Automata and Languages

Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

Today’s Topics

- Context Free Grammar
- Parsing
- Grammar Ambiguity
- Simple Grammar
- Normal Forms definition

CFG: Parsing

Recognition of strings in a language

CFG: Parsing

•Generative aspect of CFG: By now it should be clear how, from a CFG G, you can derive strings $w \in L(G)$.

•Analytical aspect: Given a CFG G and a string w, how do you decide if $w \in L(G)$ and –if so– how do you determine the derivation tree or the sequence of production rules that produce w? This is called the problem of **parsing**.

CFG: Parsing

- Parser

  Is a program that determines if a string $\omega \in L(G)$ by constructing a derivation. Equivalently, it searches the graph of *G*.

  – Top-down parsers
    - Constructs the derivation tree from root to leaves.
    - Leftmost derivation.

  – Bottom-up parsers
    - Constructs the derivation tree from leaves to root.
    - Rightmost derivation in *reverse*.

Parse trees (=Derivation Tree)

A parse tree is a graphical representation of a derivation sequence of a sentential form.

Tree nodes represent symbols of the grammar (nonterminals or terminals) and tree edges represent derivation steps.

CFG: Parsing

Parse Tree: Example

Given the following grammar:

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

Is the string -(**id** + **id**) a sentence in this grammar?

Yes because there is the following derivation:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

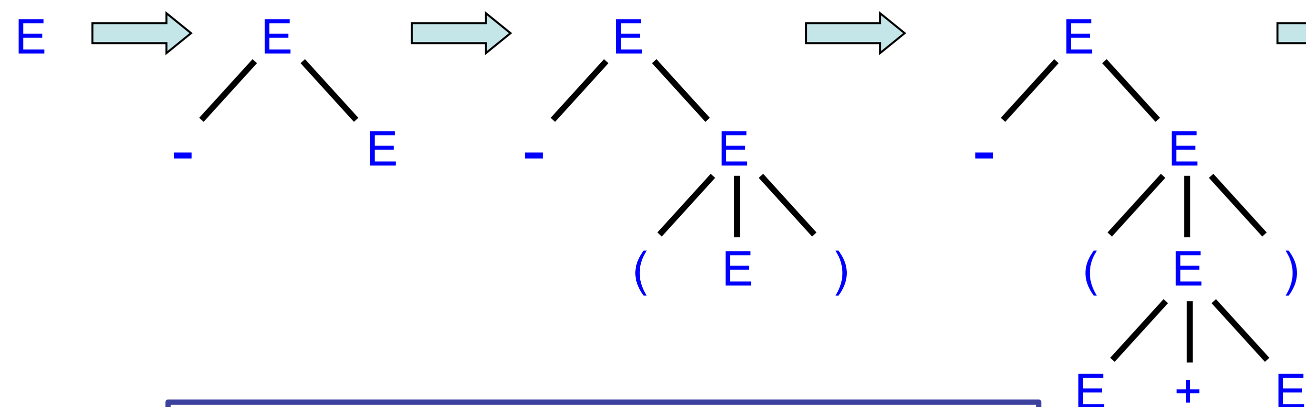
CFG: Parsing

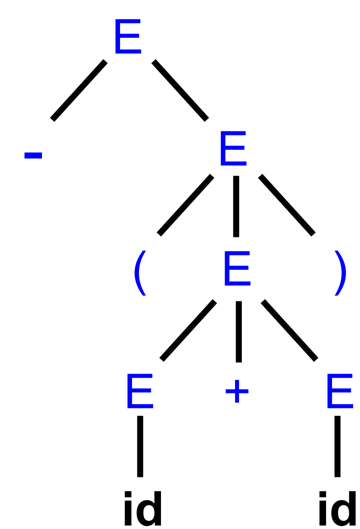
Parse Tree: Example 1

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

Lets examine this derivation:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\textbf{id} + \textbf{id})$$





parse tree

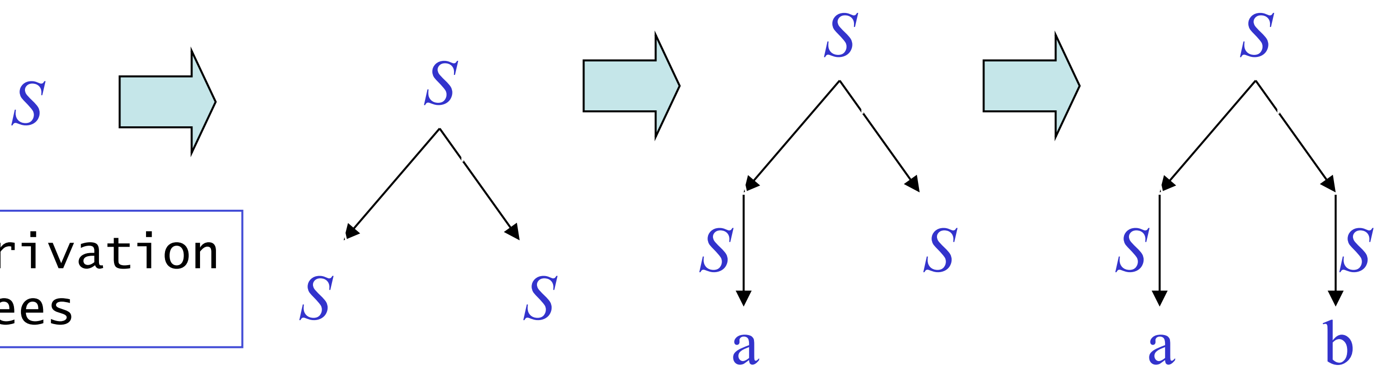
This is a top-down derivation because we start building the parse tree at the top

CFG: Parsing

Parse Tree: Example 2

$$S \rightarrow SS \mid a \mid b$$
$$ab \in L(S)$$

Derivation Trees



Leftmost derivation

$$S \Rightarrow SS \Rightarrow aS \Rightarrow ab$$

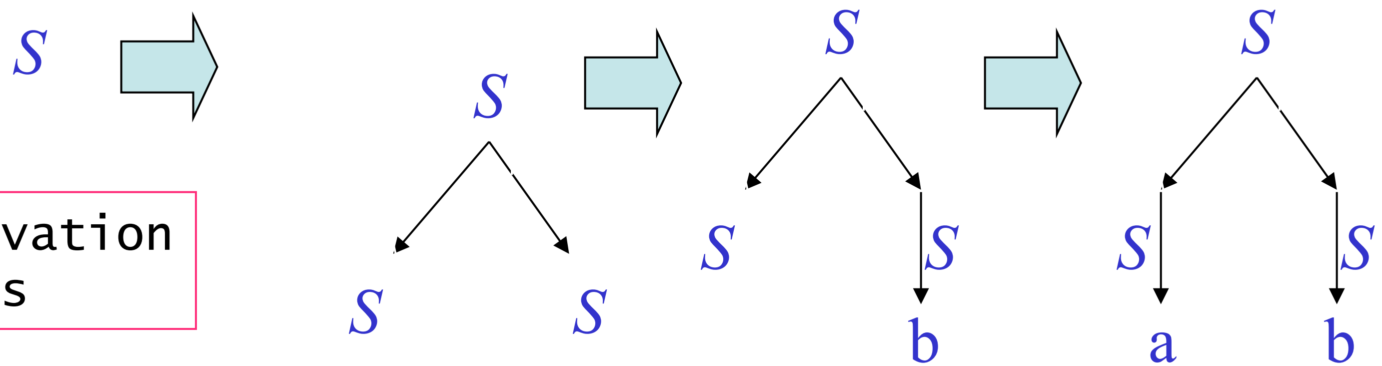
CFG: Parsing

Parse Tree: Example 2

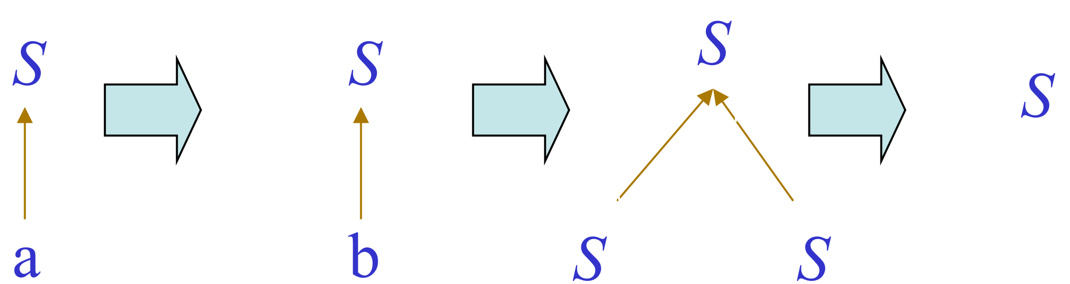
Rightmost derivation

$$S \Rightarrow SS \Rightarrow Sb \Rightarrow ab$$

Derivation Trees



Rightmost Derivation in *Reverse*



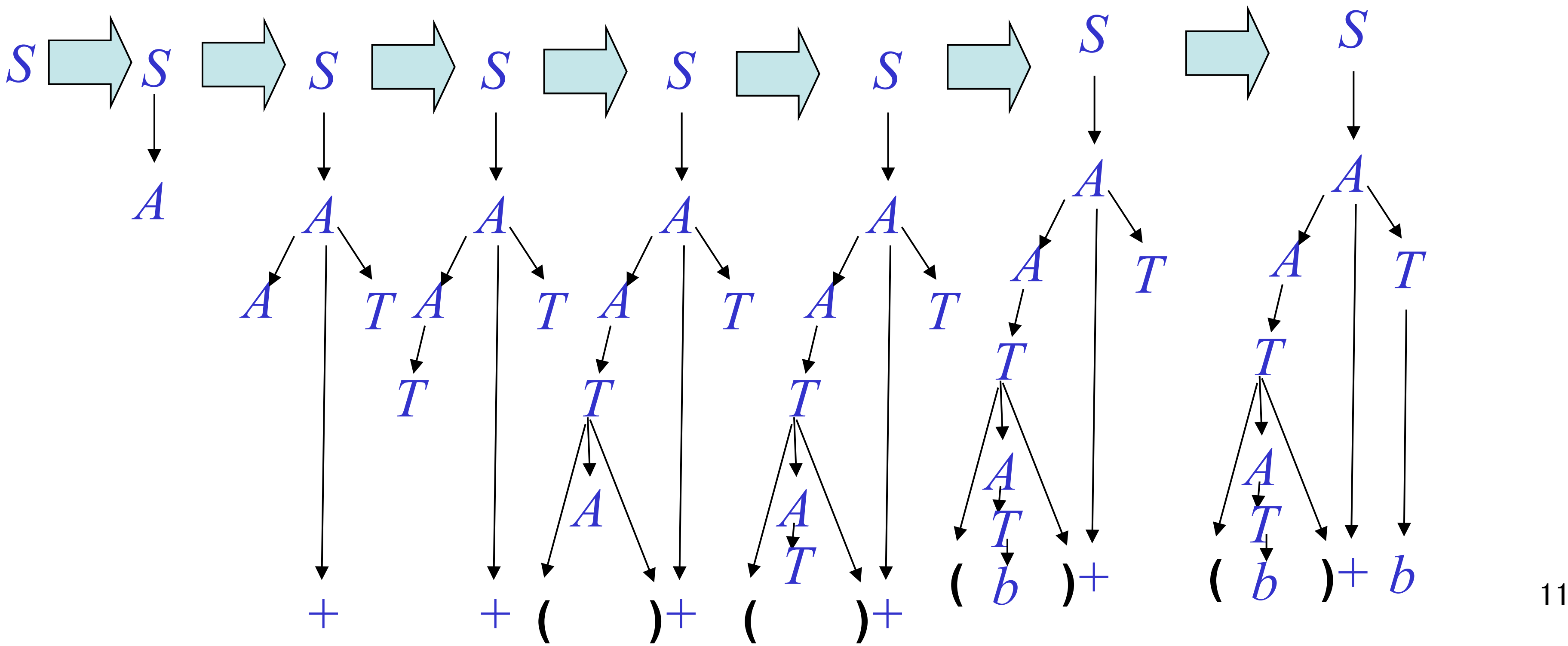
CFG: Parsing

Example 3

Consider the CFG grammar G

$$S \longrightarrow A$$
$$A \longrightarrow T \mid A + T$$
$$T \longrightarrow b \mid (A)$$

Show that (*b*)+*b* ∈ L(*G*)?



Practical Parsers

- Language/Grammar designed to enable deterministic (directed and backtrack-free) searches.
  - Top-down parsers : LL(k) languages
    - E.g., Pascal, Ada, etc.
    - Better error diagnosis and recovery.
  - Bottom-up parsers : LALR(1), LR(k) languages
    - E.g., C/C++, Java, etc.
    - Handles left recursion in the grammar.
  - Backtracking parsers
    - E.g., Prolog interpreter.

CFG: Parsing

Top-down Exhaustive Parsing

- **Exhaustive parsing** is a form of **top-down** parsing where you start with S and systematically go through all possible (say leftmost) derivations until you produce the string w.
- (You can remove sentential forms that will not work.)

- **Example:** Can the CFG $S \to SS \mid aSb \mid bSa \mid \lambda$ produce the string w = aabb, and how?
- After one step: $S \Rightarrow SS$ or aSb or bSa or λ.
- After two steps: $S \Rightarrow SSS$ or $aSbS$ or $bSaS$ or S, or $S \Rightarrow aSSb$ or $aaSbb$ or $abSab$ or ab.
- After three steps we see that: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

CFG: Parsing

Flaws of Top-down Exhaustive Parsing

- Obvious flaw: it will take a long time and a lot of memory for moderately long strings w: It is inefficient.

- For cases $w \notin L(G)$ exhaustive parsing may never end. This will especially happen if we have rules like $A \rightarrow \lambda$ that make the sentential forms 'shrink' so that we will never know if we went 'too far' with our parsing attempts.
- Similar problems occur if the parsing can get in a loop according to $A \Rightarrow B \Rightarrow A \Rightarrow B \ldots$
- Fortunately, it is always possible to remove problematic rules like $A \rightarrow \lambda$ and $A \rightarrow B$ from a CFG G.

Grammar Ambiguity

Definition

Definition: a string is derived ambiguously in a context-free grammar if it has two or more different parse trees

Definition: a grammar is ambiguous if it generates some string ambiguously

Grammar Ambiguity

A string $w \in L(G)$ is derived **ambiguously** if it has more than one derivation tree (or equivalently: if it has more than one leftmost derivation (or rightmost)).

A grammar is **ambiguous** if some strings are derived ambiguously.

Typical example: rule $S \rightarrow 0 \mid 1 \mid S+S \mid S\times S$

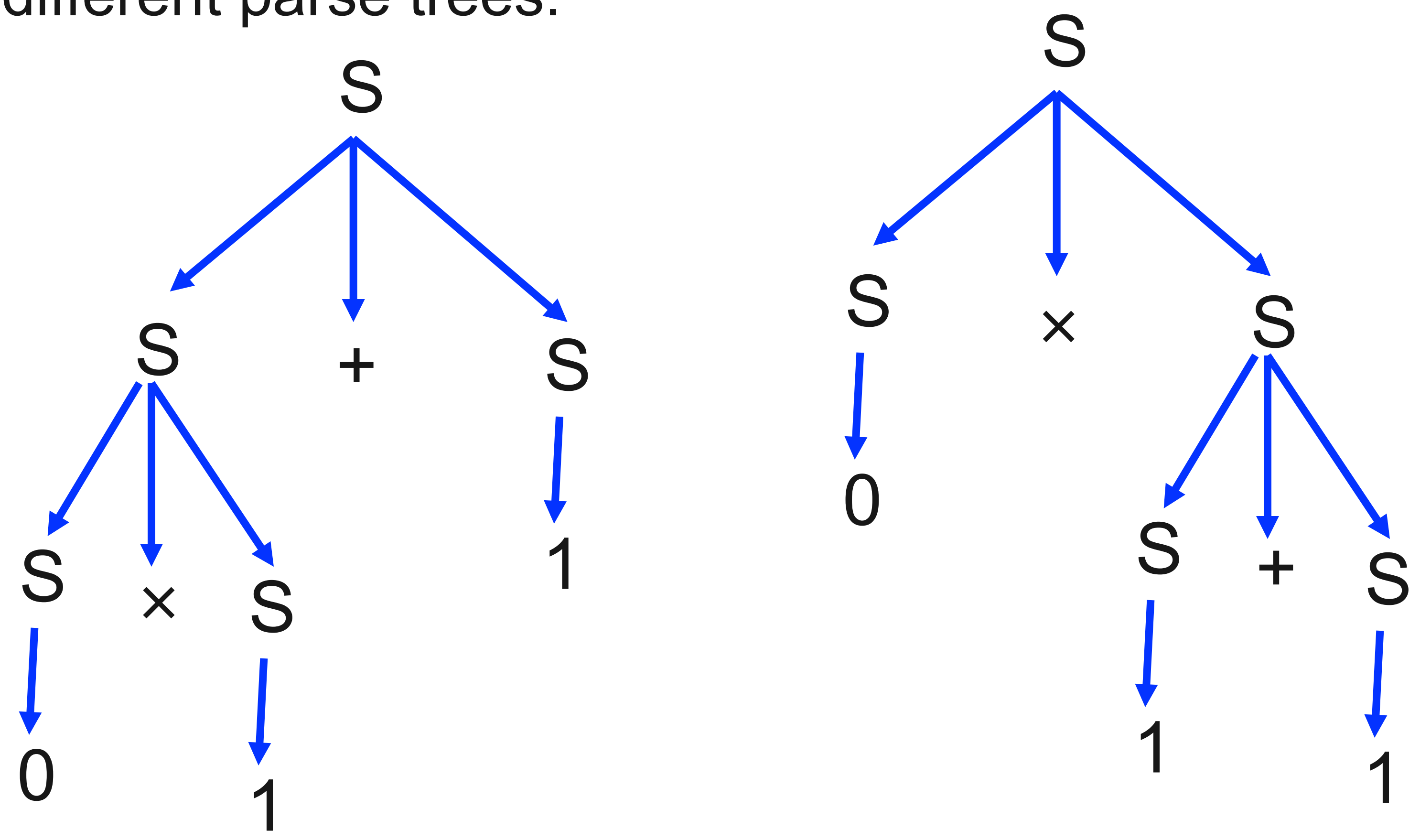
$S \Rightarrow S+S \Rightarrow S\times S+S \Rightarrow 0\times S+S \Rightarrow 0\times 1+S \Rightarrow 0\times 1+1$

versus

$S \Rightarrow S\times S \Rightarrow 0\times S \Rightarrow 0\times S+S \Rightarrow 0\times 1+S \Rightarrow 0\times 1+1$

Grammar Ambiguity

The ambiguity of $0 \times 1 + 1$ is shown by the two different parse trees:



Grammar Ambiguity

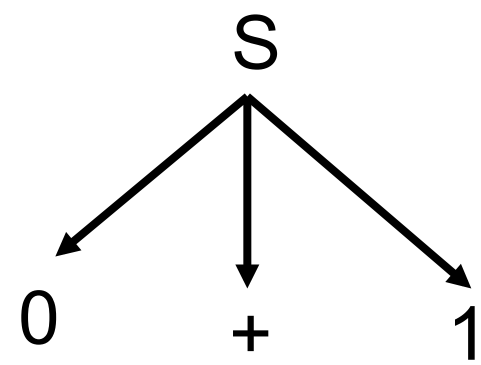
Note that the two different derivations:

$S \Rightarrow S+S \Rightarrow 0+S \Rightarrow 0+1$

and

$S \Rightarrow S+S \Rightarrow S+1 \Rightarrow 0+1$

do *not* constitute an ambiguous string 0+1 as have the same parse tree:



Ambiguity causes troubles when trying to interpret strings like: “She likes men who love women who don't smoke.”

Solutions: Use parentheses, or use precedence rules such as $a+(b\times c) = a+b\times c \neq (a+b)\times c$.

Grammar Ambiguity

Example

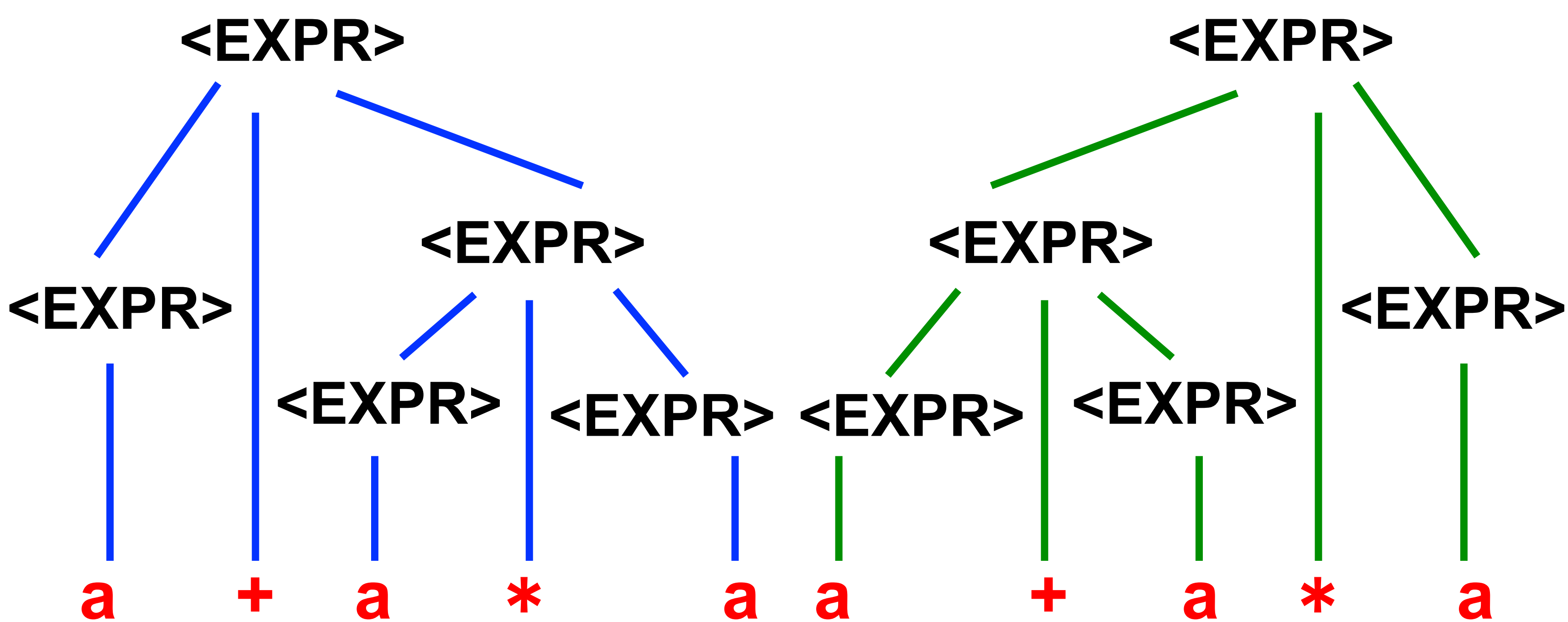
<EXPR> → <EXPR> + <EXPR>

<EXPR> → <EXPR> * <EXPR>

<EXPR> → (<EXPR>)

<EXPR> → a

Build a parse tree for a + a * a



Inherently Ambiguous

- Languages that can only be generated by ambiguous grammars are **inherently ambiguous**.

- Example: L = $\{a^n b^n c^m\} \cup \{a^n b^m c^m\}$.

$$L = \{ a^i b^j c^k \quad | \quad i = j \ \vee \ j = k \}$$

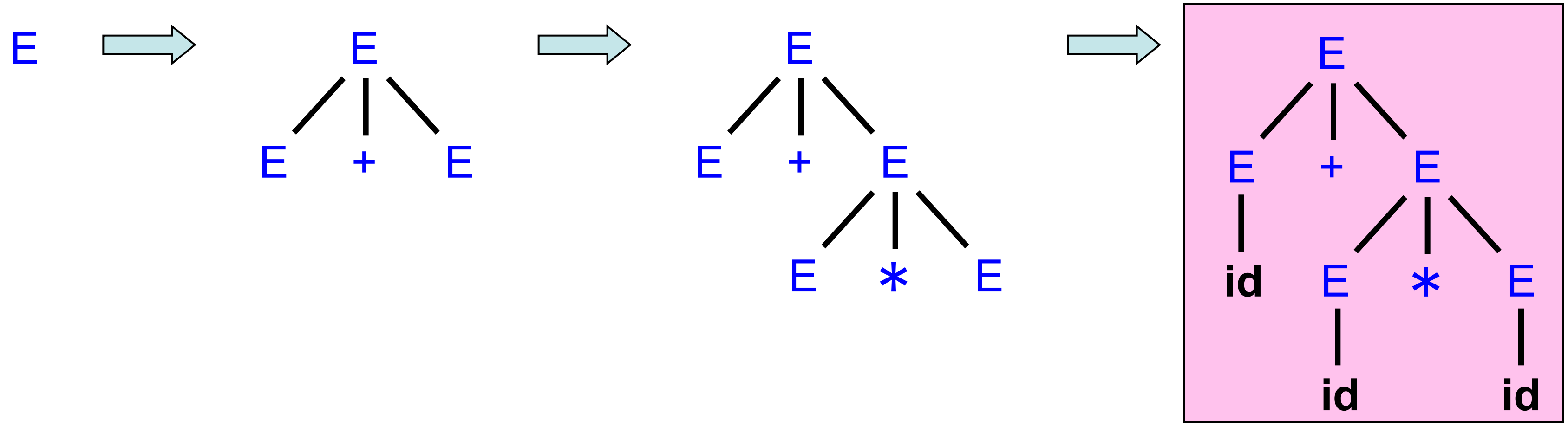
- The way to make a CFG for this L somehow has to involve the step $S \rightarrow S_1 | S_2$ where S1 produces the strings $a^n b^n c^m$ and S_2 the strings $a^n b^m c^m$.
- This will be ambiguous on strings $a^n b^n c^n$.

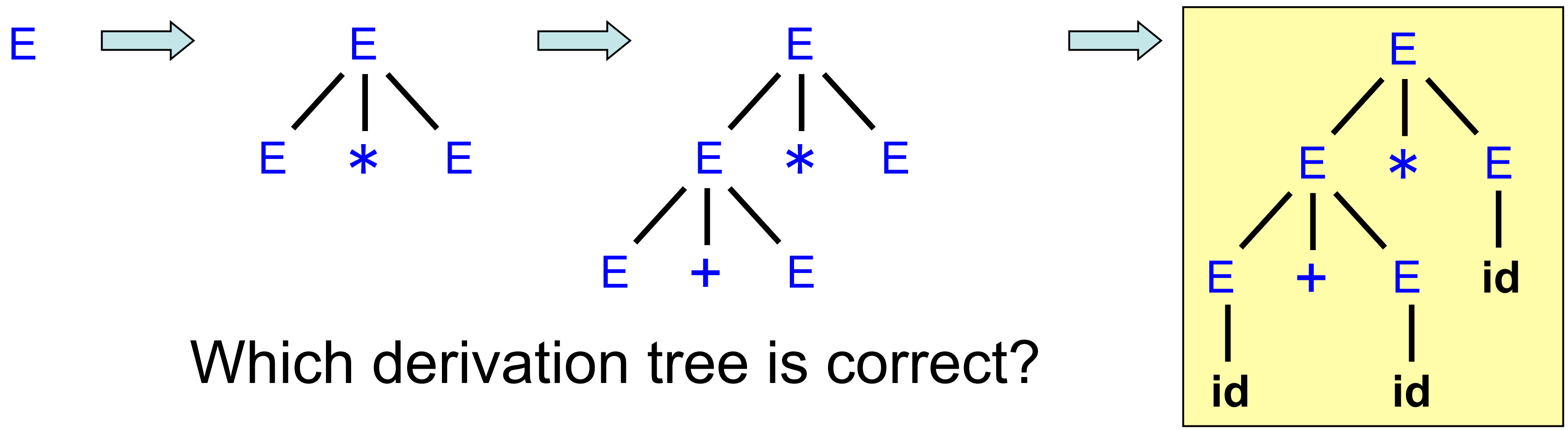
Grammar Ambiguity

Example

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

Find a derivation for the expression: **id + id * id**





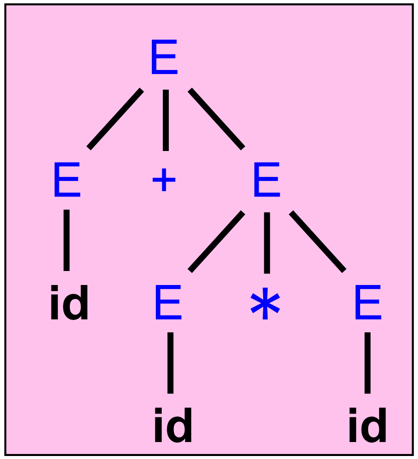
Which derivation tree is correct?

Grammar Ambiguity

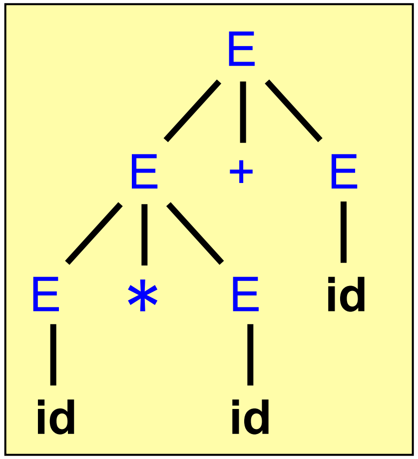
Example

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

Find a derivation for the expression: **id + id * id**



According to the grammar, both are correct.



A grammar that produces more than one parse tree for any input sentence is said to be an ambiguous grammar.

Grammar Ambiguity

One way to resolve ambiguity is to associate precedence to the operators.

Example

- $*$ has precedence over +

```
1 + 2 * 3 = 1 + (2 * 3)
1 + 2 * 3 ≠ (1 + 2)*3
```

- Associativity and precedence information is typically used to disambiguate non-fully parenthesized expressions containing unary prefix/postfix operators or binary infix operators.

Grammar Ambiguity

Example

Grammar:

$$
\begin{aligned}
\langle \mathit{stm} \rangle \;\rightarrow\; & \mathit{if} \;\langle \mathrm{expr} \rangle\; \mathit{then} \;\langle \mathit{stm} \rangle \\
\mid\; & \mathit{if} \;\langle \mathrm{expr} \rangle\; \mathit{then} \;\langle \mathit{stm} \rangle \\
& \qquad\qquad\qquad \mathit{else} \;\langle \mathit{stm} \rangle
\end{aligned}
$$

Ambiguity:

if B1 then $\underline{\mathit{if}\ \text{B2}\ \mathit{then}\ \text{S1}\ \mathit{else}\ \text{S2}}$

vs

if B1 then $\underline{\mathit{if}\ \text{B2}\ \mathit{then}\ \text{S1}}$ else S2

Grammar Ambiguity

Quiz 1

Is the following grammar ambiguous?

$$S \rightarrow PC \mid AQ$$

$$P \rightarrow aPb \mid \lambda$$

$$C \rightarrow cC \mid \lambda$$

$$Q \rightarrow bQc \mid \lambda$$

$$A \rightarrow aA \mid \lambda$$

Yes: consider the string abc

Grammar Ambiguity

Quiz 2

Is the following grammar ambiguous?

$$\boxed{S \rightarrow aS \mid Sb \mid ab \mid \lambda}$$

Yes: consider ab

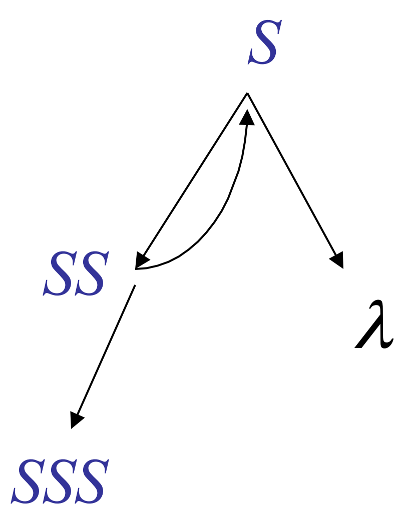
Grammar Ambiguity

Quiz

Is the following grammar ambiguous?

$$\boxed{S \to SS \mid \lambda}$$

Yes



Cyclic structure

(Illustrates ambiguous grammar with cycles.)

Simple Grammar

Definition

A CFG (V,T,S,P) is a **simple grammar** (**s-grammar**) if and only if all its productions are of the form
$A \rightarrow ax$ with
$A \in V$, $a \in T$, $x \in V^*$ and any pair (A,a) occurs at most once.

•Note, for simple grammars a left most derivation of a string $w \in L(G)$ is straightforward and requires time $|w|$.

•Example: Take the s-grammar $S \rightarrow aS|bSS|c$ with aabcc:
$S \Rightarrow aS \Rightarrow aaS \Rightarrow aabSS \Rightarrow aabcS \Rightarrow aabcc$.

Quiz: is the grammar $S \rightarrow aS|bSS|aSS|c$ s-grammar **?**

NO

Why?

The pair (S,a) occurs twice

Normal Forms

Chomsky Normal Form

Griebach Normal Form

Chomsky Normal Form CNF

A CFG is said to be in ***Chomsky Normal Form*** if every rule in the grammar has one of the following forms:

$A \rightarrow BC$	(dyadic variable productions)
$A \rightarrow a$	(unit terminal productions)
$S \rightarrow \lambda$	(λ for empty string sake only)
$\text{where } B, C \in V - \{S\}$	

Where *S* is the start variable, *A,B,C* are variables and *a* is a terminal. Thus empty string λ may only appear on the right hand side of the start symbol and other RHS are either 2 variables or a single terminal.

Chomsky Normal Form CNF

CFG→ CNF

- *Theorem:* There is an algorithm to construct a grammar G' in CNF that is *equivalent* to a CFG G.

Griebach Normal Form GNF

- A CFG is in *Griebach Normal Form* if each rule is of the form

$$A \to aA_1A_2\ldots A_n$$

$$A \to a$$

$$S \to \lambda$$

$$\text{where } A_i \in V - \{S\}$$

Griebach Normal Form GNF

CFG→ GNF

- *Theorem:* There is an algorithm to construct a grammar G' in GNF that is *equivalent* to a CFG G.

Beauty of Mathematics

Absolutely amazing!

$$1 \times 8 + 1 = 9$$
$$12 \times 8 + 2 = 98$$
$$123 \times 8 + 3 = 987$$
$$1234 \times 8 + 4 = 9876$$
$$12345 \times 8 + 5 = 98765$$
$$123456 \times 8 + 6 = 987654$$
$$1234567 \times 8 + 7 = 9876543$$
$$12345678 \times 8 + 8 = 98765432$$
$$123456789 \times 8 + 9 = 987654321$$