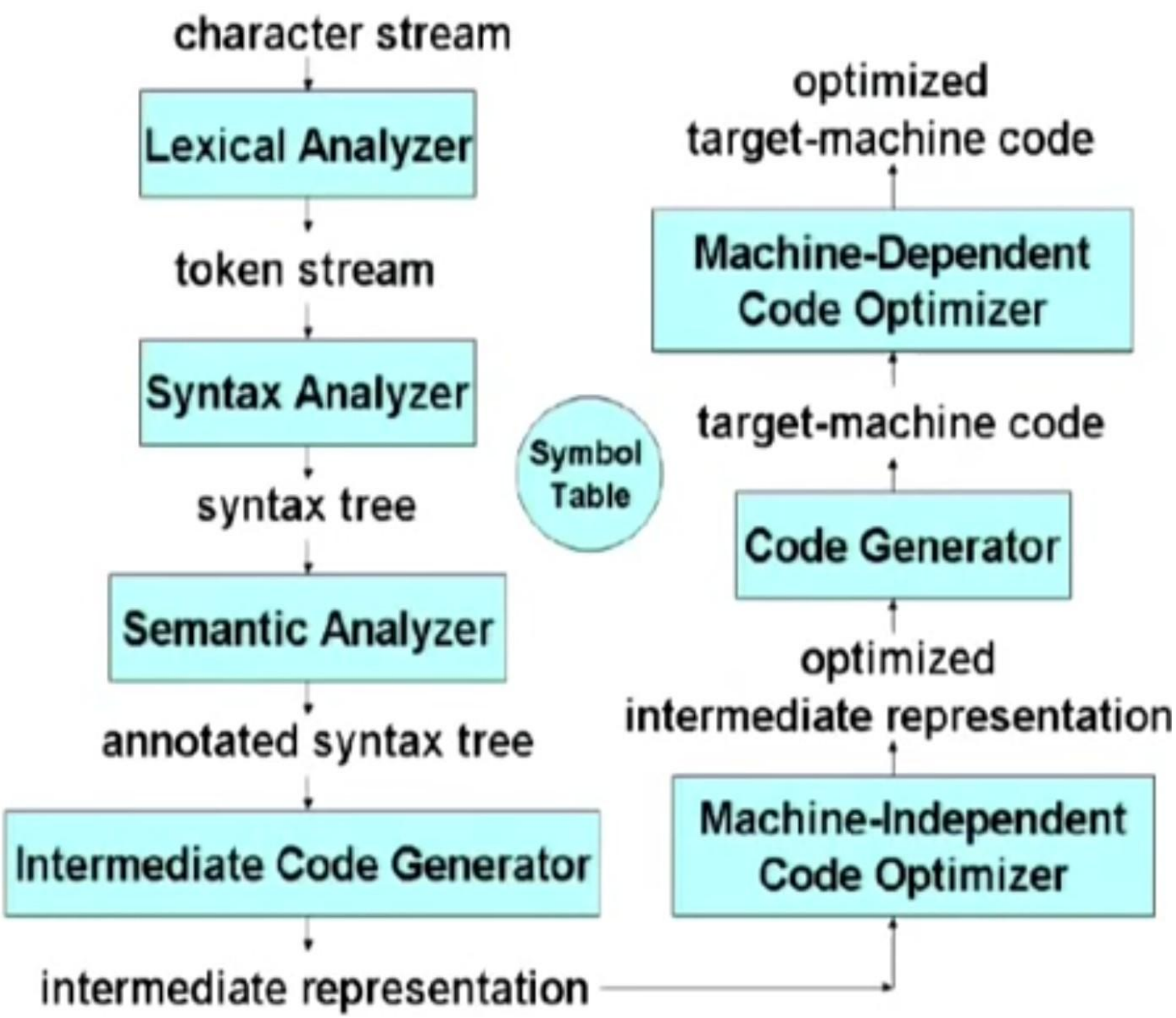
Syntax Analysis
(Part 1)

CSE 415: Compiler Construction

Phases of a Compiler



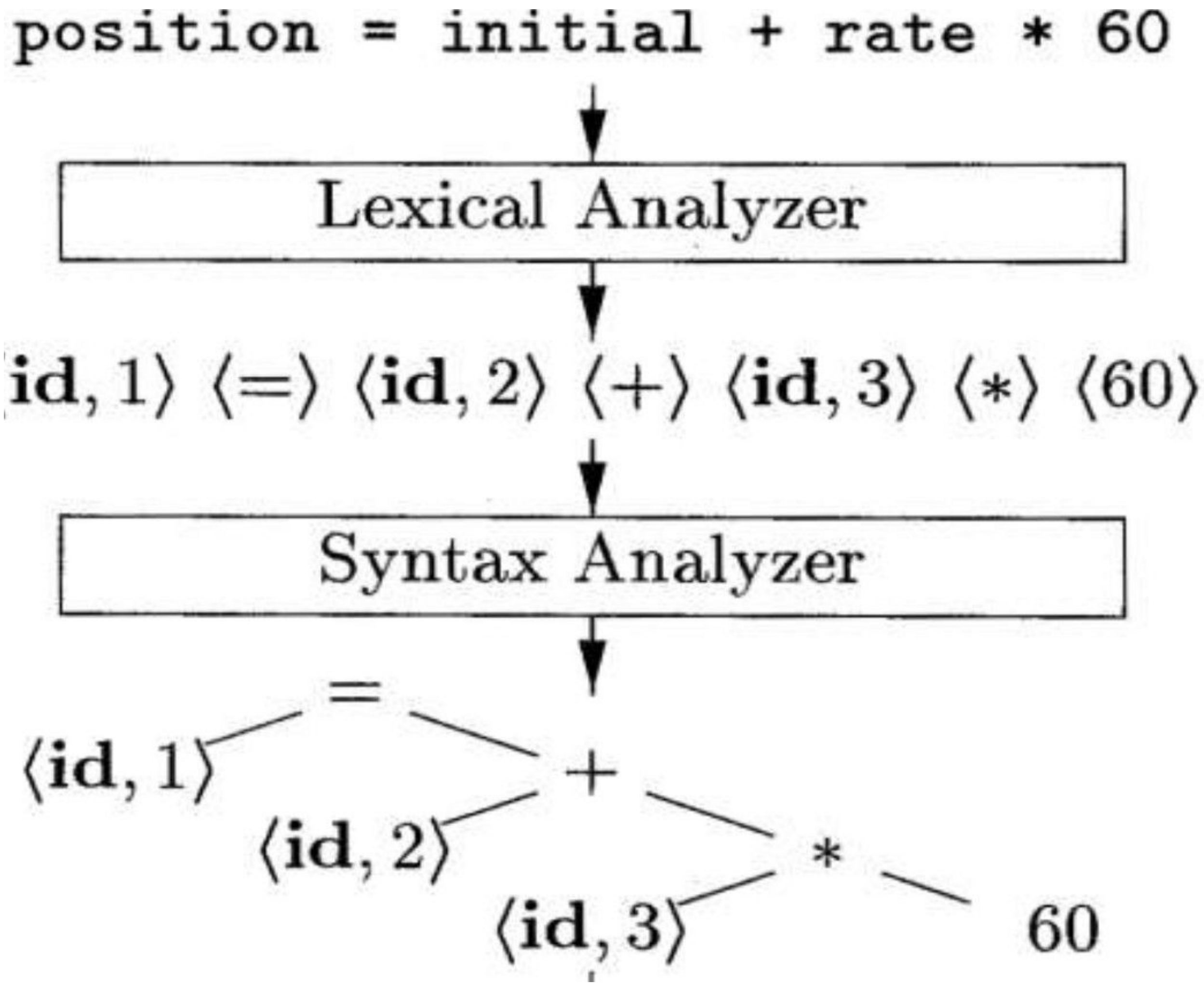
Grammars

- Every programming language has precise grammar rules that describe the syntactic structure of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.
- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars
- Parsers or syntax analyzers are generated *for* a particular grammar
- Context-free grammars are usually used for syntax specification of programming languages

What Parsing or Syntax Analysis

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on *context-free* grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

What Parsing or Syntax Analysis



Context Free Grammar (CFG)

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)	(2)	(3)	(4)
$E \rightarrow E + E$	$S \rightarrow 0S0$	$S \rightarrow aSb$	$S \rightarrow aB \mid bA$
$E \rightarrow E * E$	$S \rightarrow 1S1$	$S \rightarrow \epsilon$	$A \rightarrow a \mid aS \mid bAA$
$E \rightarrow (E)$	$S \rightarrow 0$		$B \rightarrow b \mid bS \mid aBB$
$E \rightarrow id$	$S \rightarrow 1$		
	$S \rightarrow \epsilon$		

Dervations

- $E \Rightarrow^{E \to E+E} E + E \Rightarrow^{E \to id} id + E \Rightarrow^{E \to id} id + id$
  is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \to E + E$, $E \to id$, and $E \to id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as,
  $E \Rightarrow^* id + id$, and is read as S **derives** $id + id$

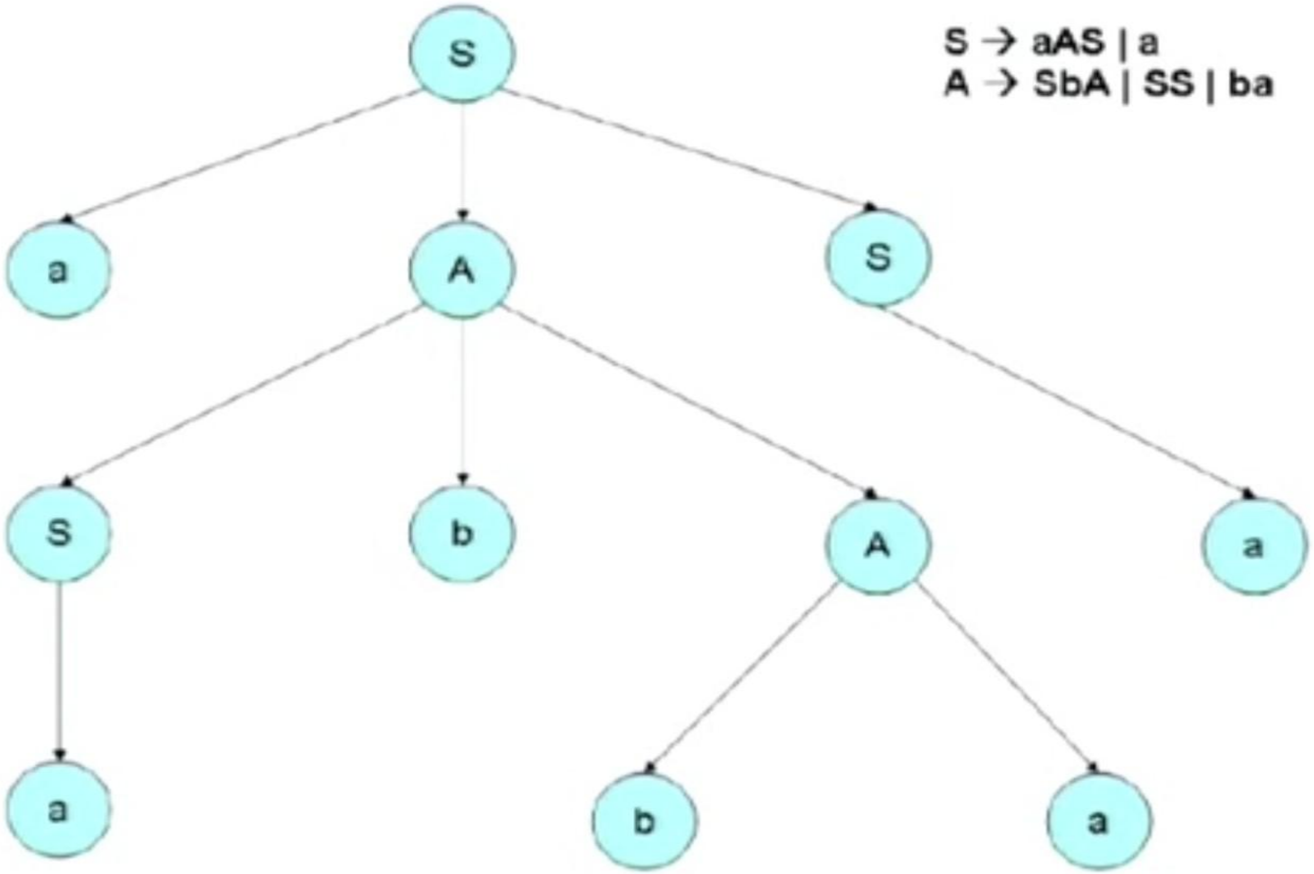
Context Free Languages (CFLs)

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by G*, denoted $L(G)$, is
  $L(G) = \{w \mid w \in T^*, \text{ and } S \Rightarrow^* w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S
- Examples
  1. $L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'
  2. $L(G_2)$ = Set of palindromes over 0 and 1
  3. $L(G_3) = \{a^n b^n \mid n \geq 0\}$
  4. $L(G_4) = \{x \mid x \text{ has equal no. of } a's \text{ and } b's\}$
- A string $\alpha \in (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in P$, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Derivation Tree Example

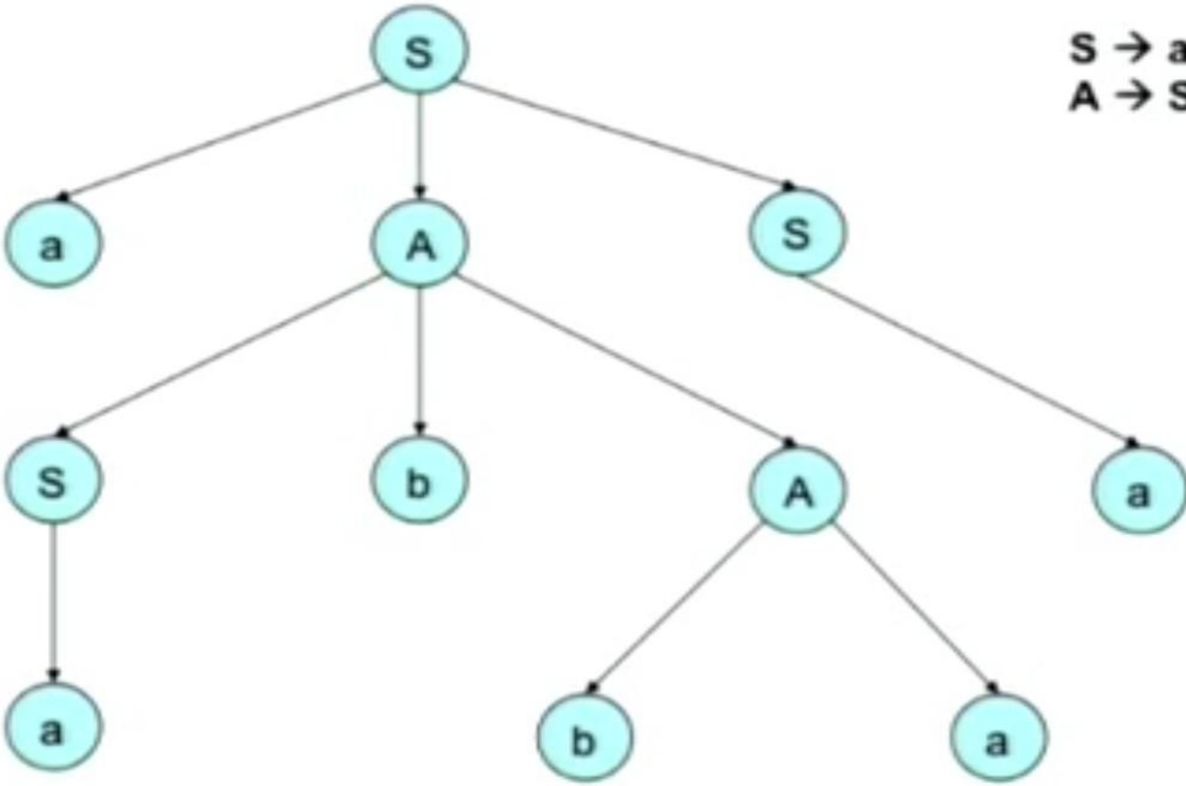


S => aAS => aSbAS => aabAS => aabbaS => aabbaa

Leftmost and Rightmost Derivations

- If at each step in a derivation, a production is applied to the leftmost nonterminal, then the derivation is said to be **leftmost**. Similarly **rightmost derivation**.
- If $w \in L(G)$ for some G, then w has at least one parse tree and corresponding to a parse tree, w has unique leftmost and rightmost derivations
- If some word w in $L(G)$ has two or more parse trees, then G is said to be **ambiguous**
- A CFL for which every G is ambiguous, is said to be an **inherently ambiguous** CFL

Leftmost and Rightmost Derivations: Example



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S => aAS => aSbAS => aabAS => aabbaS => aabbaa**

Rightmost derivation: **S => aAS => aAa => aSbAa => aSbbaa => aabbaa**

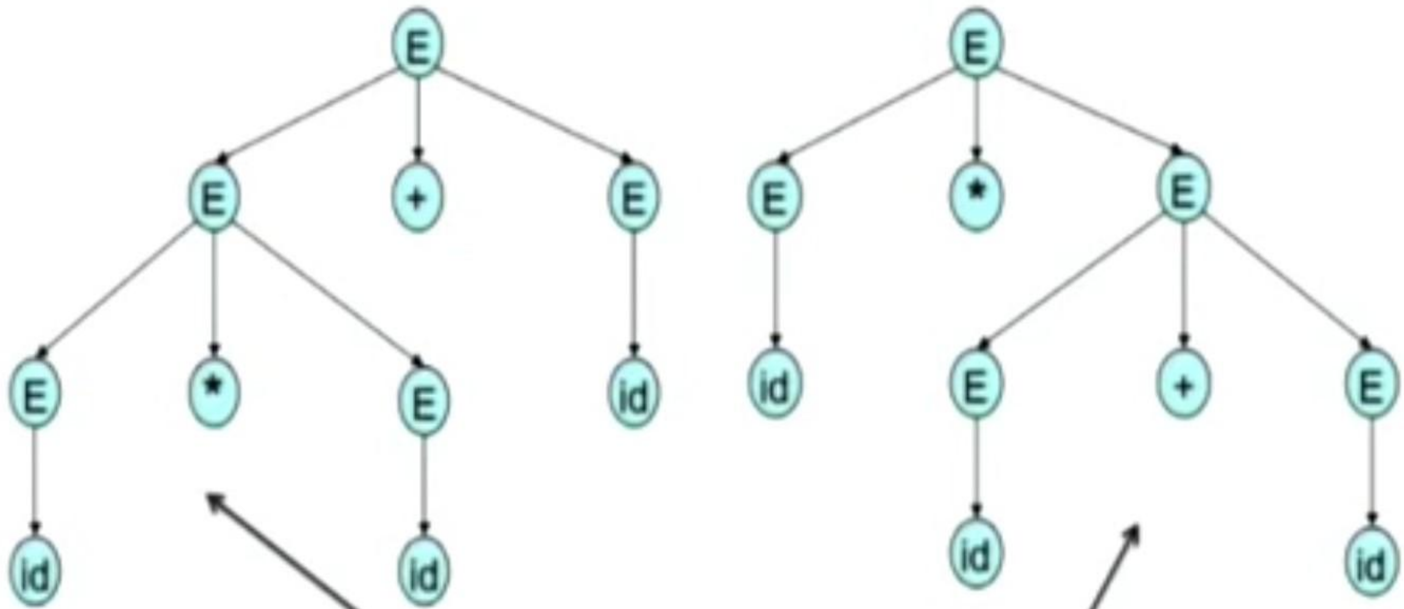
Ambiguous Grammar Examples

- The grammar, $E \to E + E | E * E | (E) | id$
  is ambiguous, but the following grammar for the same language is unambiguous
  $E \to E + T | T,\ T \to T * F | F,\ F \to (E) | id$
- The grammar,
  $stmt \to IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$

  is ambiguous, but the following equivalent grammar is not

  $stmt \to IF\ expr\ stmt | IF\ expr\ matched\_stmt\ ELSE\ stmt$
  $matched\_stmt \to$
  $IF\ expr\ matched\_stmt\ ELSE\ matched\_stmt | other\_stmt$
- The language,
  $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$,
  is inherently ambiguous

Ambiguity Example 1

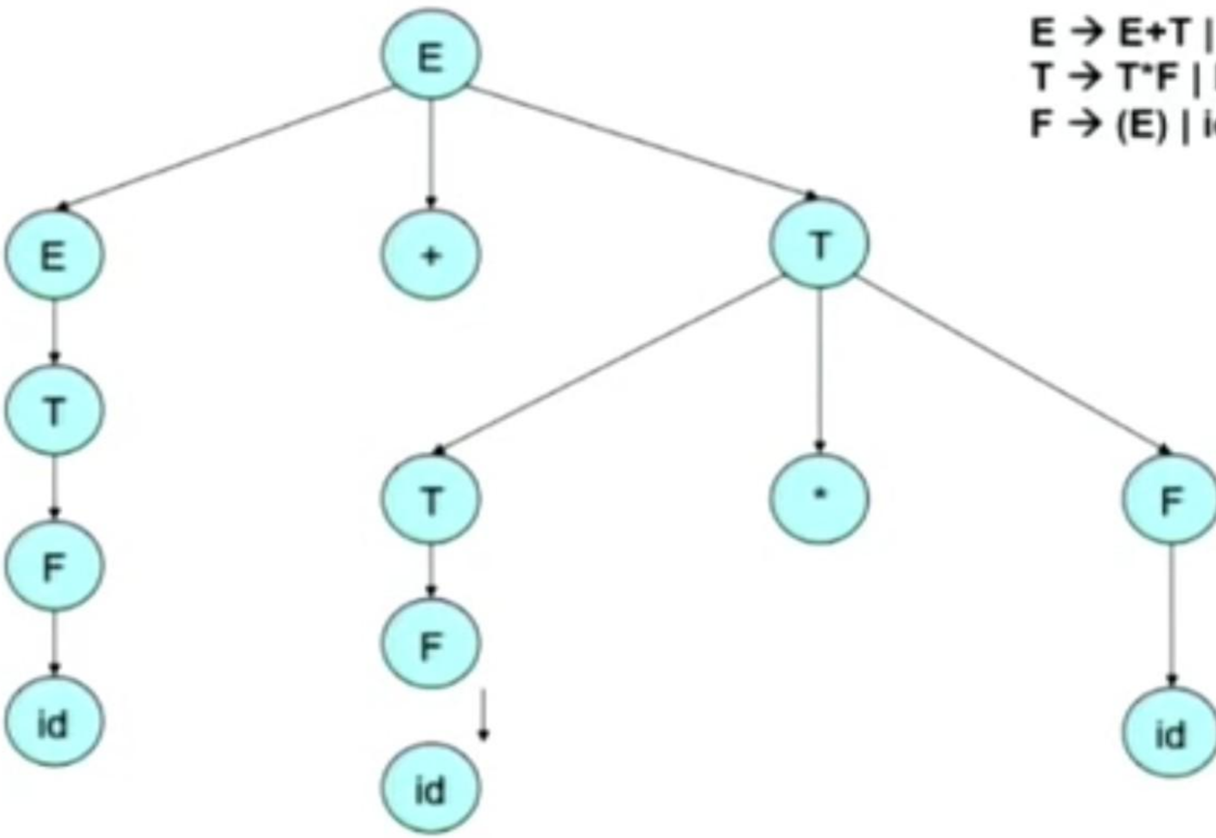


$E \Rightarrow E+E \Rightarrow E*E+E \Rightarrow id*E+E \Rightarrow id*id+E \Rightarrow id*id+id$

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

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

Equivalent Unambiguous Grammar

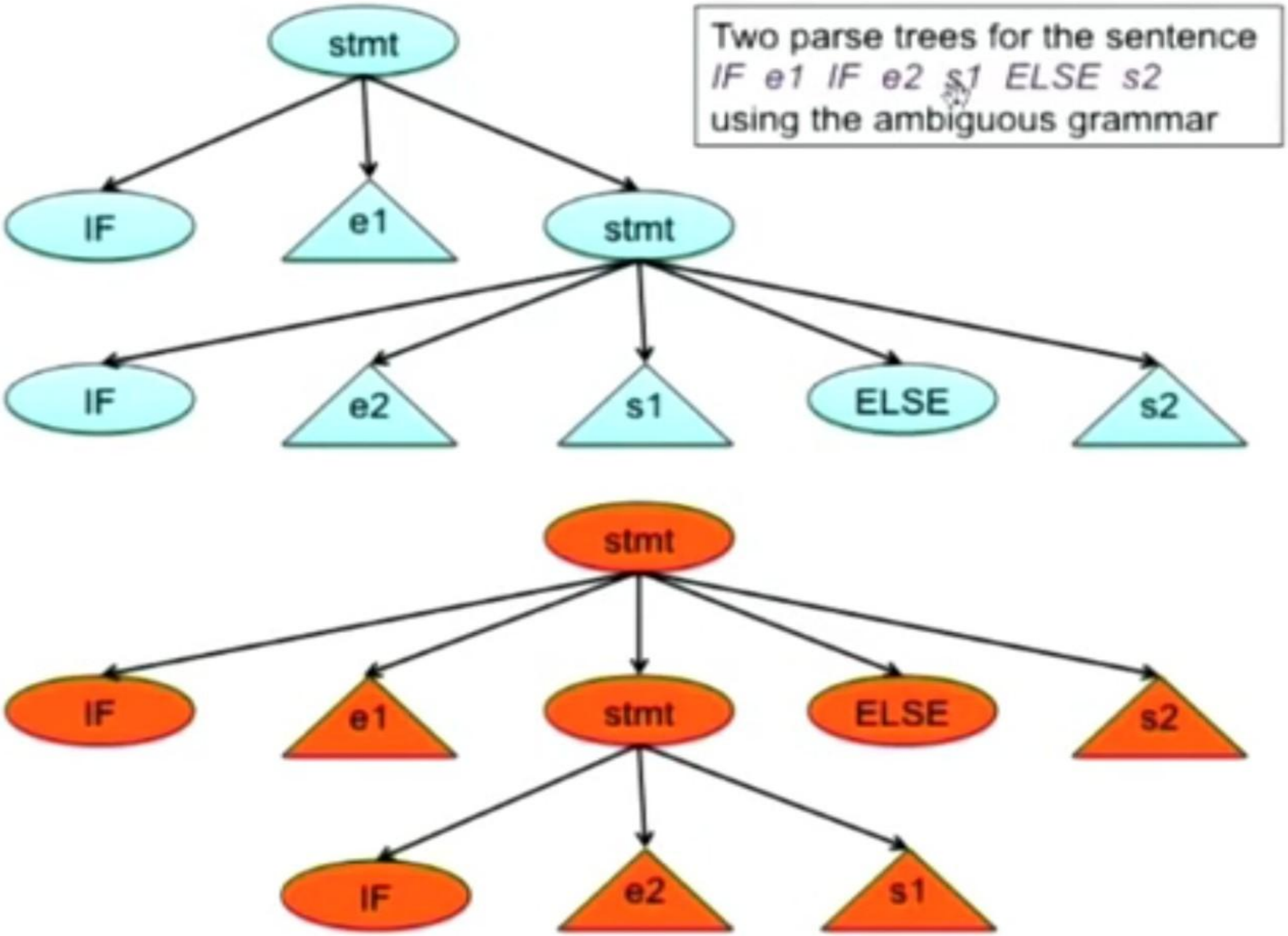


E → E+T | T
T → T*F | F
F → (E) | id

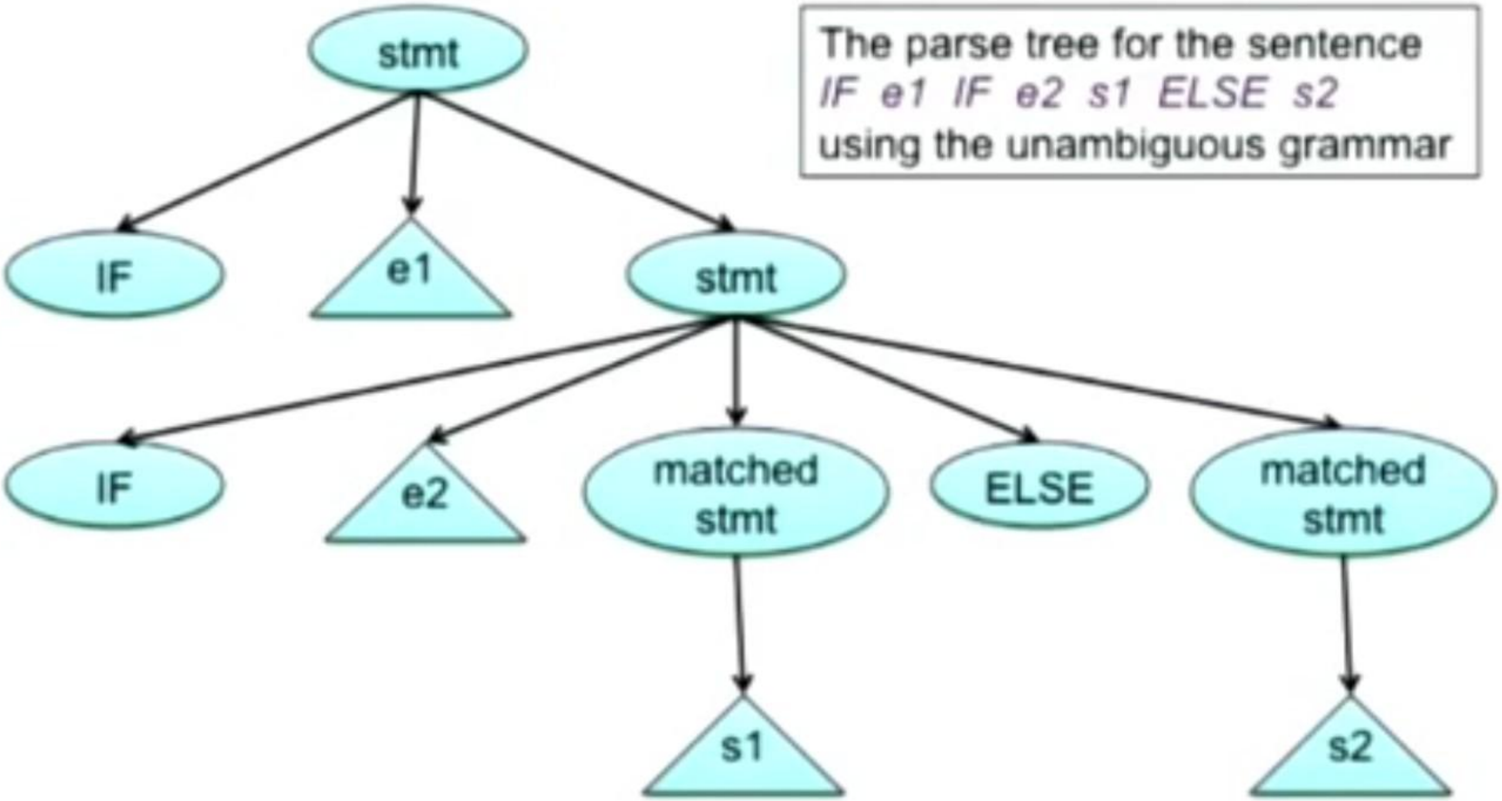
E => E+T => T+T => F+T => id+T => id+T*F => id+F*F => id+id*F => id+id*id

E => T*F => F*F => (E)*F => (E+T)*F => (T+T)*F => (F+T)*F => (id+T)*F
=> (id+F)*id => (id+id)*F => (id+id)*id

Ambiguity Example 2



Ambiguity Example 2



The parse tree for the sentence
IF e1 IF e2 s1 ELSE s2
using the unambiguous grammar

s→ IF e s | IF e ms ELSE s
ms → IF e ms ELSE ms | other_s