Languages and Computation (COMP2049/AE2LAC)

Context-Free Grammars

Dr. Tianxiang Cui

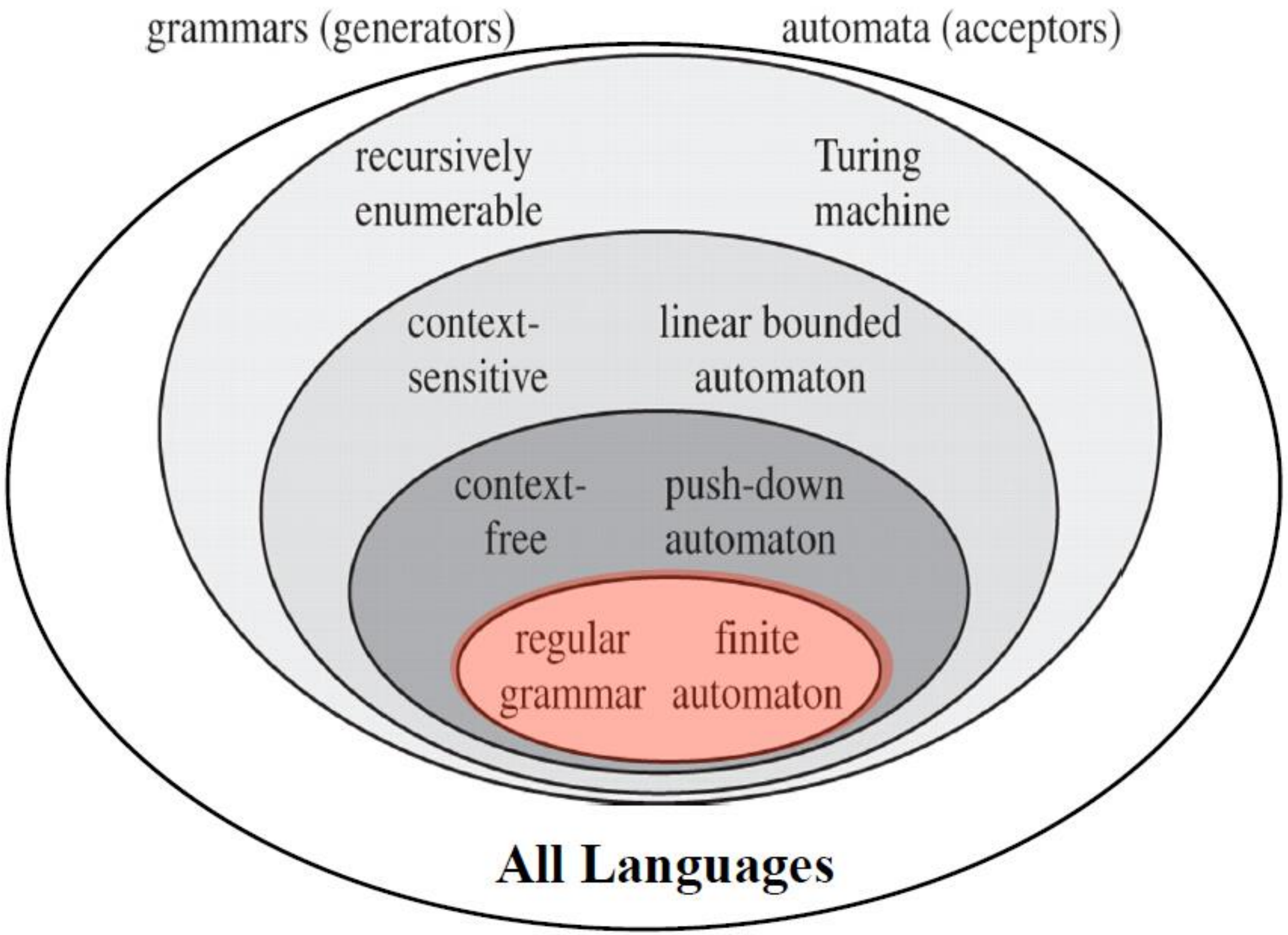
tianxiang.cui@nottingham.edu.cn

Regular Languages: Review

- Regular Languages are exactly the languages that can be accepted by a Finite Automaton
- Relatively simple:
  - Empty set (∅), null string (ε)
  - Union, concatenation, Kleene star
- NFA can represent a regular expression
- NFA can be converted into DFA
  - Eliminate ε-Transitions
  - Subset construction

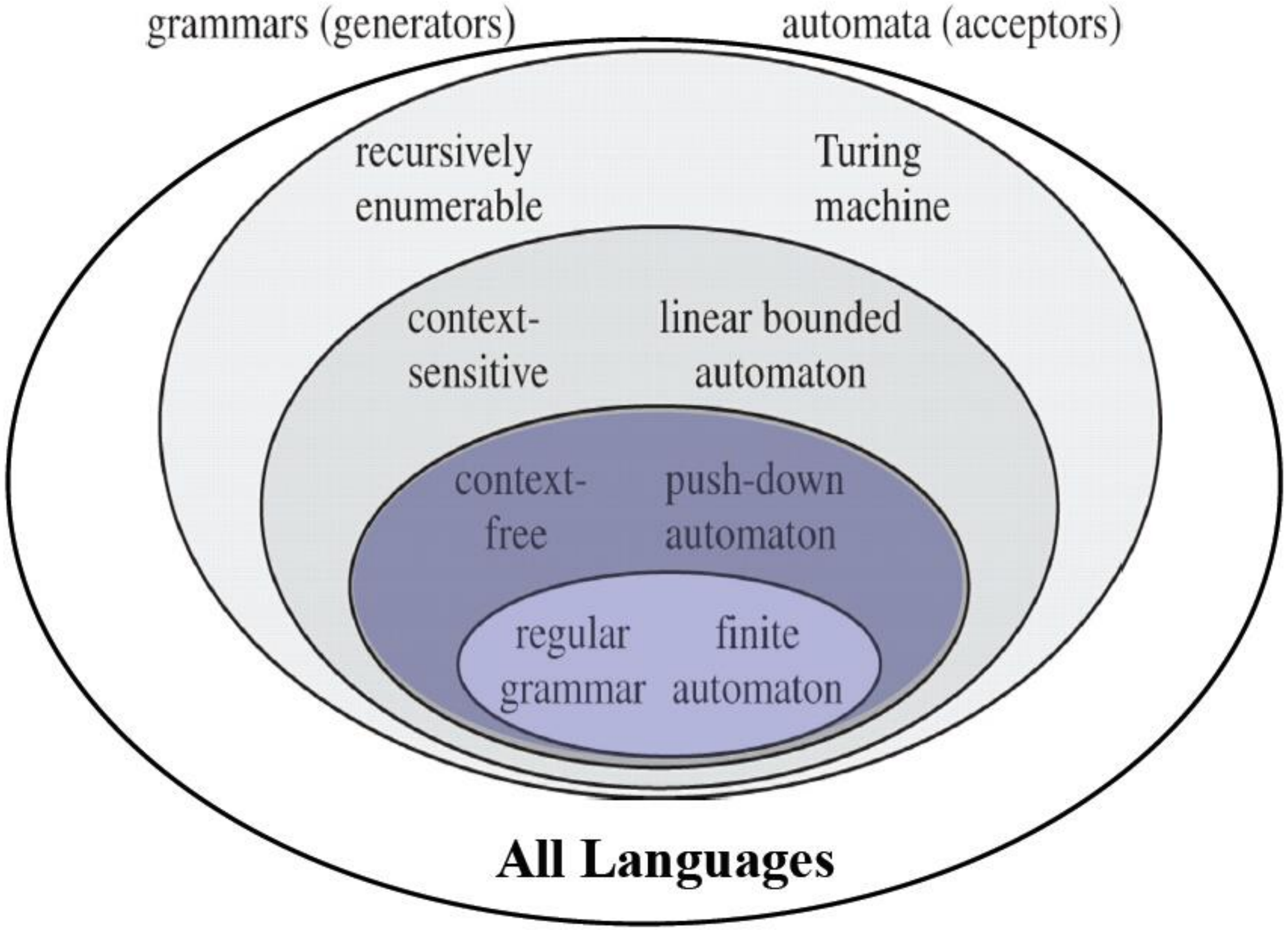
The Chomsky Hierarchy

- We have now finished looking at **regular languages** (Type 3). These are **exactly** the languages accepted by a **finite automaton**



The Chomsky Hierarchy

- Now, we will look at **Context-Free Grammars** (CFG). Regular grammars are a **subset** of CFG



Non-Regular Languages

- We have established that the following language is not regular:

$$L = \{a^n b^n \mid n \geq 0\}$$

- Others? What about B: the language of "balanced parentheses"?

$$()() \in B$$

$$((()())()) \in B$$

$$(() \notin B$$

- Is B regular?
  - Can prove this formally using the Pumping Lemma for regular languages

Non-Regular Languages

- But of course, “balanced parentheses” is a key feature of many important classes of languages; e.g.:
  - Arithmetic expressions: **(,)**
  - Matching keywords in programming languages: **begin**, **end**, **repeat**, **until**
  - Markup languages; e.g. HTML: **<p>, </p>, , **
- Q: Can such languages be described formally? How?
- A: Through Context-free Grammars (CFG)

Grammar Rules

- Regular languages and finite automata are too simple for many purposes
  - Using **context-free grammars** allows us to describe more interesting languages
  - Much high-level programming language syntax can be expressed with context-free grammars
  - Context-free grammars with a very simple form provide another way to describe the regular languages
- Grammars can be **ambiguous**
- We will study how derivations can be related to the structure of the string being derived

Grammar Rules

- A grammar is a set of rules, usually simpler than those of English, by which strings in a language can be generated
- Consider the language $L = \{a^n b^n \mid n \geq 0\}$, defined using the recursive definition:

$$\varepsilon \in L$$

$$\text{For every } S \in L,\ aSb \in L$$

- Think of S as a variable representing an arbitrary element, and write these rules as

$$S \to \varepsilon$$

$$S \to aSb$$

- In the process of obtaining an element of L, S can be replaced by either string

Grammar Rules

- If α and β are strings, and α contains at least one occurrence of S, then $\alpha \Rightarrow \beta$ means that β is obtained from α in one step, by using **one of the two rules** to **replace** a single occurrence of S by either ε or aSb
- For example, to describe a derivation of the string *aaabbb*, we could write:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

- We can simplify the rules by using the "**|**" symbol to mean "**or**", so that the rules become

$$S \rightarrow \varepsilon \mid aSb$$

Context-Free Grammars

- CFGs originated as an attempt to describe grammars for natural languages like English
- Key idea:
- Rules, called **productions**, that describe how symbols called **nonterminals**, can be replaced by nonterminals and **terminals** until only terminals left

$$nonterminal \rightarrow terminals\ and\ nonterminals$$

Context-Free Grammars

- **Definition**
- A context-free grammar (CFG) is a 4-tuple $G=(N, T, S, P)$, where
  - N is a finite set of **nonterminals**
    - Or **variables**, and consequently sometimes it is denoted by V
  - T is a finite set of **terminals**
    - The terminals are the alphabet of the language defined by a context-free grammar, and for that reason the set of terminals is sometimes denoted by Σ
  - $N \cap T = \varnothing$ (N and T are disjoint finite sets)
  - $S \in N$ is the **start symbol**
  - P is a finite set of **productions** (or **grammar rules**) of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$

Context-Free Grammars

- We use $\rightarrow$ for productions in a grammar and $\Rightarrow$ for a step in a derivation
- The notations $\alpha \Rightarrow^n \beta$ and $\alpha \Rightarrow^* \beta$ refer to n steps and zero or more steps, respectively
- We will sometimes write $\Rightarrow_G$ to indicate a derivation in a particular grammar G
- $\alpha \Rightarrow \beta$ means that there are strings α_1, α_2 and γ in $(N \cup T)^*$ and a production $A \rightarrow \gamma$ in P such that $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$
  - This is a single step in a derivation
- What makes the grammar **context-free** is that the production $A \rightarrow \gamma$, with left side A, can be applied wherever A occurs in the string (independent of the context, i.e., regardless of what α_1 and α_2 are)

Context-Free Languages

- **Definition**
- If G=(N, T, S, P) is a CFG, the language generated by G is

$$L(G) = \{ x \in T^* \mid S \Rightarrow_G^* x\}$$

- S is the start variable, and x is a string of terminals
- A language L is a **context-free language** (CFL) if there is a CFG G with $L = L(G)$

CFG Example

- Consider $AEqB = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$
- Let's develop a CFG for $AEqB$
- If x is a non-null string in $AEqB$ then either $x = ay$, where $y \in L_b = \{z \mid n_b(z) = n_a(z) + 1\}$, or $x = by$, where $y \in L_a = \{z \mid n_a(z) = n_b(z) + 1\}$
- If we represent L_b by the variable B and L_a by the variable A
- The productions so far are $S \rightarrow \varepsilon \mid aB \mid bA$
- We need to know the productions for A and B

CFG Example

- Now consider a string $y \in L_a = \{z \mid n_a(z) = n_b(z) + 1\}$
- If y starts with a, then the remainder is a member of $AEqB$

$$S \rightarrow \varepsilon \mid aB \mid bA$$

$$A \rightarrow aS$$

- If y starts with b, the rest has two more a's than b's
  - A string containing two more a's than b's must be the concatenation of two strings, each with one more a

$$A \rightarrow bAA$$

- Similar for L_b, the resulting grammar would be

$$S \rightarrow \varepsilon \mid aB \mid bA$$

$$A \rightarrow aS \mid bAA$$

$$B \rightarrow bS \mid aBB$$

Context-Free Grammars: More

- **Theorem**: If L_1 and L_2 are CFLs over $\sum$, then so are $L_1 \cup L_2$, $L_1 L_2$, and $L_1{}^*$
- Suppose G_1 and G_2 are CFGs that generate L_1 and L_2 respectively, and assume that they have no variables in common
- Suppose that S_1 and S_2 are the start variables. S_u, S_c and S_k , the start variables of the new grammars, will be new variables

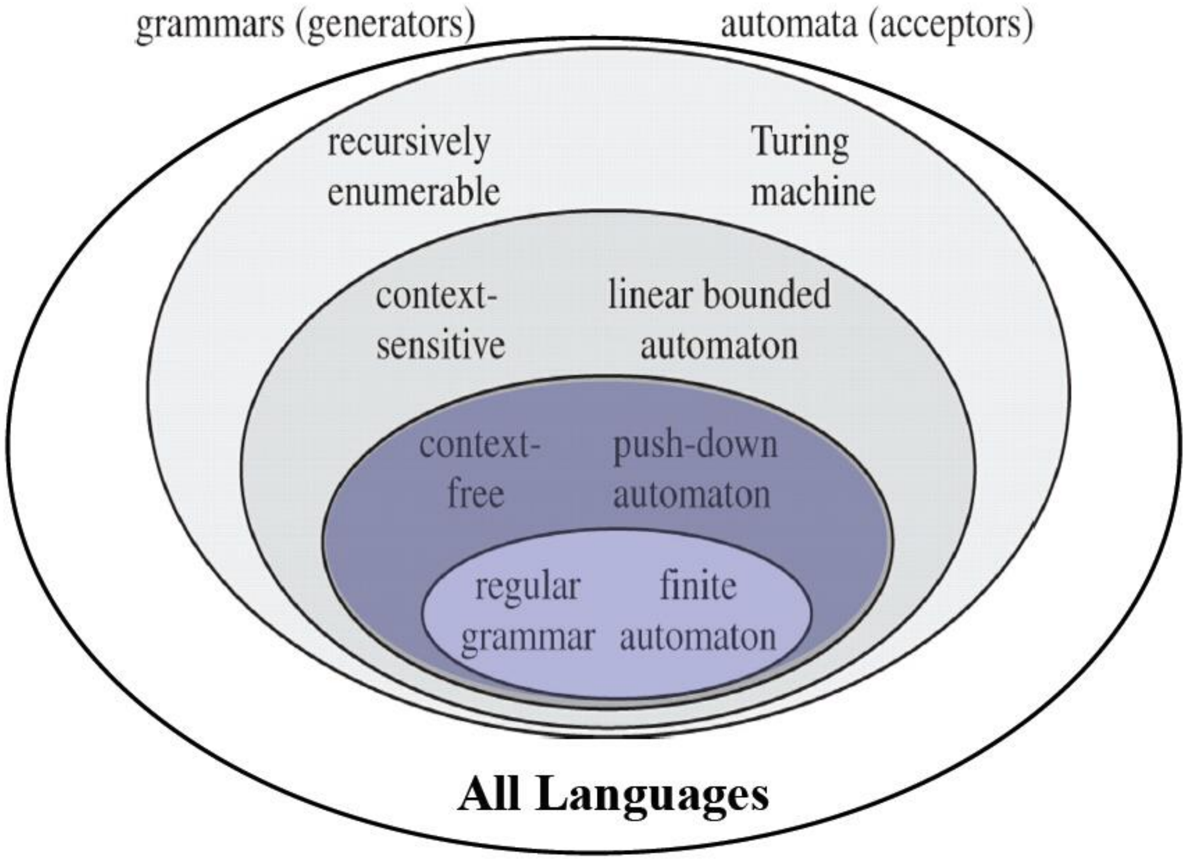
G_u just adds the rules $S_u \to S_1 \mid S_2$ to G_1 and G_2

G_c just adds the rule $S_c \to S_1 S_2$ to G_1 and G_2

G_k just adds the rules $S_k \to \varepsilon \mid S_k S_1$ to G_1

Regular Grammar

- The three operations previous theorem are the ones involved in the recursive definition of regular languages
- In fact, every regular language over $\sum$ is a CFL
- But, we also know that some languages are CFL but not regular
  - So CFL is a strict **superset** of the regular language



Derivation Tree

- $\alpha \Rightarrow^* \beta$ means that it is possible to get from α to β using a sequence of productions
- A derivation of β is the sequence of steps that gets to β, and it can be drawn as a derivation tree
- In a derivation tree
  - The root is the start variable
  - All internal nodes are labeled with nonterminals (variables)
  - All leaves are labeled with terminals (alphabets)
  - All internal node and its children represent a production used in the derivation
  - The string derived is read off from left to right, ignoring ε's

Derivation Tree: Example

- Given a CFG G=(N, T, S, P), where N = {E}, T = {+, *, (,), id}, S = E and P is given by

$$E \rightarrow E + E$$

$$\mid\ E * E$$

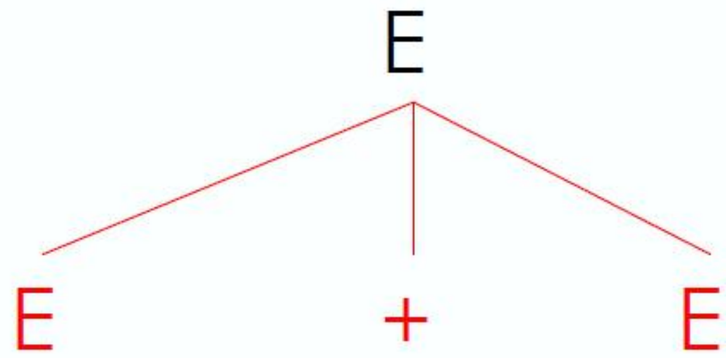
$$\mid\ (E)$$

$$\mid\ id$$

- Draw the derivation tree for the string id*id+id

Derivation Tree: Example

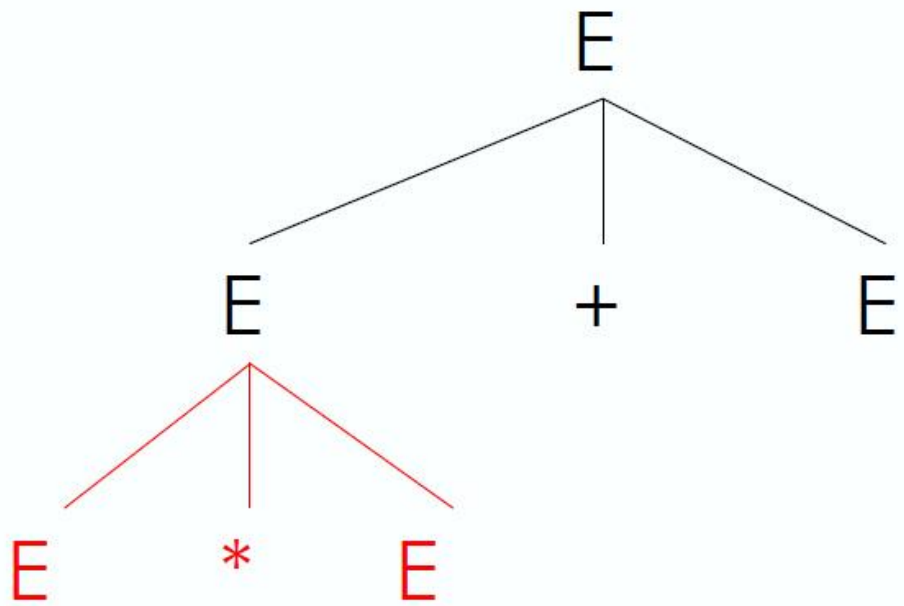
Derivation Tree: Example



$$
\begin{aligned}
& E \\
\rightarrow\ & E+E
\end{aligned}
$$

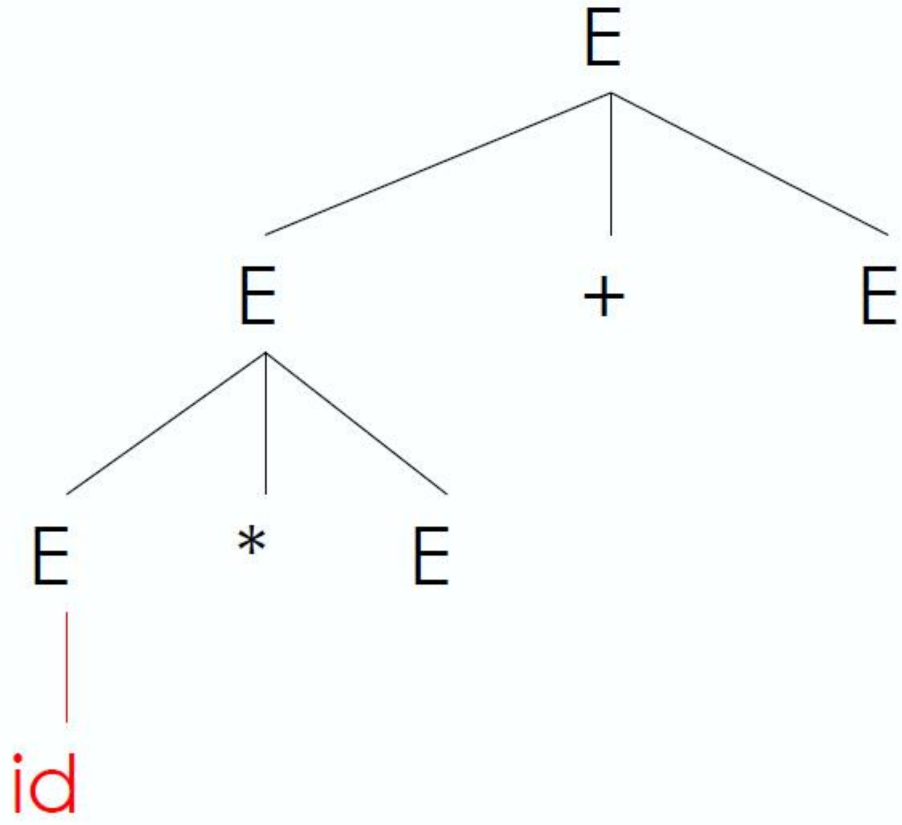
Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E
\end{aligned}
$$



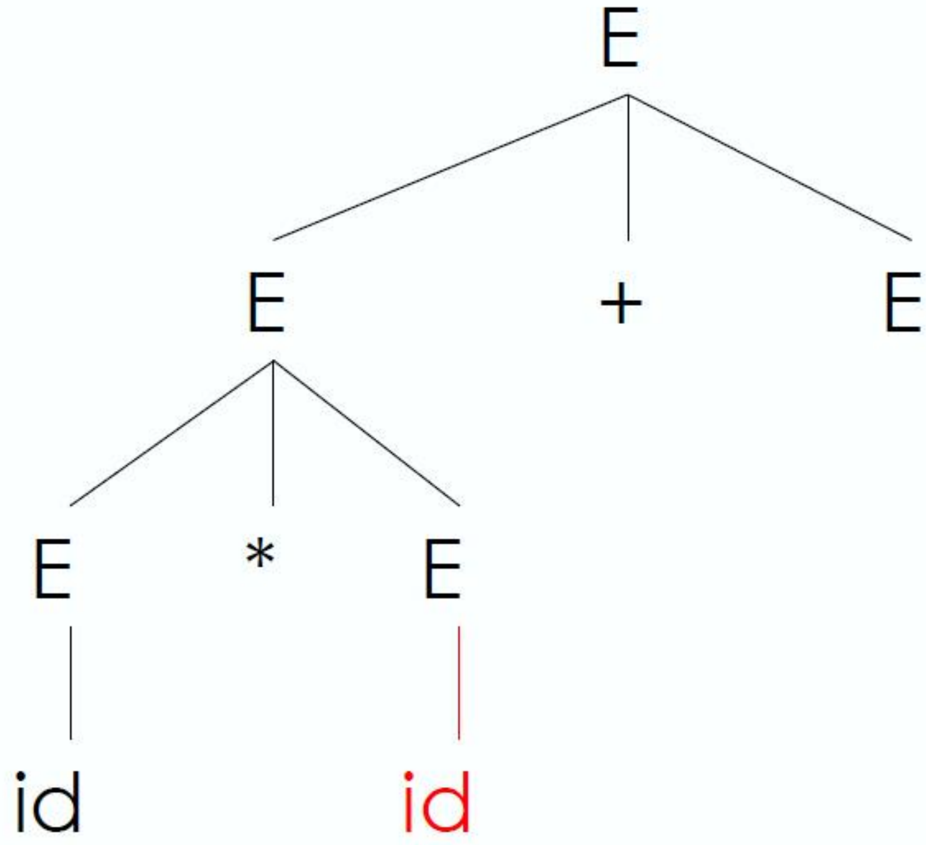
Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E
\end{aligned}
$$



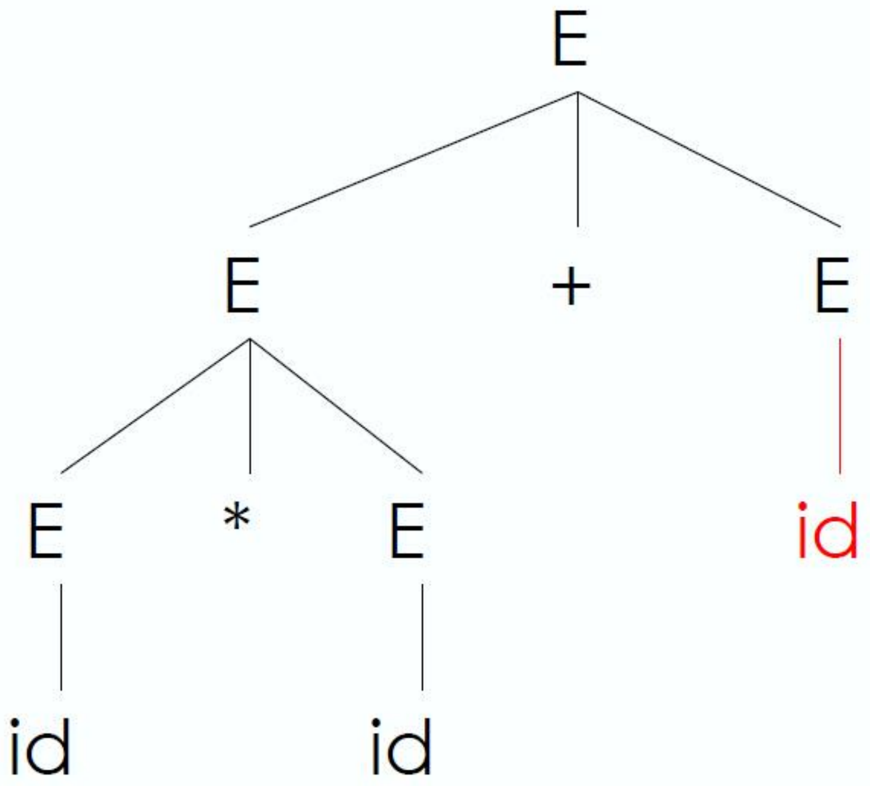
Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E
\end{aligned}
$$



Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$

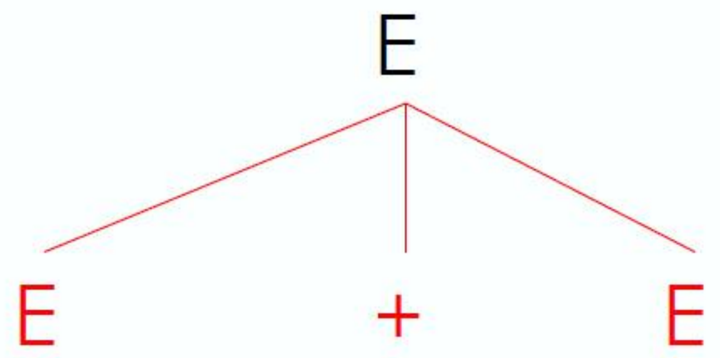


Notes on Derivation Tree

- A derivation tree has
  - **Terminals** at the leaves
  - **Non-terminals** at the internal nodes
- An in-order traversal of the leaves is the original input
- The previous example is a **leftmost** derivation
  - At each step, replace the **left-most** non-terminal
- There is an equivalent notion of a **rightmost** derivation
  - At each step, replace the **right-most** non-terminal

Derivation Tree: Example

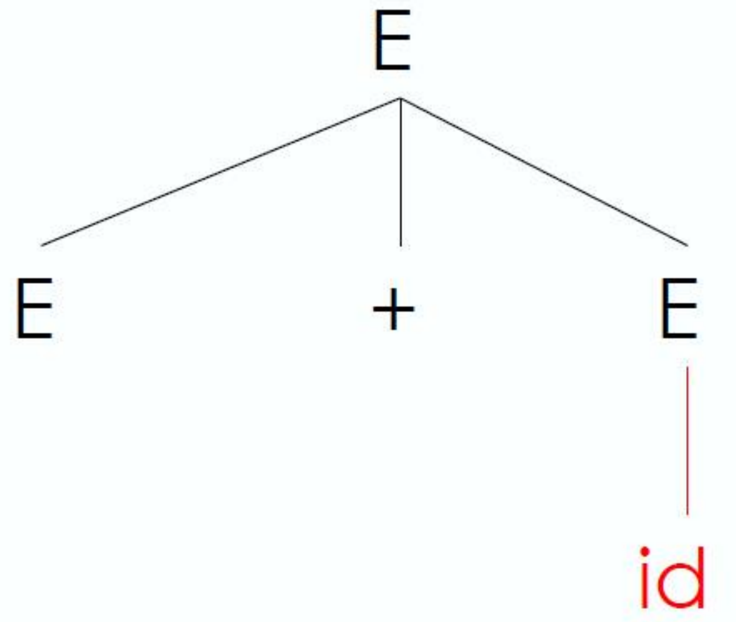
Derivation Tree: Example



$$
\begin{aligned}
& E \\
\rightarrow \quad & E{+}E
\end{aligned}
$$

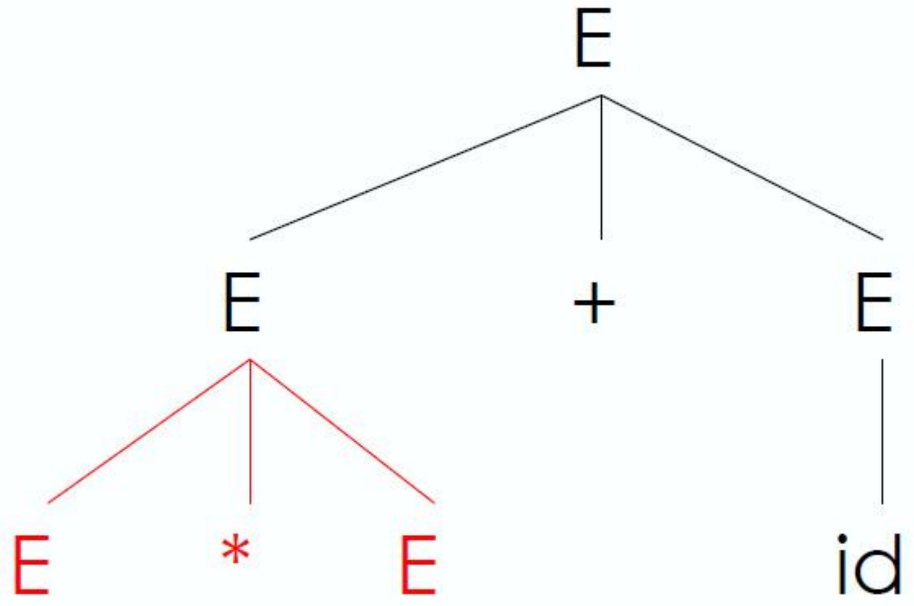
Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id
\end{aligned}
$$



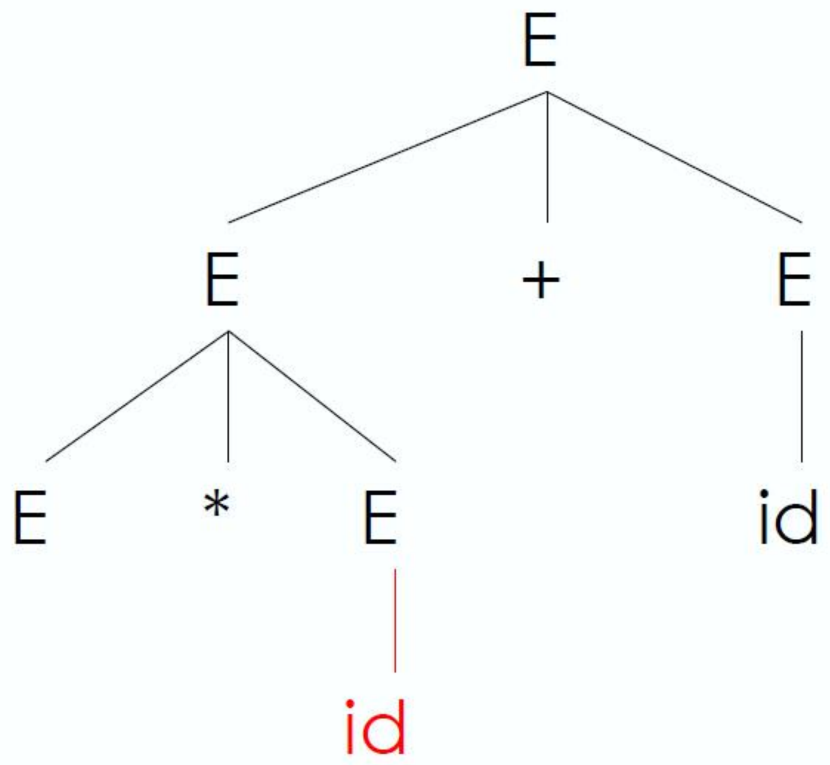
Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id
\end{aligned}
$$



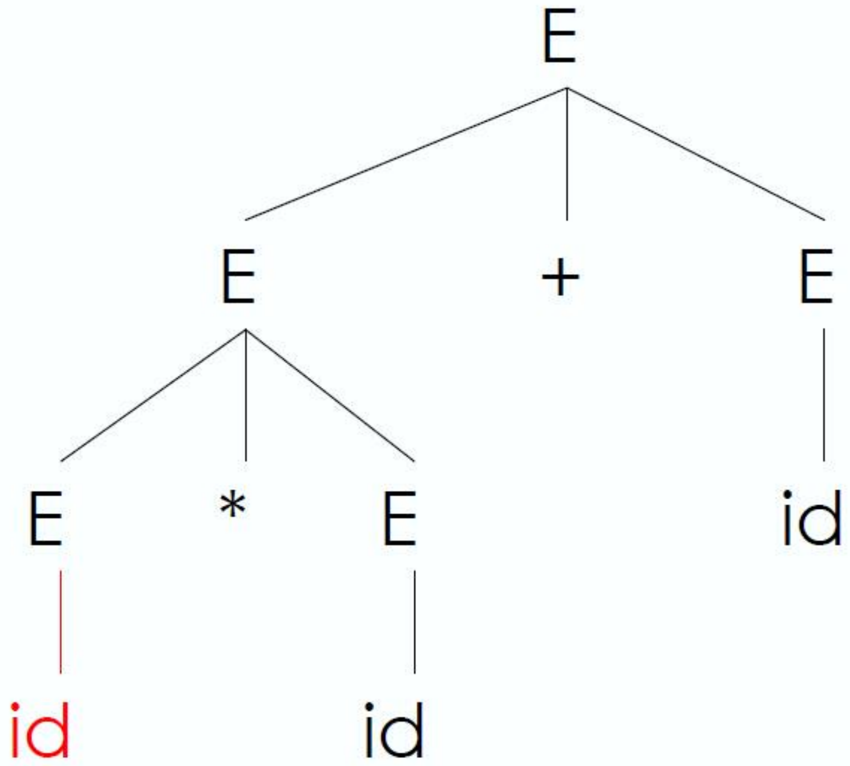
Derivation Tree: Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id
\end{aligned}
$$



Derivation Tree: Example

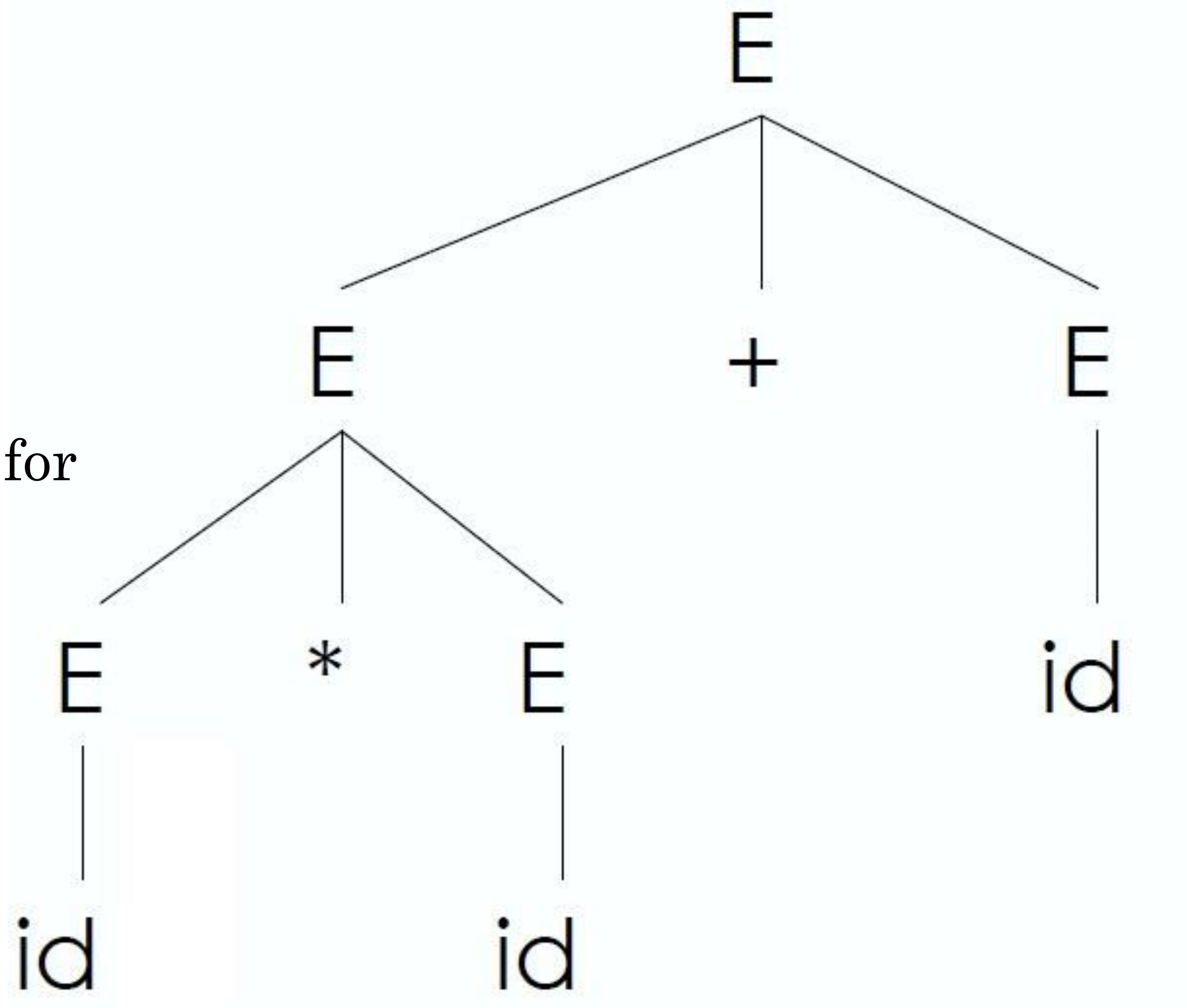
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Back to The Example

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

- We've seen the derivation tree for the string $id*id+id$
- Any other possibilities?



Ambiguity

- A CFG G=(N, T, S, P) is ambiguous if it has more than one derivation tree for some string $x \in L(G)$
- Equivalently, G is ambiguous if there are
  - More than one **left-most derivations**, or
  - More than one **right-most derivations**

  for some string $x \in L(G)$
- Ambiguity can be problematic, as the structure of a derivation tree often is used to suggest a meaning for the word
- Ambiguity is common in programming languages

Ambiguity: Dangling *else*

- In the C programming language, an if-statement can be defined using the following grammar rules:

$$S \rightarrow if\ \ (E)\ \ S$$

$$\mid\ if\ \ (E)\ \ else\ S$$

$$\mid\ OS$$

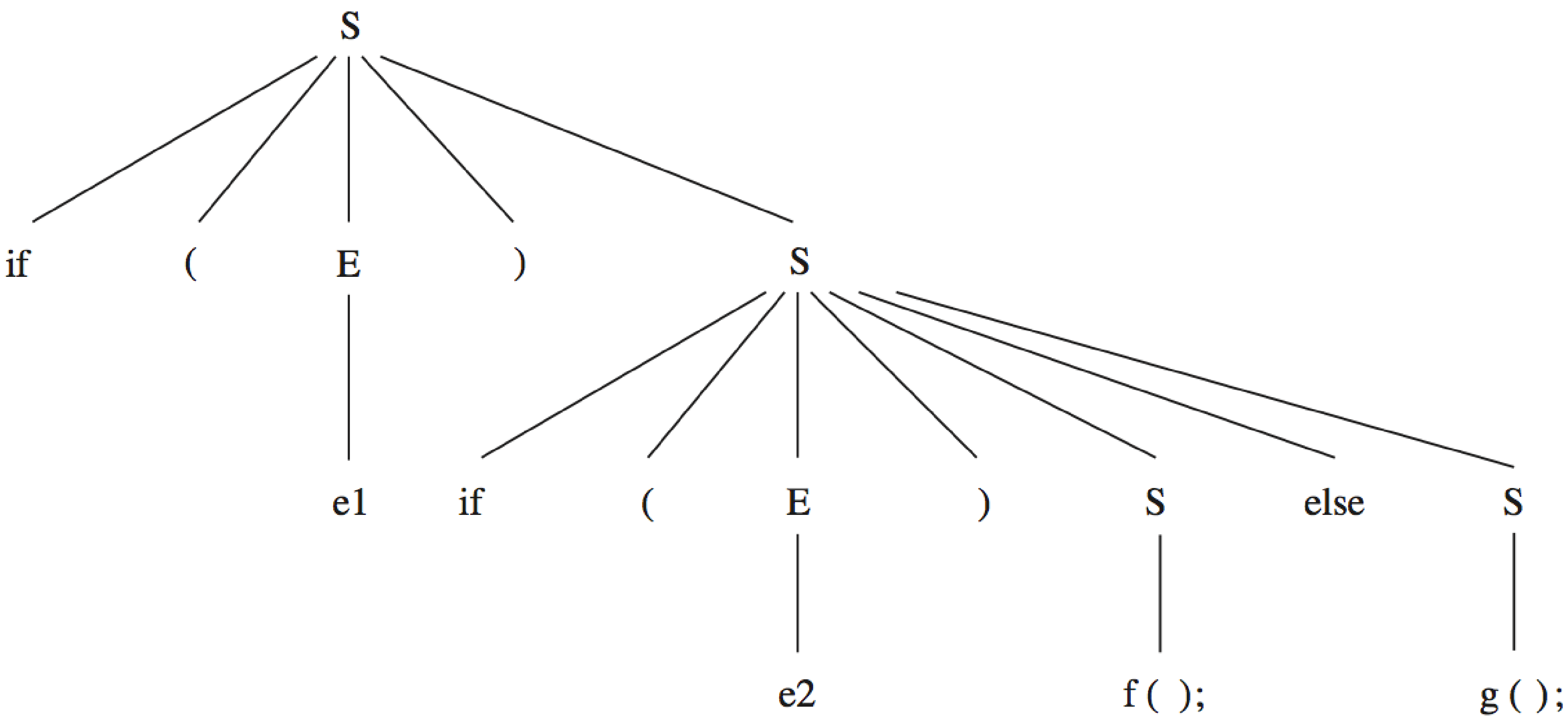
- E is short for <expression>, S is short for <statement> and OS is short for <otherstatement>
- Consider the statement in C:

```
if (e1) if (e2) f(); else g();
```

Ambiguity: Dangling *else*

- "**Correct**" interpretation:

```
if (e1) {if (e2) f(); else g();}
```

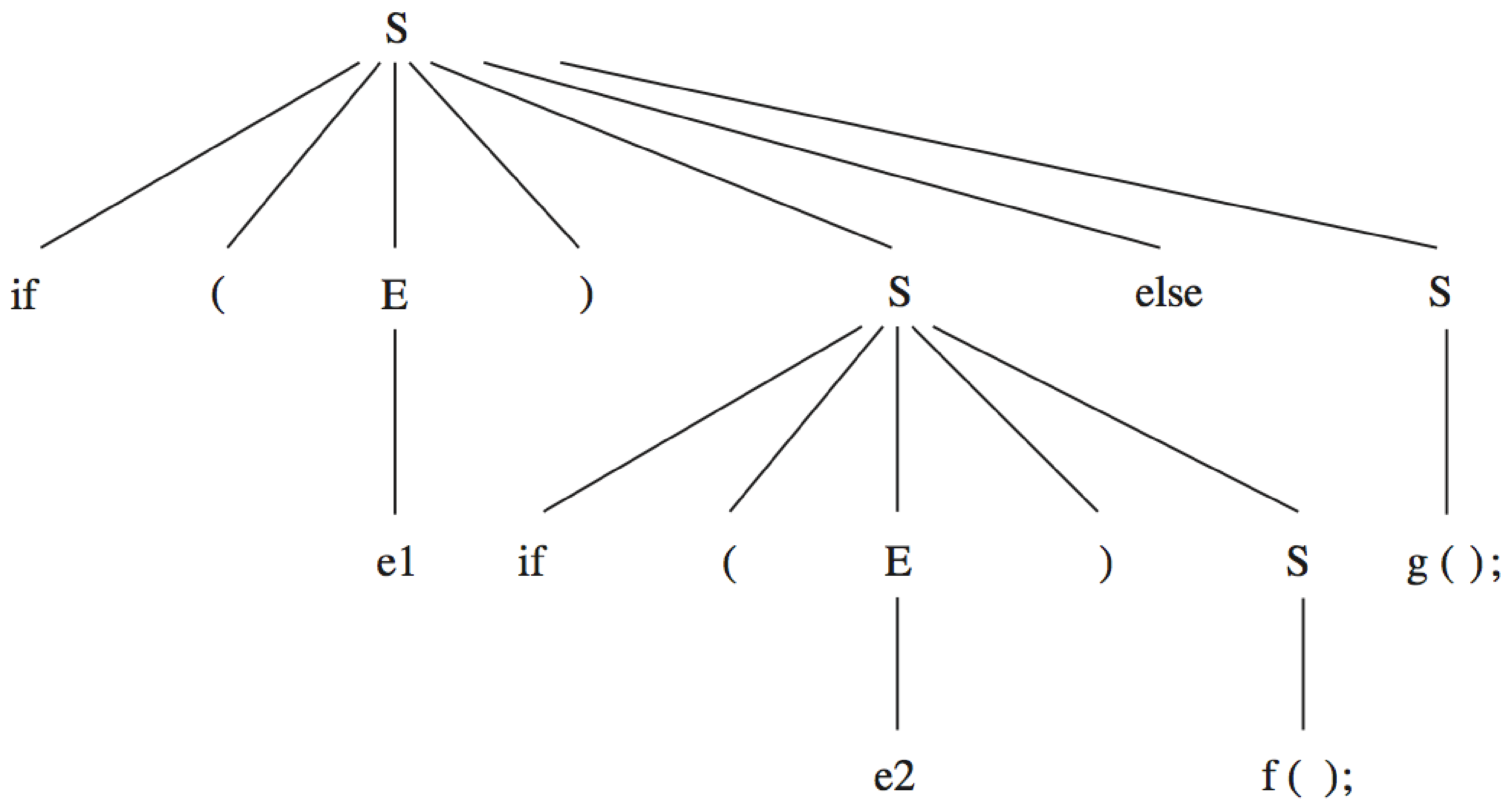


(*a*)

Ambiguity: Dangling *else*

- "**Incorrect**" interpretation:

```
if (e1) {if (e2) f();} else g();
```



(*b*)

Dealing With Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously
- Use our first example, the grammar

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

can be rewritten as

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

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

- Now the string $id*id+id$ may only have **one** derivation tree

Dangling *else*: A Fix

- The grammar for if-statement is also ambiguous:

$$S \rightarrow if\ \ (E)\ \ S \mid if\ \ (E)\ \ else\ S\ \mid OS$$

- We can rewrite it as

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow if\ \ (E)\ S_1\ else\ S_1 \mid OS$$

$$S_2 \rightarrow if\ \ (E)\ S \mid$$

$$if\ \ (E)\ S_1\ else\ S_2$$

- S_1 represents the matched *if* (*if-else*)
- S_2 contains at least one unmatched *if*, and *else* in S_2 matches the closest unmatched *if*

Disambiguating Grammars

- Given an ambiguous grammar G, it is often possible to construct an **equivalent** grammar G' (i.e., $L(G) = L(G')$), such that G' is **unambiguous**
- Some languages are **inherently ambiguous** CFLs, meaning that every CFG generating the language necessarily is ambiguous
- However, there are no general techniques for handling ambiguity, and it is impossible to convert automatically an ambiguous grammar to an unambiguous one
- Not every ambiguous grammar can be turned into an equivalent unambiguous one
- But, there are some possible techniques to disambiguate expression grammars
  - Operator precedence
  - Associativity

Operator Precedence

- Consider the grammar

$$E \to E + E \mid E \text{ - } E \mid E * E \mid E \,/\, E \mid D$$

$$D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

- Let's try 1+2*3
- **Stratify** the grammar so that operators having **higher** precedence occur as **subexpressions** of expressions involving operators of lower precedence
  - i.e., arrange in "layers" with higher precedence at lower layer
- We know that "*****", "**/**", have higher precedence than "**+**", "**-**", therefore the grammar can be rewritten as

$$E \to E + E \mid E \text{ - } E \mid E_2$$

$$E_2 \to E_2 * E_2 \mid E_2 / E_2 \mid D$$

$$D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

Operator Associativity

- Now, our new production rule:

$$E \rightarrow E + E \mid E - E \mid E_2$$

$$E_2 \rightarrow E_2 * E_2 \mid E_2 / E_2 \mid D$$

$$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

- Try to draw the parse tree for 1-2-3
- Operators may be associative
  - Left-associative (meaning the operations are grouped from the left, e.g. arithmetic operators)
  - Right-associative (meaning the operations are grouped from the right e.g. exponential operator)
  - Non-associative (meaning operations cannot be chained, e.g. comparison operators)

Operator Associativity

- Refine the grammar so that it enforces operator associativity
- Productions for left-associative operators should be left-recursive

$$E \rightarrow E \otimes E_2$$

- Productions for right-associative operators should be right-recursive

$$E \rightarrow E_2 \otimes E$$

- Productions for non-associative operators should not be immediately recursive

$$E \rightarrow E_2 \otimes E_2$$

Operator Associativity

- Back to our grammar:

$$E \rightarrow E + E \mid E - E \mid E_2$$

$$E_2 \rightarrow E_2 * E_2 \mid E_2 / E_2 \mid D$$

$$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

- Make all operators **left-associative**

$$E \rightarrow E + E_2 \mid E - E_2 \mid E_2$$

$$E_2 \rightarrow E_2 * D \mid E_2 / D \mid D$$

$$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

- Let's try 1-2-3 again

Allowing Explicit Grouping

- Moreover, we want to allow parentheses for grouping
- Parentheses have **higher precedence** than anything else and should thus only be allowed as sub-expressions of expressions involving operators of lowest precedence

$$E \rightarrow E + E_2 \mid E - E_2 \mid E_2$$

$$E_2 \rightarrow E_2 * E_3 \mid E_2 / E_3 \mid E_3$$

$$E_3 \rightarrow (E) \mid D$$

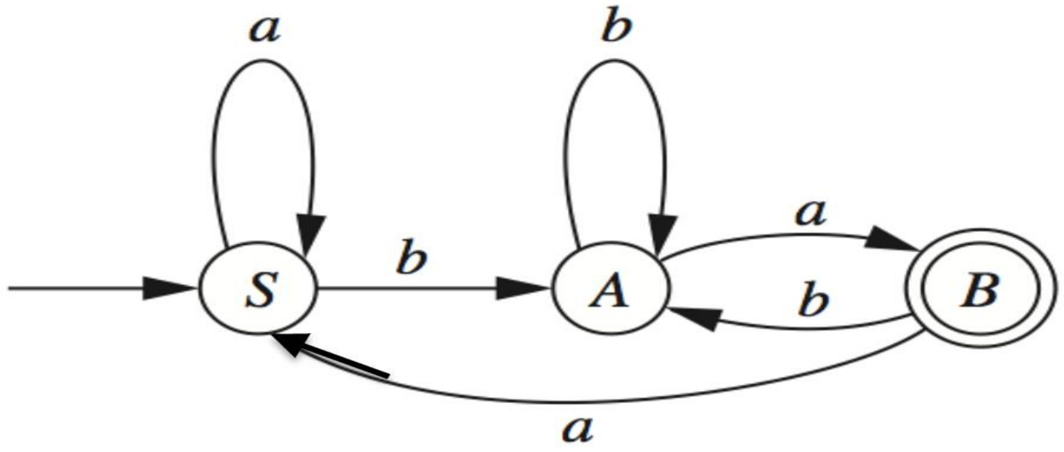
$$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

- Let's try to draw the derivation tree for 1-2*(3-4)-5

CFG For DFA

- Regular grammars are a strict **subset** of CFG
- Every regular language can be generated by a CFG (of a particularly simple form)
- If we have a DFA, then the corresponding CFG can be generated by transforming:
  - States in the DFA correspond to variables in the grammar
  - And transitions of form $\delta(P,s) = Q$ will have the production rule $P \rightarrow sQ$
  - The accepting (final) state will have a null-production in addition

CFG For DFA: Example



- States correspond to variables, transitions of form $\delta(P,s) = Q$ will have the production rule $P \rightarrow sQ$

$$S \rightarrow aS \mid bA \quad A \rightarrow aB \mid bA \quad B \rightarrow aS \mid bA \mid \varepsilon$$

- Let's try the input string *bbaaba*

Regular Grammar

- **Definition**
- A context-free grammar is regular if every production is of the form

$$A \rightarrow sB \text{ or } A \rightarrow \varepsilon$$

$$\text{Where } A, B \in N \text{ and } s \in \sum$$

Chomsky Normal Form

- If we don't find a derivation that produces a string x, how long should we keep trying?
- If grammar G has:
  - **No** ε-productions (of the form $A \to \varepsilon$)
  - And **no** unit-productions (of the form $A \to B$)
  - Then no derivation of a string x can take more than $2|x| - 1$ steps (see pp 149 of textbook for details)
- We could then, in principle, determine whether x can be derived by considering derivations no longer than this
  - If we try all derivations with this many steps and don't find one that generates x, **we may conclude that x is not in the language $L(G)$**

Chomsky Normal Form

- **Definition**
- A CFG is said to be in **Chomsky normal form** if every production is of one of these two types:

$$A \rightarrow BC \ \text{ (where } B \text{ and } C \text{ are variables)}$$

$$A \rightarrow s \ \text{ (where s is a terminal)}$$

Concluding Remarks

- Context-free grammars
  - Nonterminal, terminal, start symbol, production
- Derivation tree
  - Left-most
  - Right-most
- Ambiguity
- Deal with ambiguity
  - Operator precedence
  - Associativity
- Chomsky normal form