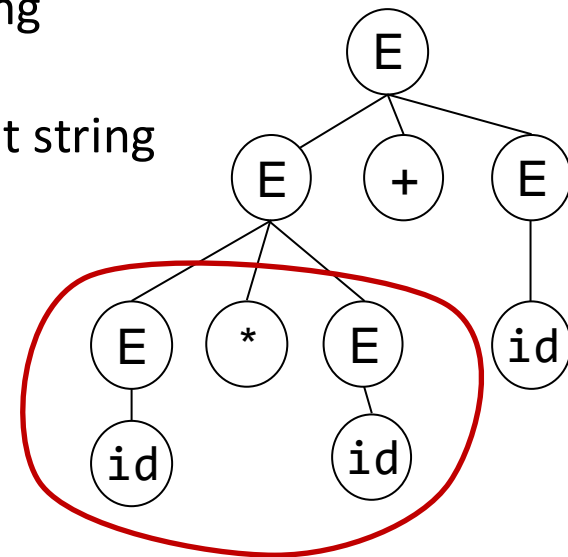
# Derivations and Parse Trees

- Note in previous slides:
  - Replacement done on left-most non-terminal in the string - **called left-most derivation**
  - Terminals at leaves and non-terminal as interior nodes
  - Inorder traversal of leaves produces input string `id*id+id`



# Derivations and Parse Trees

- Note in previous slides:
  - Replacement done on left-most non-terminal in the string - **called left-most derivation**
  - Terminals at leaves and non-terminal as interior nodes
  - Inorder traversal of leaves produces input string  $id*id+id$
  - Parse tree shows association of operations. Input string doesn't
    - \* associated with identifiers in the subtree  
 $(id * id)+id$



# Derivations and Parse Trees

- Consider the same grammar (having the following rules):

1:  $E \rightarrow E + E$

2:  $\quad \mid E * E$

3:  $\quad \mid id$

- Produce derivations for the string: **id\*id+id**
  - Using **right-most derivations**  
i.e. replace the right-most non-terminal

# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

Start with E, the start symbol



Parse Tree

E



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

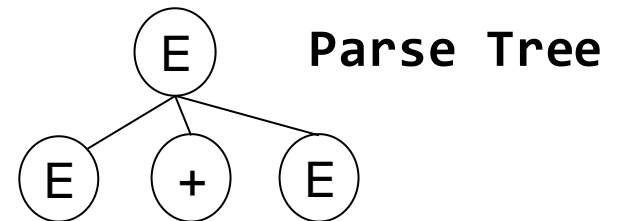
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

Apply 2: Replace E with E+E

E  
E+E



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

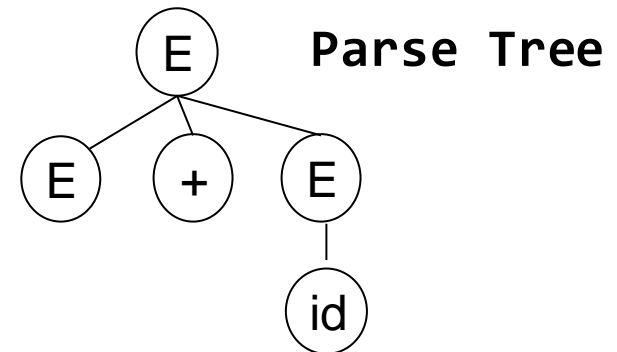
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

Apply 1: Replace E with id

E  
E+E  
E+id



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

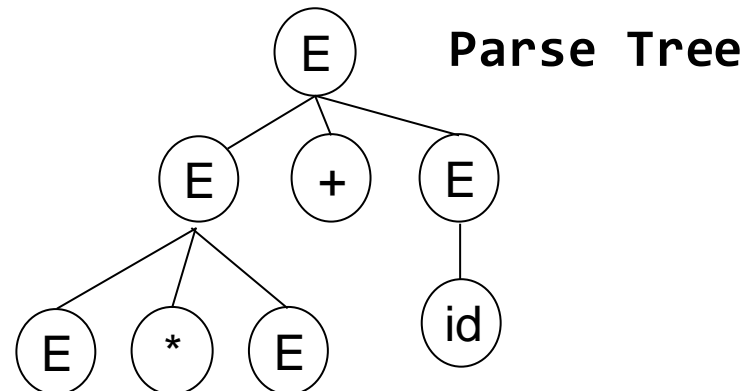
Apply 3: Replace E with  $E * E$

E

E+E

E+id

E\*E+id



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

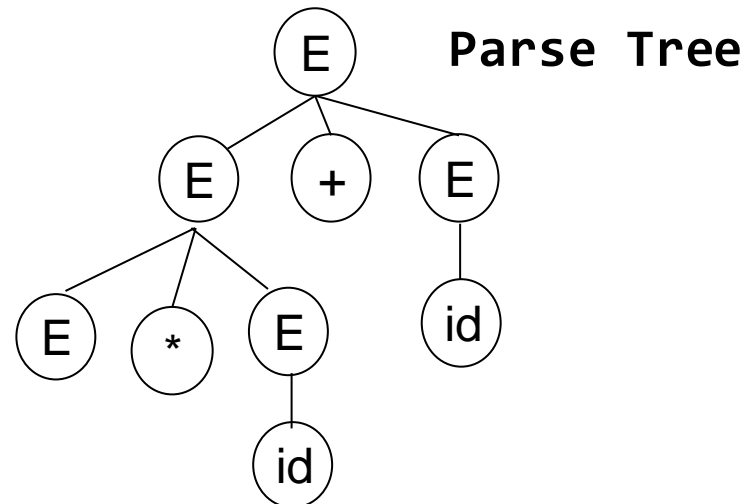
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

Apply 3: Replace E with id

E  
E+E  
E+id  
E\*E+id  
E\*id+id



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

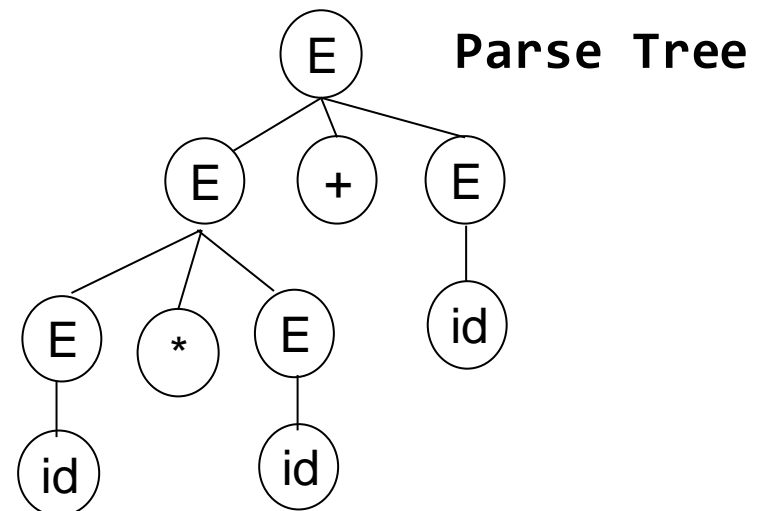
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

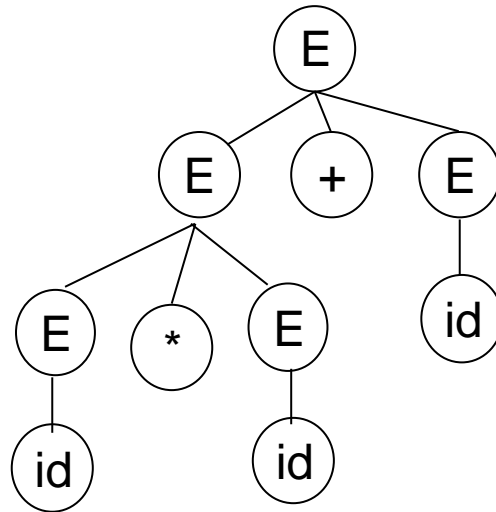
Apply 3: Replace E with id

E  
E+E  
E+id  
E\*E+id  
E\*id+id  
id\*id+id



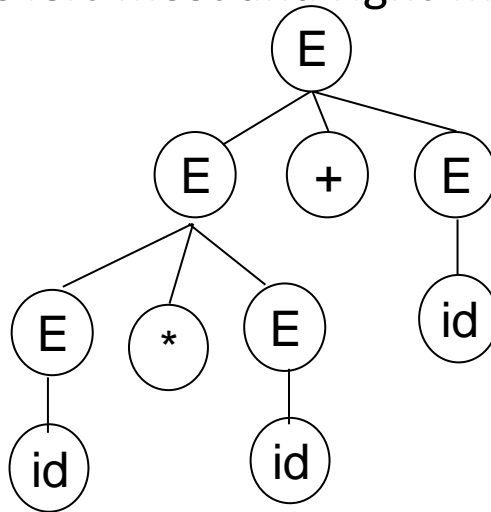
# Derivations and Parse Trees

- We get the **same parse tree** using left-most and right-most derivations.
  - Every parse tree has left-most and right-most (and any random order) derivations.



# Derivations and Parse Trees

- We get the **same parse tree** using left-most and right-most derivations.
  - Every parse tree has left-most and right-most (and any random order) derivations.



- But there could be a string (or more than one strings) for which there exists derivations that would get different parse trees

# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

Start with E, the start symbol



Parse Tree

E



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

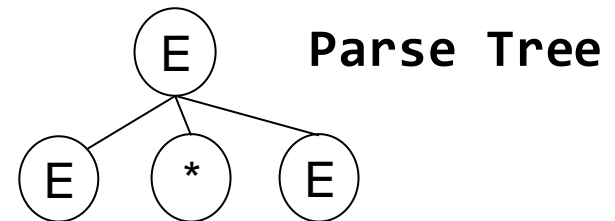
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

Apply 2: **Replace E with E\*E** *Earlier it was replace E with E+E*

E  
E\*E



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

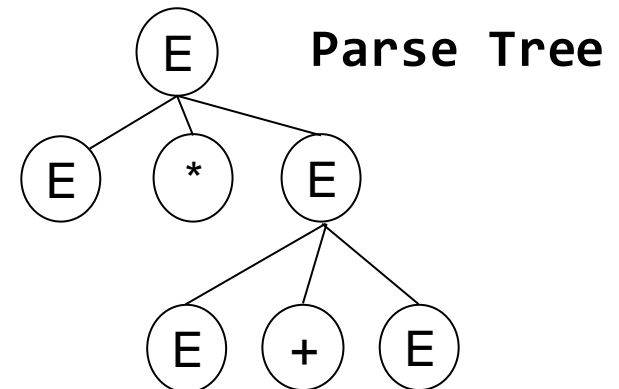
$$2: \quad | \quad E * E$$

```
3: | id
```

- Produce derivations for the string: **id\*id+id**

## Apply 1: Replace E with E+E

# E

 $E * E$ 
$$E * E + E$$


# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

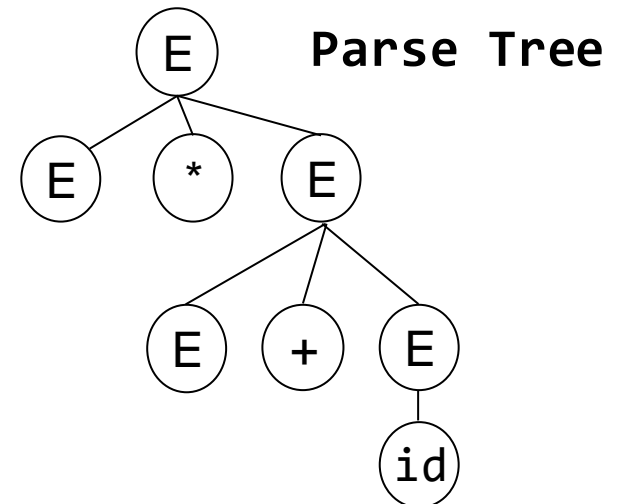
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

Apply 3: Replace E with id

E  
E \* E  
E \* E + E  
E \* E + id



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

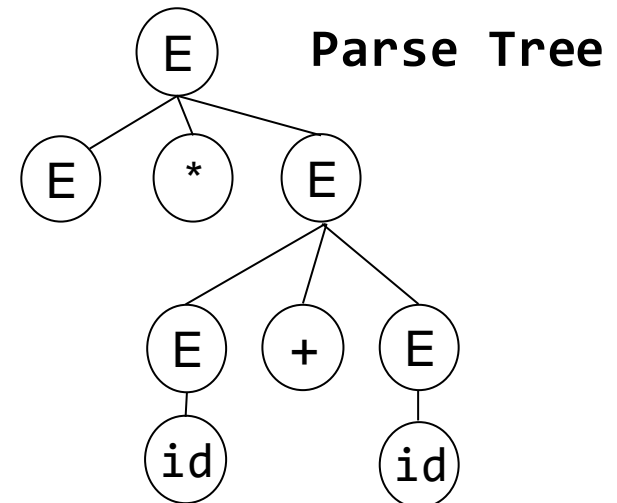
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string:  **$id*id+id$**

Apply 3: Replace E with id

E  
E \* E  
E \* E + E  
E \* E + id  
E \* id + id



# Derivations and Parse Trees

- Consider the grammar with the following rules:

1:  $E \rightarrow E + E$

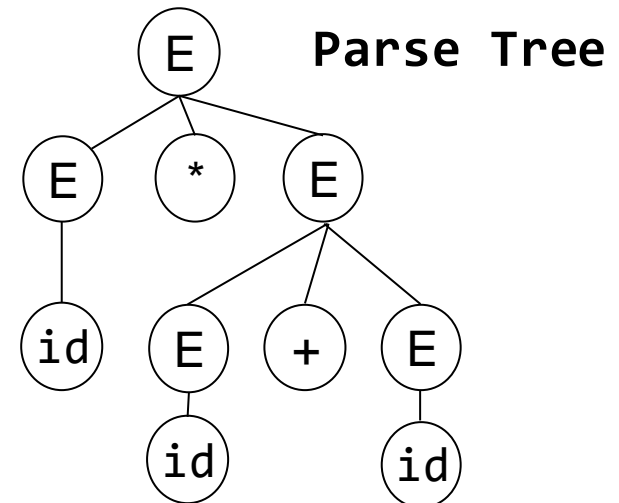
2:  $E \rightarrow E * E$

3:  $E \rightarrow id$

- Produce derivations for the string: **id\*id+id**

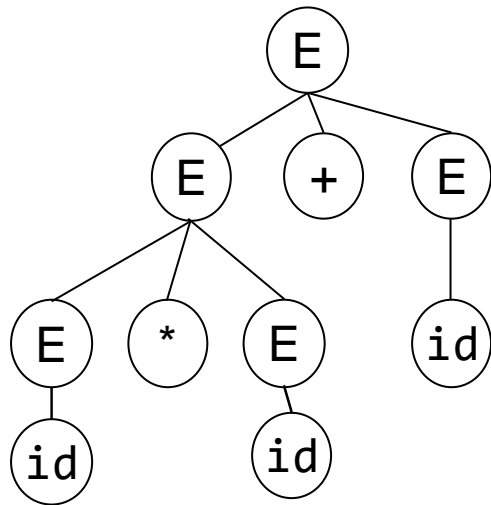
Apply 3: Replace E with id

E  
E \* E  
E \* E + E  
E \* E + id  
E \* id + id  
id \* id + id

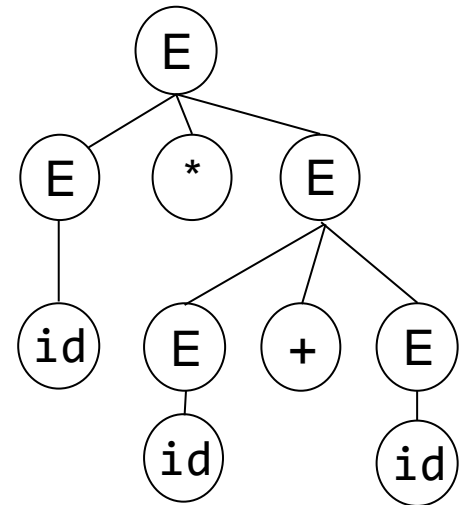


# Derivations and Parse Trees

- Input string:  $\text{id} * \text{id} + \text{id}$



**earlier**



**now**

- Inorder traversal of leaves in both trees produces the same input string

# Ambiguous Grammar

- Grammar that produces more than one parse tree for some string

```
1: E -> E + E
2:   | E * E
3:   | id
```

# Ambiguity – what to do?

- Ignore it (let it be ambiguous)
  - Give hints to other components of the compiler on how to resolve it
- Fix it (Manually)
  - May make the grammar complicated and difficult to maintain

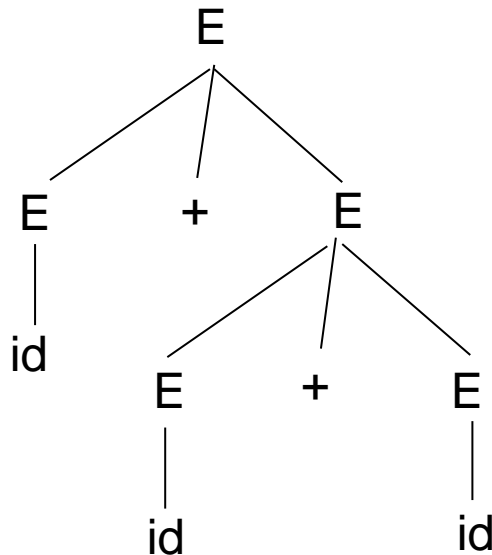


# Ambiguity – ignore

- Grammar:  $E \rightarrow \underline{E + E} \mid \underline{id}$

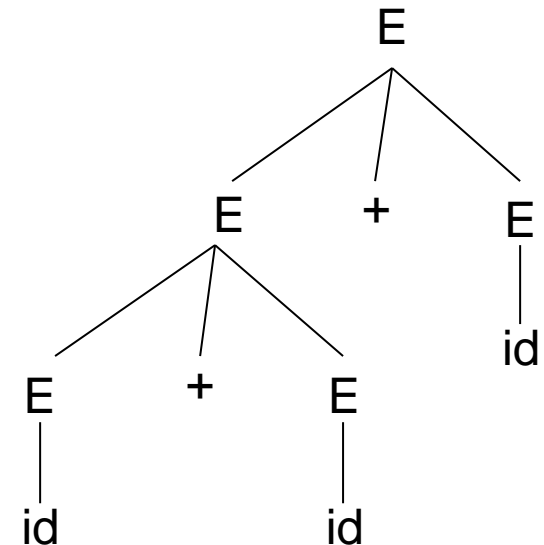
input: `id+id+id`

$E \rightarrow E + E$   
 $E \rightarrow id + E$   
 $E \rightarrow id + E + E$   
 $E \rightarrow id + id + E$   
 $E \rightarrow id + id + id$



Matches the input, which would be evaluated (later) as: `id+(id+id)`

$E \rightarrow E + E$   
 $E \rightarrow E + E + E$   
 $E \rightarrow id + E + E$   
 $E \rightarrow id + id + E$   
 $E \rightarrow id + id + id$



`(id+id)+id`

`%left +`



Provide hint (in Bison). Associativity declaration.

(left associative for +. So, produces the parse tree on the right)

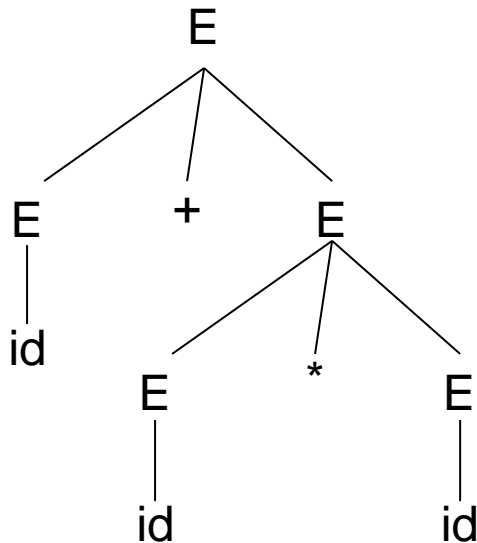
# Ambiguity - ignore

•  $E \rightarrow E + E \mid E * E \mid id$



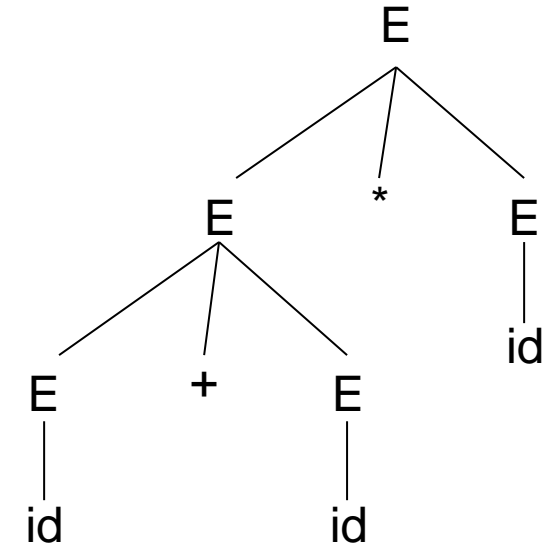
$E \rightarrow E + E$   
 $E \rightarrow id + E$   
 $E \rightarrow id + E * E$   
 $E \rightarrow id + id * E$   
 $E \rightarrow id + id * id$

Produces  
tree for:  
 $id + (id * id)$



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

Produces  
tree for:  
 $(id + id) * id$



%left +  
%left \*



*Tells that \* has higher precedence over + and both are left associative. So, we get the tree on left.*

# Ambiguity – fixing

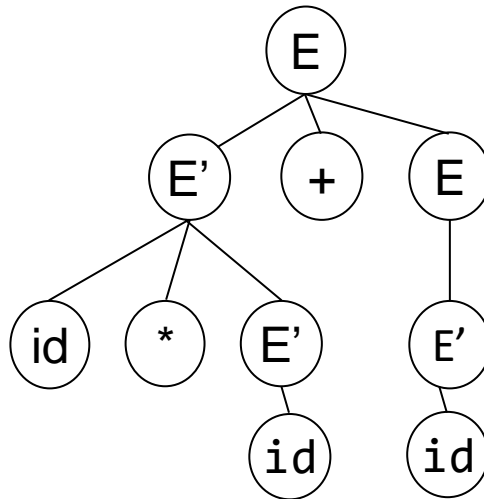
- Rewrite  $E \rightarrow E + E$  as:

$$\begin{array}{l} | E * E \\ | id \end{array}$$

$$\begin{array}{l} E \rightarrow E' + E \mid E' \\ E' \rightarrow id * E' \mid id \\ \quad \mid (E) * E' \mid (E) \end{array}$$

Parse tree for input id\*id+id

$E \rightarrow E' + E$   
 $E' \rightarrow id * E'$   
 $E' \rightarrow id$   
 $E \rightarrow E'$   
 $E' \rightarrow id$



If you want to handle parenthesized expressions such as  $(id+id)*id$

- E controls generation of +
- E' controls generation of \*

\*'s are always nested deeper in the parse tree.

*is the above sequence left-most or right-most derivation?*

# Ambiguity Fixing - Exercise

**Exercise:** *Is this grammar ambiguous? Draw parse tree(s) for the following*    **input:** `if e1 then if e2 then s1 else s2`

```
1: STMT -> if EXPR then STMT
2:       |   if EXPR then STMT else STMT
3:       |   s1
4:       |   s2
5: EXPR -> e1 | e2
```

# Ambiguity Fixing - Exercise

**Exercise:** *Is this grammar ambiguous? Draw parse trees for the following*      **input:** `if e1 then if e2 then s1 else s2`

1: STMT -> if EXPR then STMT  
2:       | if EXPR then STMT else STMT  
3:       | s1  
4:       | s2  
5: EXPR -> e1 | e2

# Ambiguity Fixing - Exercise

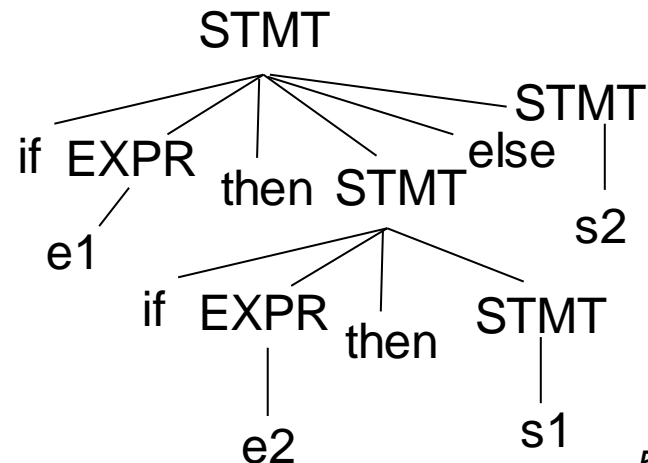
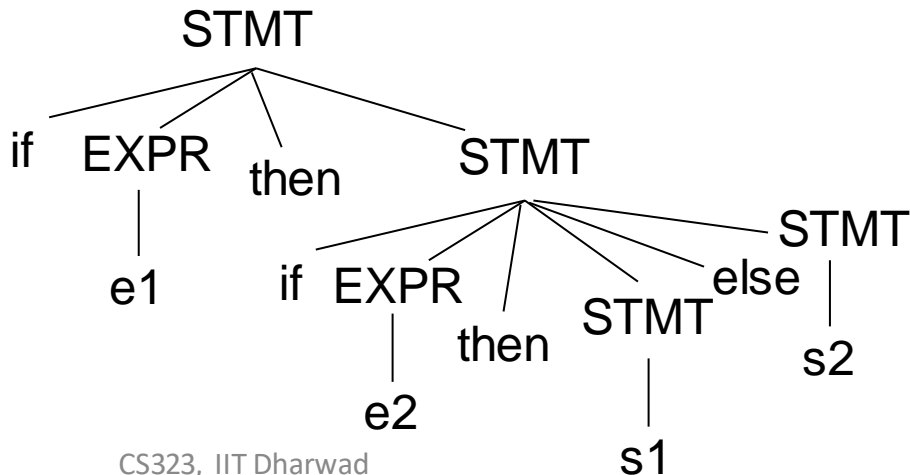
**Exercise:** *Is this grammar ambiguous? Draw parse trees for the following*      **input:** `if e1 then if e2 then s1 else s2`

1: STMT -> if EXPR then STMT  
2:        |    if EXPR then STMT else STMT  
3:        |    s1  
4:        |    s2  
5: EXPR -> e1 | e2

# Ambiguity Fixing - Exercise

**Exercise:** *Is this grammar ambiguous? Draw parse trees for the following*      **input:** if e1 then if e2 then s1 else s2

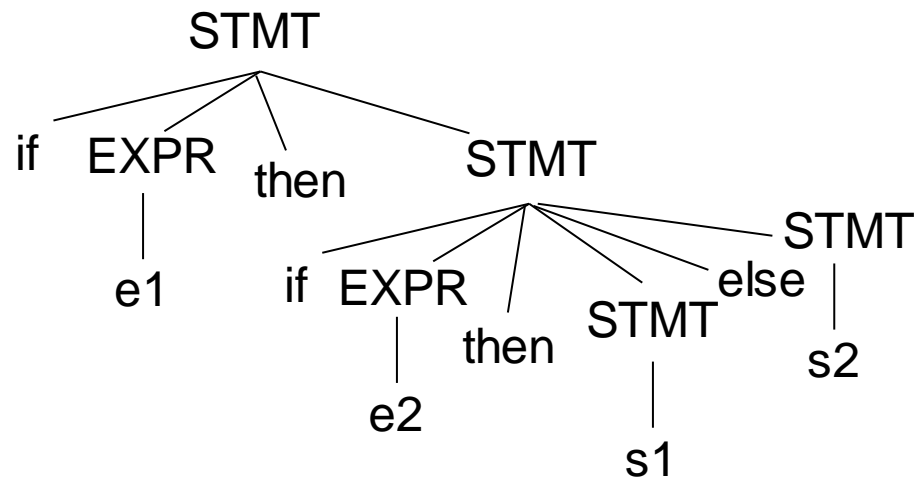
1: STMT  $\rightarrow$  if EXPR then STMT  
2:        |    if EXPR then STMT else STMT  
3:        |    s1  
4:        |    s2  
5: EXPR  $\rightarrow$  e1 | e2



# Ambiguity Fixing - Exercise

**Exercise:** Which if is the else associated with?

**String:** if e1 then if e2 then s1 else s2

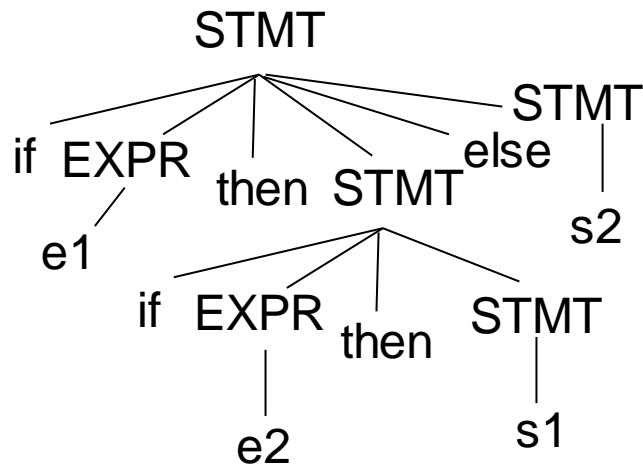




# Ambiguity Fixing - Exercise

**Exercise:** Which if is the else associated with?

**String:** if e1 then if e2 then s1 else s2



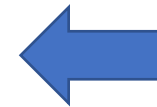
# Ambiguity Fixing - Exercise

**Exercise:** Rewrite the grammar to make it unambiguous.

```
1: STMT -> if EXPR then STMT
2:       | if EXPR then STMT else STMT
3:       | s1
4:       | s2
5: EXPR -> e1 | e2
```

# CFG and Parsers

- Is it enough if parsers answer “yes” or “no” to check if a string belongs to context-free language?
  - Also need a parse tree
- What if the answer is a “no”?
  - Handle errors
- How do we implement CFGs?
  - E.g. Bison



Next

# Error Handling

- Objective: detect invalid programs and provide meaningful feedback to programmer
  - Report errors accurately
  - Recover from errors quickly
  - Don't slow down compilation

# Error Types

- Many types of errors:
  - Lexical – `int 9abc; //invalid identifier`
  - Syntactic – extra brace inserted {
  - Semantic – `float sqr; sqr(2);`  
//use variable name with function call syntax
  - Logical – use = instead of ==

# Error Handling - Types

1. Panic mode
2. Error production
3. Automatic local or global correction