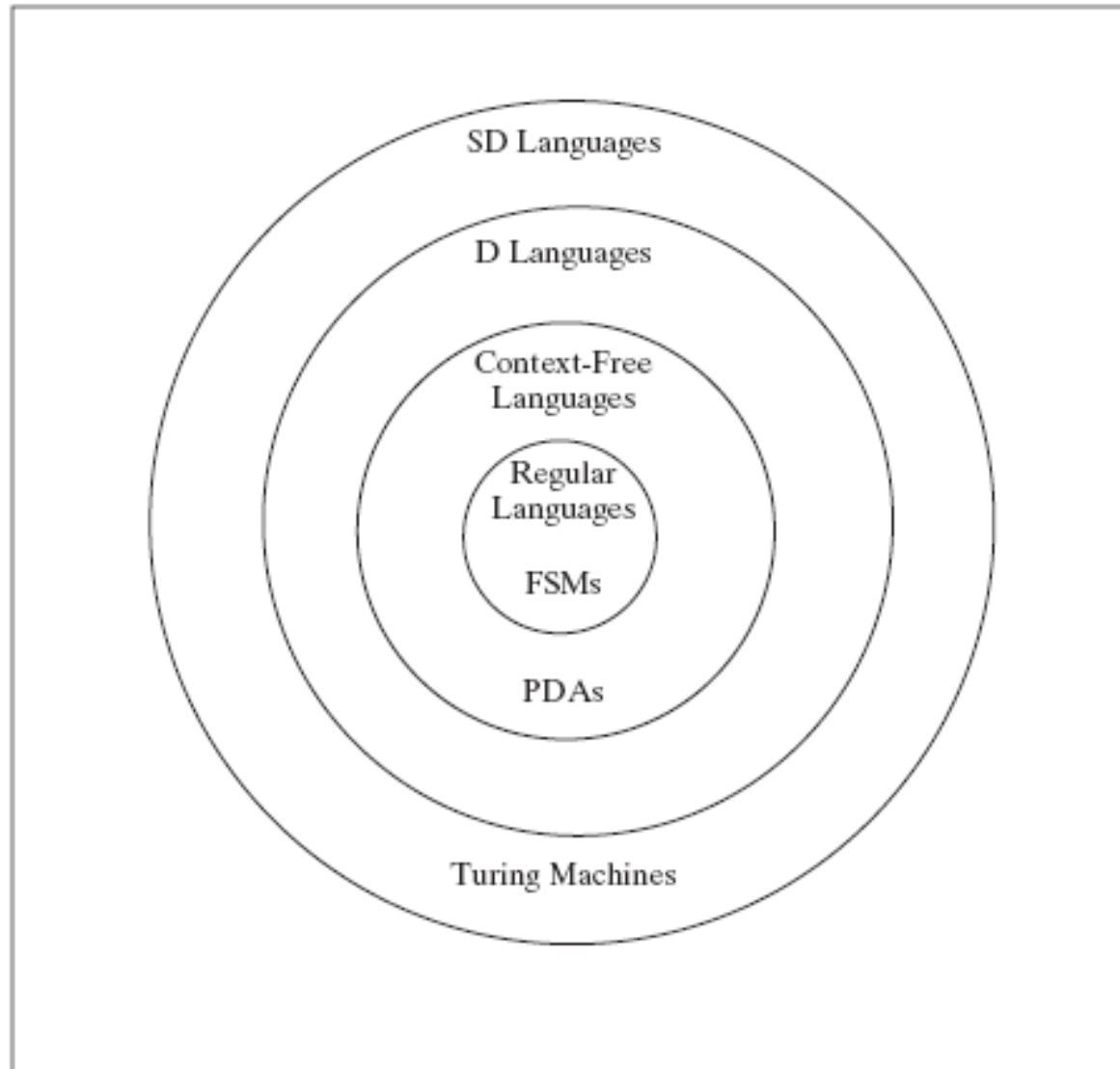




# Context-Free Grammars

## Chapter 11

# Languages and Machines





# Rewrite Systems and Grammars

A **rewrite system** (or **production system** or **rule-based system**) is:

- a list of rules, and
- an algorithm for applying them.

Each rule has a left-hand side and a right hand side.

Example rules:

$$S \rightarrow aSb$$

$$aS \rightarrow \varepsilon$$

$$aSb \rightarrow bSabSa$$

# ***Simple-rewrite***

*simple-rewrite*( $R$ : rewrite system,  $w$ : initial string) =

1. Set *working-string* to  $w$ .
2. Until told by  $R$  to halt do:  
Match the lhs of some rule against some part of *working-string*.

Replace the matched part of *working-string* with the rhs of the rule that was matched.

3. Return *working-string*.



# A Rewrite System Formalism

A rewrite system formalism specifies:

- The form of the rules
- How *simple-rewrite* works:
  - How to choose rules?
  - When to quit?

# An Example

$$w = SaS$$

Rules:

- [1]  $S \rightarrow aSb$
- [2]  $aS \rightarrow \varepsilon$

- What order to apply the rules?
- When to quit?



# Rule Based Systems

- Expert systems
- Cognitive modeling
- Business practice modeling
- General models of computation
- **Grammars**



# Grammars Define Languages

A grammar is a set of rules that are stated in terms of two alphabets:

- a ***terminal alphabet***,  $\Sigma$ , that contains the symbols that make up the strings in  $L(G)$ , and
- a ***nonterminal alphabet***, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string.

A grammar has a unique start symbol, often called  $S$ .





# Using a Grammar to Derive a String

*Simple-rewrite*  $(G, S)$  will generate the strings in  $L(G)$ .

We will use the symbol  $\Rightarrow$  to indicate steps in a derivation.

A derivation could begin with:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \dots$$

# Generating Many Strings

- Multiple rules may match.

Given:  $S \rightarrow aSb$ ,  $S \rightarrow bSa$ , and  $S \rightarrow \varepsilon$

Derivation so far:  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow$

Three choices at the next step:

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

(using rule 1),

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabSabb$

(using rule 2),

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

(using rule 3).

# Generating Many Strings

- One rule may match in more than one way.

Given:  $S \rightarrow aTTb$ ,  $T \rightarrow bTa$ , and  $T \rightarrow \varepsilon$

Derivation so far:  $S \Rightarrow aTTb \Rightarrow$

Two choices at the next step:

$S \Rightarrow a\underline{TT}b \Rightarrow abTaTb \Rightarrow$

$S \Rightarrow aT\underline{T}b \Rightarrow aTbTab \Rightarrow$

# When to Stop

May stop when:

1. The working string no longer contains any nonterminal symbols (including, when it is  $\varepsilon$ ).

In this case, we say that the working string is ***generated*** by the grammar.

Example:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

# When to Stop

May stop when:

2. There are nonterminal symbols in the working string but none of them appears on the left-hand side of any rule in the grammar.

In this case, we have a blocked or non-terminated derivation but no generated string.

Example:

Rules:  $S \rightarrow aSb$ ,  $S \rightarrow bTa$ , and  $S \rightarrow \varepsilon$

Derivations:  $S \Rightarrow aSb \Rightarrow abTab \Rightarrow$  [blocked]

# When to Stop

It is possible that neither (1) nor (2) is achieved.

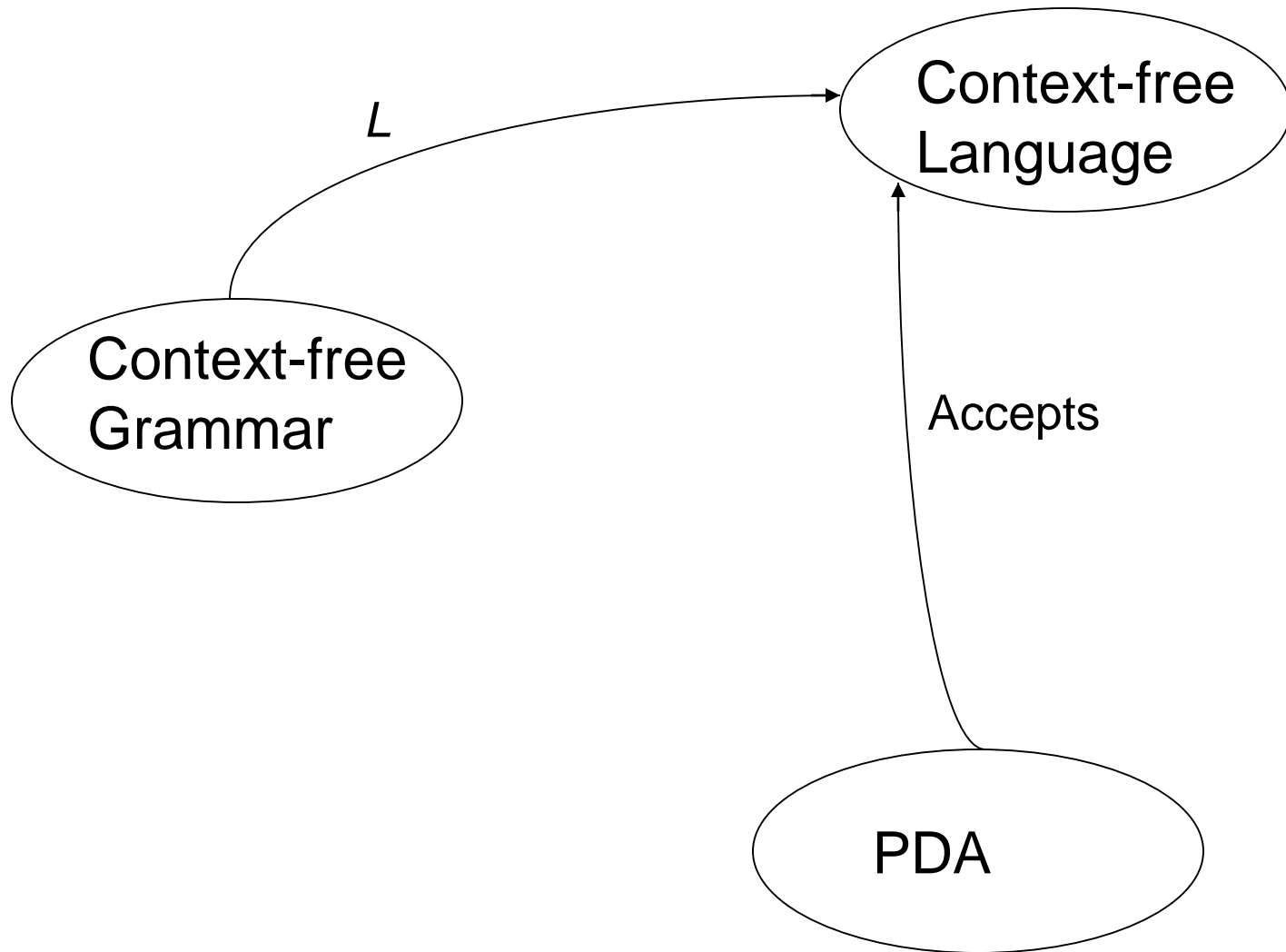
Example:

$G$  contains only the rules  $S \rightarrow Ba$  and  $B \rightarrow bB$ , with  $S$  the start symbol.

Then all derivations proceed as:

$$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow \dots$$

# Context-free Grammars, Languages, and PDAs



# More Powerful Grammars

Regular grammars must always produce strings one character at a time, moving left to right.

But it may be more natural to describe generation more flexibly.

Example 1:  $L = ab^*a$

$$\begin{array}{ll} S \rightarrow aBa & S \rightarrow aB \\ B \rightarrow \varepsilon & \text{vs.} \quad B \rightarrow a \\ B \rightarrow bB & B \rightarrow bB \end{array}$$

Example 2:  $L = \{a^n b^* a^n, n \geq 0\}$

$$\begin{array}{l} S \rightarrow B \\ S \rightarrow aSa \\ B \rightarrow \varepsilon \\ B \rightarrow bB \end{array}$$

Key distinction: Example 1 is not self-embedding.





# Context-Free Grammars

No restrictions on the form of the right hand sides.

$$S \rightarrow abDeFGab$$

But require single non-terminal on left hand side.

$$S \rightarrow$$

but not  $ASB \rightarrow$

**$A^n B^n$**



**$A^n B^n$**

$S \rightarrow \varepsilon$

$S \rightarrow a S b$



# Balanced Parentheses





# Balanced Parentheses

$$S \rightarrow \varepsilon$$

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

# Context-Free Grammars

A context-free grammar  $G$  is a quadruple,  
 $(V, \Sigma, R, S)$ , where:

- $V$  is the rule alphabet, which contains nonterminals and terminals.
- $\Sigma$  (the set of terminals) is a subset of  $V$ ,
- $R$  (the set of rules) is a finite subset of  $(V - \Sigma) \times V^*$ ,
- $S$  (the start symbol) is an element of  $V - \Sigma$ .

Example:

$(\{S, a, b\}, \{a, b\}, \{S \rightarrow a S b, S \rightarrow \varepsilon\}, S)$

# Derivations

$$x \Rightarrow_G y \text{ iff } x = \alpha A \beta$$

and  $A \rightarrow \gamma$  is in  $R$

$$y = \alpha \gamma \beta$$

$w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \dots \Rightarrow_G w_n$  is a derivation in  $G$ .

Let  $\Rightarrow_G^*$  be the reflexive, transitive closure of  $\Rightarrow_G$ .

Then the language generated by  $G$ , denoted  $L(G)$ , is:

$$\{w \in \Sigma^* : S \Rightarrow_G^* w\}.$$

# An Example Derivation

Example:

Let  $G = (\{S, a, b\}, \{a, b\}, \{S \rightarrow a S b, S \rightarrow \varepsilon\}, S)$

$S \Rightarrow a S b \Rightarrow aa S bb \Rightarrow aaa S bbb \Rightarrow aaabbb$

$S \Rightarrow^* aaabbb$





# Definition of a Context-Free Grammar

A language  $L$  is ***context-free*** iff it is generated by some context-free grammar  $G$ .

# Recursive Grammar Rules

- A rule is **recursive** iff it is  $X \rightarrow w_1 Y w_2$ , where:  
 $Y \Rightarrow^* w_3 X w_4$  for some  $w_1, w_2, w_3$ , and  $w$  in  $V^*$ .
- A grammar is recursive iff it contains at least one recursive rule.
- Examples:  $S \rightarrow (S)$

# Recursive Grammar Rules

- A rule is **recursive** iff it is  $X \rightarrow w_1 Y w_2$ , where:  
 $Y \Rightarrow^* w_3 X w_4$  for some  $w_1, w_2, w_3$ , and  $w$  in  $V^*$ .
- A grammar is recursive iff it contains at least one recursive rule.
- Examples:  $S \rightarrow (S)$   $S \rightarrow (T)$

# Recursive Grammar Rules

- A rule is **recursive** iff it is  $X \rightarrow w_1 Y w_2$ , where:  
 $Y \Rightarrow^* w_3 X w_4$  for some  $w_1, w_2, w_3$ , and  $w$  in  $V^*$ .
- A grammar is recursive iff it contains at least one recursive rule.
- Examples:  
 $S \rightarrow (S)$   
 $S \rightarrow (T)$   
 $T \rightarrow (S)$

# Self-Embedding Grammar Rules

- A rule in a grammar  $G$  is **self-embedding** iff it is :  
$$X \rightarrow w_1 Y w_2, \text{ where } Y \Rightarrow^* w_3 X w_4 \text{ and}$$
$$\text{both } w_1 w_3 \text{ and } w_4 w_2 \text{ are in } \Sigma^+.$$
- A grammar is self-embedding iff it contains at least one self-embedding rule.
- Example:

$S \rightarrow a S a$	is self-embedding
$S \rightarrow a S$	is recursive but not self-embedding
$S \rightarrow a T$	
$T \rightarrow S a$	is self-embedding



# Recursive and Self-Embedding Grammar Rules

- A rule in a grammar  $G$  is ***self-embedding*** iff it is :  
$$X \rightarrow w_1 Y w_2, \text{ where } Y \Rightarrow^* w_3 X w_4 \text{ and}$$
$$\text{both } w_1 w_3 \text{ and } w_4 w_2 \text{ are in } \Sigma^+.$$
- A grammar is self-embedding iff it contains at least one self-embedding rule.
- Example:  $S \rightarrow aSa$  is self-embedding  
 $S \rightarrow aS$  is recursive but not self-embedding



# Where Context-Free Grammars Get Their Power

- If a grammar  $G$  is not self-embedding then  $L(G)$  is regular.
- If a language  $L$  has the property that every grammar that defines it is self-embedding, then  $L$  is not regular.


$$\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$$




$$\text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$$

$G = \{\{S, a, b\}, \{a, b\}, R, S\}$ , where:

$$R = \{ \begin{array}{l} S \rightarrow aSa \\ S \rightarrow bSb \\ S \rightarrow \varepsilon \end{array} \}.$$

# Equal Numbers of a's and b's

Let  $L = \{w \in \{a, b\}^*: \#_a(w) = \#_b(w)\}$ .





# Equal Numbers of a's and b's

Let  $L = \{w \in \{a, b\}^*: \#_a(w) = \#_b(w)\}$ .

$G = \{\{S, a, b\}, \{a, b\}, R, S\}$ , where:

$$R = \{ \begin{array}{l} S \rightarrow aSb \\ S \rightarrow bSa \\ S \rightarrow SS \\ S \rightarrow \varepsilon \end{array} \}.$$

# Arithmetic Expressions

$G = (V, \Sigma, R, E)$ , where

$V = \{+, *, (, ), \text{id}, E\}$ ,

$\Sigma = \{+, *, (, ), \text{id}\}$ ,

$R = \{$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \text{id} \}$

# BNF

A notation for writing practical context-free grammars

- The symbol  $|$  should be read as “or”.

Example:  $S \rightarrow aSb \mid bSa \mid SS \mid \varepsilon$

- Allow a nonterminal symbol to be any sequence of characters surrounded by angle brackets.

Examples of nonterminals:

$\langle \text{program} \rangle$

$\langle \text{variable} \rangle$

# BNF for a Java Fragment

```
<block> ::= {<stmt-list>} | {}  
<stmt-list> ::= <stmt> | <stmt-list> <stmt>  
<stmt> ::= <block> | while (<cond>) <stmt> |  
           if (<cond>) <stmt> |  
           do <stmt> while (<cond>); |  
           <assignment-stmt>; |  
           return | return <expression> |  
           <method-invocation>;
```

# Spam Generation

<spc> → space | . | - | \_ | = | : | \* | / | : | empty

<Word> → <V> <spc> <l> <spc> <A> <spc> <G> <spc> <R> <spc> <A>

<V> → V | v | \/

<l> → I | i | ! | ; | : | ì | í | ï | î | ï | Í | Î | Î | ï | l | l

<A> → A | a | / \ | @ | ^ | Á | Â | À | Å | Ã | á | â | ä | à | å | ã

<G> → G | g | & | 6 | 9

<R> → R | r | ®

Example production:

<spc> → -

<V> → v    <l> ::= !    <A> ::= ä    <G> ::= G    <R> ::= ®    <A> ::= ^

<Word> → v-!-ä-G-®-^

These production rules yield 1,843,200 possible spellings.

How Many Ways Can You Spell V1@gra? By [Brian Hayes](#)

**American Scientist**, July-August 2007

<http://www.americanscientist.org/template/AssetDetail/assetid/55592>

# HTML

```
<ul>
  <li>Item 1, which will include a sublist</li>
    <ul>
      <li>First item in sublist</li>
      <li>Second item in sublist</li>
    </ul>
  <li>Item 2</li>
</ul>
```

A grammar:

*/\* Text is a sequence of elements.*

$$HTMLtext \rightarrow Element \ HTMLtext \mid \varepsilon$$
$$Element \rightarrow UL \mid LI \mid \dots \quad (\text{and other kinds of elements that are allowed in the body of an HTML document})$$

*/\* The <ul> and </ul> tags must match.*

$$UL \rightarrow \langle ul \rangle HTMLtext \langle /ul \rangle$$

*/\* The <li> and </li> tags must match.*

$$LI \rightarrow \langle li \rangle HTMLtext \langle /li \rangle$$



# English

$S \rightarrow NP VP$

$NP \rightarrow \text{the } Nominal \mid a \text{ } Nominal \mid Nominal \mid$   
 $ProperNoun \mid NP PP$

$Nominal \rightarrow N \mid Adjs N$

$N \rightarrow \text{cat} \mid \text{dogs} \mid \text{bear} \mid \text{girl} \mid \text{chocolate} \mid \text{rifle}$

$ProperNoun \rightarrow \text{Chris} \mid \text{Fluffy}$

$Adjs \rightarrow Adj Adjs \mid Adj$

$Adj \rightarrow \text{young} \mid \text{older} \mid \text{smart}$

$VP \rightarrow V \mid V NP \mid VP PP$

$V \rightarrow \text{like} \mid \text{likes} \mid \text{thinks} \mid \text{shots} \mid \text{smells}$

$PP \rightarrow Prep NP$

$Prep \rightarrow \text{with}$



# Designing Context-Free Grammars

- Generate related regions together.

$$A^n B^n$$

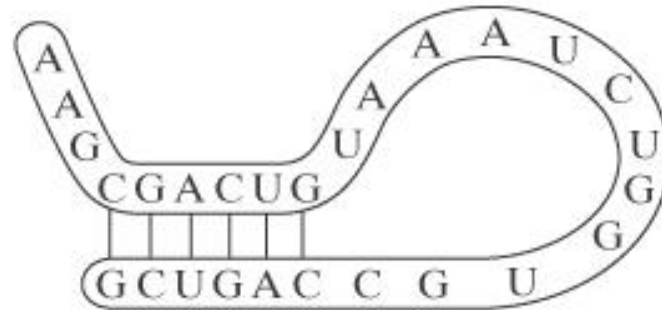
- Generate concatenated regions:

$$A \rightarrow BC$$

- Generate outside in:

$$A \rightarrow aAb$$

# Outside-In Structure and RNA Folding



(a)



(b)



# A Grammar for RNA Folding

$\langle family \rangle \rightarrow \langle tail \rangle \langle stemloop \rangle$  [1]  
 $\langle tail \rangle \rightarrow \langle base \rangle \langle base \rangle \langle base \rangle$  [1]  
 $\langle stemloop \rangle \rightarrow C \langle stemloop-5 \rangle G$  [.23]  
 $\langle stemloop \rangle \rightarrow G \langle stemloop-5 \rangle C$  [.23]  
 $\langle stemloop \rangle \rightarrow A \langle stemloop-5 \rangle U$  [.23]  
 $\langle stemloop \rangle \rightarrow U \langle stemloop-5 \rangle A$  [.23]  
 $\langle stemloop \rangle \rightarrow G \langle stemloop-5 \rangle U$  [.03]  
 $\langle stemloop \rangle \rightarrow U \langle stemloop-5 \rangle G$  [.03]  
 $\langle stemloop-5 \rangle \rightarrow \dots$



# Concatenating Independent Languages

Let  $L = \{a^m b^n c^m : n, m \geq 0\}$ .

The  $c^m$  portion of any string in  $L$  is completely independent of the  $a^m b^n$  portion, so we should generate the two portions separately and concatenate them together.



# Concatenating Independent Languages

Let  $L = \{a^m b^n c^m : n, m \geq 0\}$ .

The  $c^m$  portion of any string in  $L$  is completely independent of the  $a^m b^n$  portion, so we should generate the two portions separately and concatenate them together.

$G = (\{S, N, C, a, b, c\}, \{a, b, c\}, R, S)$  where:

$$R = \{ \begin{array}{l} S \rightarrow NC \\ N \rightarrow aNb \\ N \rightarrow \varepsilon \\ C \rightarrow cC \\ C \rightarrow \varepsilon \end{array} \}.$$


$$L = \{ a^{n_1} b^{n_1} a^{n_2} b^{n_2} .. a^{n_k} b^{n_k} : k \geq 0 \text{ and } \forall i (n_i \geq 0) \}$$

Examples of strings in  $L$ :  $\varepsilon$ , abab, aabbbaabbbbabab

Note that  $L = \{a^n b^n : n \geq 0\}^*$ .


$$L = \{ a^{n_1} b^{n_1} a^{n_2} b^{n_2} \dots a^{n_k} b^{n_k} : k \geq 0 \text{ and } \forall i (n_i \geq 0) \}$$

Examples of strings in  $L$ :  $\varepsilon$ ,  $abab$ ,  $aabbbaabbbbabab$

Note that  $L = \{a^n b^n : n \geq 0\}^*$ .

$G = (\{S, M, a, b\}, \{a, b\}, R, S)$  where:

$$R = \{ \begin{array}{l} S \rightarrow MS \\ S \rightarrow \varepsilon \\ M \rightarrow aMb \\ M \rightarrow \varepsilon \end{array} \}.$$



# Another Example: Unequal a's and b's

$$L = \{a^n b^m : n \neq m\}$$

$$G = (V, \Sigma, R, S), \text{ where}$$
$$V = \{a, b, S, \quad \},$$
$$\Sigma = \{a, b\},$$
$$R =$$

# Another Example: Unequal a's and b's

$$L = \{a^n b^m : n \neq m\}$$

$G = (V, \Sigma, R, S)$ , where

$$V = \{a, b, S, A, B\},$$

$$\Sigma = \{a, b\},$$

$$R =$$

$S \rightarrow A$  /\* more a's than b's

$S \rightarrow B$  /\* more b's than a's

$A \rightarrow a$  /\* at least one extra a generated

$A \rightarrow aA$

$A \rightarrow aAb$

$B \rightarrow b$  /\* at least one extra b generated

$B \rightarrow Bb$

$B \rightarrow aBb$



# Simplifying Context-Free Grammars

$G = (\{S, A, B, C, D, a, b\}, \{a, b\}, R, S)$ , where

$R =$

$\{ S \rightarrow AB \mid AC$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow aA$

$C \rightarrow bCa$

$D \rightarrow AB \}$

# Unproductive Nonterminals

*removeunproductive*( $G$ : CFG) =

1.  $G' = G$ .
2. Mark every nonterminal symbol in  $G'$  as unproductive.
3. Mark every terminal symbol in  $G'$  as productive.
4. Until one entire pass has been made without any new symbol being marked do:
  - For each rule  $X \rightarrow \alpha$  in  $R$  do:
    - If every symbol in  $\alpha$  has been marked as productive and  $X$  has not yet been marked as productive then:
      - Mark  $X$  as productive.
5. Remove from  $G'$  every unproductive symbol.
6. Remove from  $G'$  every rule that contains an unproductive symbol.
7. Return  $G'$ .

# Unreachable Nonterminals

*removeunreachable*( $G$ : CFG) =

1.  $G' = G$ .
2. Mark  $S$  as reachable.
3. Mark every other nonterminal symbol as unreachable.
4. Until one entire pass has been made without any new symbol being marked do:
  - For each rule  $X \rightarrow \alpha A \beta$  (where  $A \in V - \Sigma$ ) in  $R$  do:
    - If  $X$  has been marked as reachable and  $A$  has not then:
      - Mark  $A$  as reachable.
5. Remove from  $G'$  every unreachable symbol.
6. Remove from  $G'$  every rule with an unreachable symbol on the left-hand side.
7. Return  $G'$ .

# Proving the Correctness of a Grammar

$$A^nB^n = \{a^n b^n : n \geq 0\}$$

$$G = (\{S, a, b\}, \{a, b\}, R, S),$$

$$R = \{ \begin{array}{l} S \rightarrow a S b \\ S \rightarrow \varepsilon \end{array} \}$$

- Prove that  $G$  generates only strings in  $L$ .
- Prove that  $G$  generates all the strings in  $L$ .

# Proving the Correctness of a Grammar

To prove that  $G$  generates only strings in  $L$ :

Imagine the process by which  $G$  generates a string as the following loop:

1.  $st := S$ .
2. Until no nonterminals are left in  $st$  do:
  - 2.1. Apply some rule in  $R$  to  $st$ .
3. Output  $st$ .

Then we construct a loop invariant  $I$  and show that:

- $I$  is true when the loop begins,
- $I$  is maintained at each step through the loop, and
- $I \wedge (st \text{ contains only terminal symbols}) \rightarrow st \in L$ .

# Proving the Correctness of a Grammar

$A^nB^n = \{a^n b^n : n \geq 0\}$ .  $G = (\{S, a, b\}, \{a, b\}, R, S)$ ,

$$R = \{ \begin{array}{l} S \rightarrow a S b \\ S \rightarrow \varepsilon \end{array} \}.$$

- Prove that  $G$  generates only strings in  $L$ :

Let  $I = (\#_a(st) = \#_b(st)) \wedge (st \in a^*(S \cup \varepsilon) b^*)$ .



# Proving the Correctness of a Grammar


$A^nB^n = \{a^n b^n : n \geq 0\}$ .  $G = (\{S, a, b\}, \{a, b\}, R, S)$ ,


$$R = \{ \begin{array}{l} S \rightarrow a S b \\ S \rightarrow \varepsilon \end{array} \}.$$

- Prove that  $G$  generates all the strings in  $L$ :

Base case:  $|w| = 0$ .

Prove: If every string in  $A^nB^n$  of length  $k$ , where  $k$  is even, can be generated by  $G$ , then every string in  $A^nB^n$  of length  $k + 2$  can also be generated. For any even  $k$ , there is exactly one string in  $A^nB^n$  of length  $k$ :  $a^{k/2}b^{k/2}$ . There is also only one string of length  $k + 2$ , namely  $a a^{k/2} b^{k/2} b$ . It can be generated by first applying rule (1) to produce  $a S b$ , and then applying to  $S$  whatever rule sequence generated  $a^{k/2} b^{k/2}$ . By the induction hypothesis, such a sequence must exist.


$$L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$$


$$L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$$

$G = \{\{S, a, b\}, \{a, b\}, R, S\}$ , where:

$$R = \{S \rightarrow aSb \quad (1)$$

$$S \rightarrow bSa \quad (2)$$


$$S \rightarrow SS \quad (3)$$

$$S \rightarrow \varepsilon \}. \quad (4)$$

- Prove that  $G$  generates only strings in  $L$ :

Let  $\Delta(w) = \#_a(w) - \#_b(w)$ .

Let  $I = st \in \{a, b, S\}^* \wedge \Delta(st) = 0$ .


$$L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$$

$G = \{\{S, a, b\}, \{a, b\}, R, S\}$ , where:

$$R = \{ S \rightarrow aSb \quad (1)$$

$$S \rightarrow bSa \quad (2)$$

$$S \rightarrow SS \quad (3)$$

$$S \rightarrow \varepsilon \}. \quad (4)$$

- Prove that  $G$  generates all the strings in  $L$ :


Base case:

Induction step: if every string of length  $k$  can be generated, then every string  $w$  of length  $k+2$  can be.

$w$  is one of:  $aSb$ ,  $bSa$ ,  $aSa$ , or  $bSb$ .

Suppose  $w$  is  $aSb$  or  $bSa$ : Apply rule (1) or (2), then whatever sequence generates  $x$ .

Suppose  $w$  is  $aSa$  or  $bSb$ :


$$L = \{w \in \{a, b\}^*: \#_a(w) = \#_b(w)\}$$

$G = \{\{S, a, b\}, \{a, b\}, R, S\}$ , where:

$$R = \begin{array}{ll} S \rightarrow aSb & (1) \\ S \rightarrow bSa & (2) \\ S \rightarrow SS & (3) \\ S \rightarrow \varepsilon & (4) \end{array}.$$

Suppose  $w$  is  $axa$ :  $|w| \geq 4$ . We show that  $w = vy$ , where  $v$  and  $y$  are in  $L$ ,  $2 \leq |v| \leq k$ , and  $2 \leq |y| \leq k$ .

If that is so, then  $G$  can generate  $w$  by first applying rule (3) to produce  $SS$ , and then generating  $v$  from the first  $S$  and  $y$  from the second  $S$ . By the induction hypothesis, it must be possible for it to do that since both  $v$  and  $y$  have length  $\leq k$ .

$$L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$$

$G = \{\{S, a, b\}, \{a, b\}, R, S\}$ , where:

$$R = \{ \quad S \rightarrow aSb \quad (1)$$

$$S \rightarrow bSa \quad (2)$$

$$S \rightarrow SS \quad (3)$$

$$S \rightarrow \varepsilon \}. \quad (4)$$

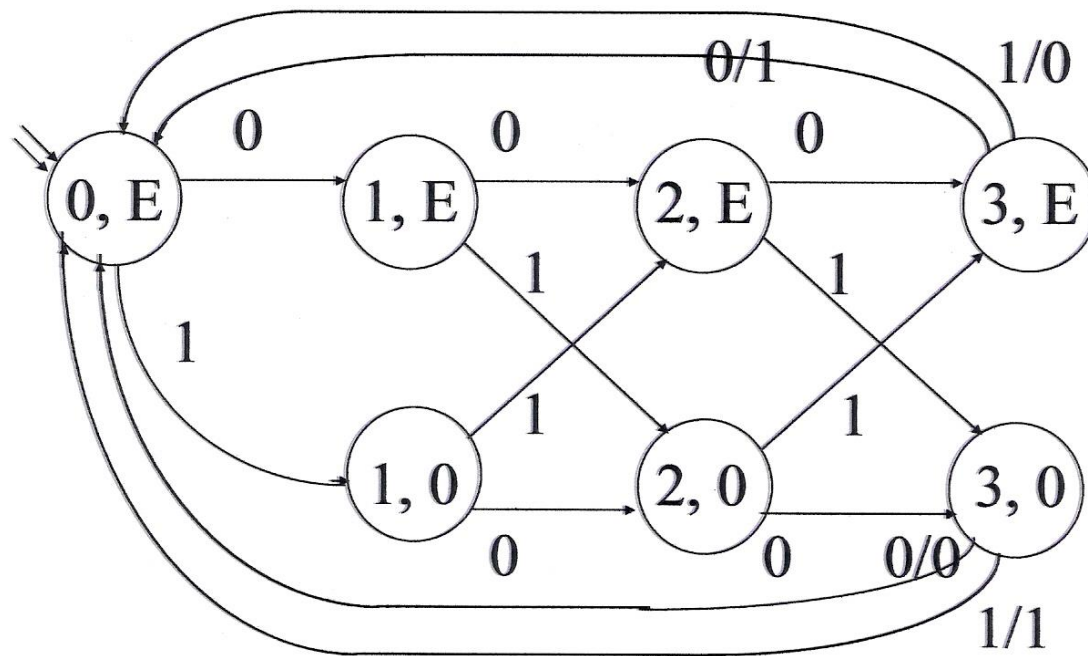
Suppose  $w$  is  $axa$ : we show that  $w = vy$ , where  $v$  and  $y$  are in  $L$ ,  $2 \leq |v| \leq k$ , and  $2 \leq |y| \leq k$ .

Build up  $w$  one character at a time. After one character, we have  $a$ .  $\Delta(a) = 1$ . Since  $w \in L$ ,  $\Delta(w) = 0$ . So  $\Delta(ax) = -1$ . The value of  $\Delta$  changes by exactly 1 each time a symbol is added to a string. Since  $\Delta$  is positive when only a single character has been added and becomes negative by the time the string  $ax$  has been built, it must at some point before then have been 0. Let  $v$  be the shortest nonempty prefix of  $w$  to have a value of 0 for  $\Delta$ . Since  $v$  is nonempty and only even length strings can have  $\Delta$  equal to 0,  $2 \leq |v|$ . Since  $\Delta$  became 0 sometime before  $w$  became  $ax$ ,  $v$  must be at least two characters shorter than  $w$ , so  $|v| \leq k$ . Since  $\Delta(v) = 0$ ,  $v \in L$ . Since  $w = vy$ , we know bounds on the length of  $y$ :  $2 \leq |y| \leq k$ . Since  $\Delta(w) = 0$  and  $\Delta(v) = 0$ ,  $\Delta(y)$  must also be 0 and so  $y \in L$ .

# Accepting Strings

Regular languages:

We care about recognizing patterns and taking appropriate actions.

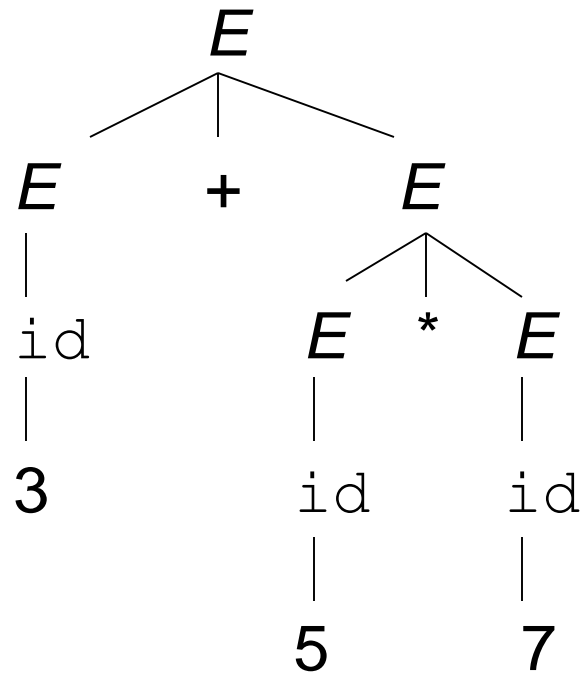




# Structure

Context free languages:

We care about structure.





# Derivations

To capture structure, we must capture the path we took through the grammar. **Derivations** do that.

Example:

$$S \rightarrow \varepsilon$$

$$S \rightarrow SS$$

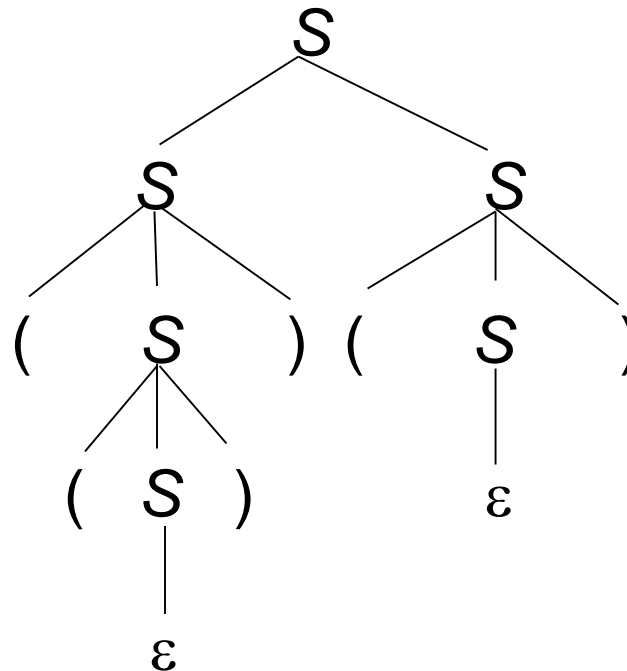
$$S \rightarrow (S)$$

1	2	3	4	5	6
$S$	$\Rightarrow SS$	$\Rightarrow (S)S$	$\Rightarrow ((S))S$	$\Rightarrow (())S$	$\Rightarrow (())(S) \Rightarrow (())()$
$S$	$\Rightarrow SS$	$\Rightarrow (S)S$	$\Rightarrow ((S))S$	$\Rightarrow ((S))(S)$	$\Rightarrow (())(S) \Rightarrow (())()$
1	2	3	5	4	6

But the order of rule application doesn't matter.

## Two vertical strips of colorful, abstract, and patterned fabric or textile designs. The left strip features a dark, textured background with vibrant, multi-colored patterns. The right strip is a lighter, more intricate design with a mix of floral, geometric, and abstract motifs in various colors.

## Parse trees capture essential structure:

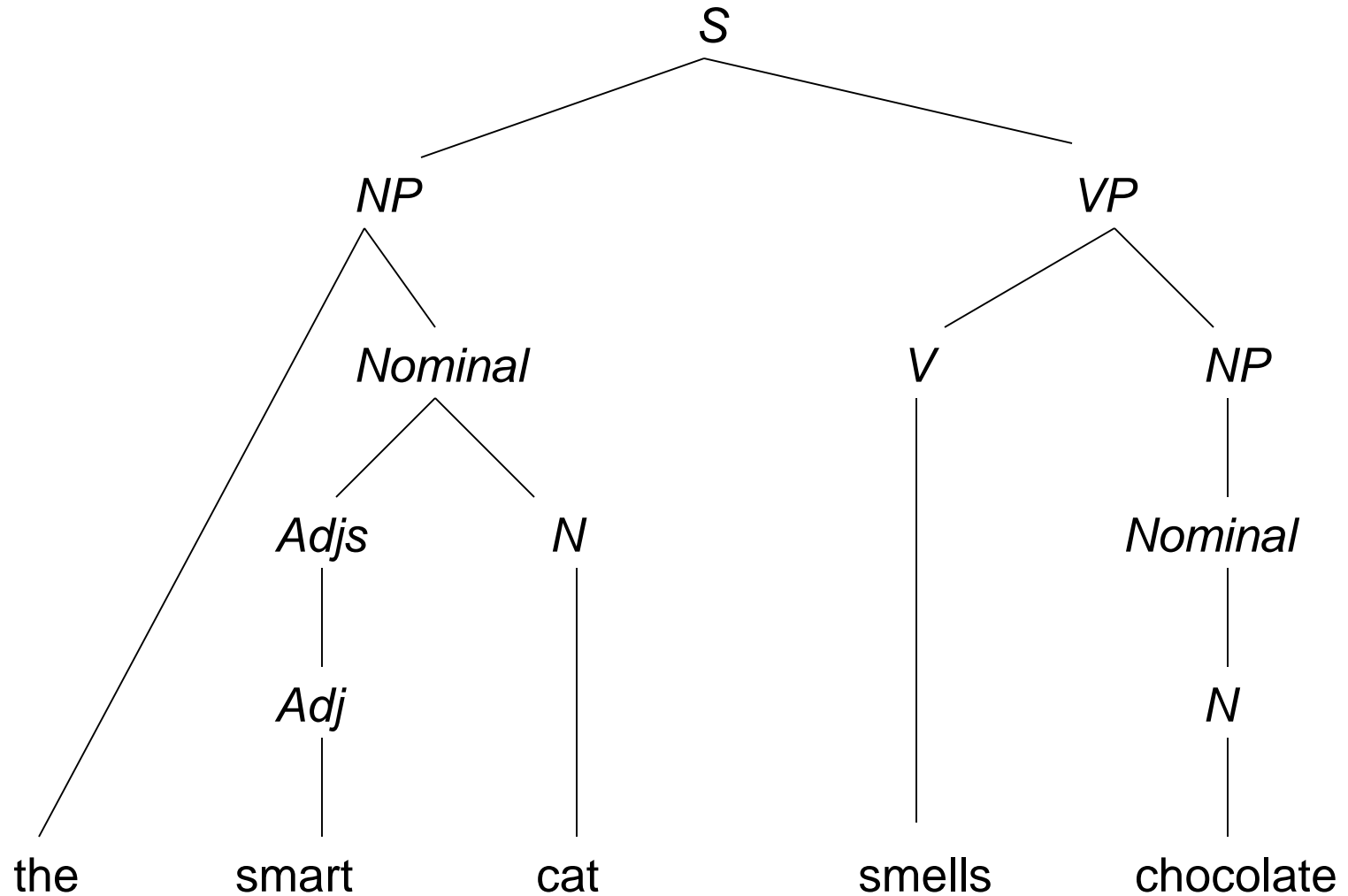
$$\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (()S \Rightarrow (())(S) \Rightarrow (()>() \\
S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))() \Rightarrow (())(S) \Rightarrow (()>() \\
1 & 2 & 3 & 5 & 4 & 6
\end{array}$$


# Parse Trees

A parse tree, derived by a grammar  $G = (V, \Sigma, R, S)$ , is a rooted, ordered tree in which:

- Every leaf node is labeled with an element of  $\Sigma \cup \{\varepsilon\}$ ,
- The root node is labeled  $S$ ,
- Every other node is labeled with some element of:  
 $V - \Sigma$ , and
- If  $m$  is a nonleaf node labeled  $X$  and the children of  $m$  are labeled  $x_1, x_2, \dots, x_n$ , then  $R$  contains the rule  
$$X \rightarrow x_1, x_2, \dots, x_n.$$

# Structure in English





# Generative Capacity

Because parse trees matter, it makes sense, given a grammar  $G$ , to distinguish between:

- $G$ 's ***weak generative capacity***, defined to be the set of strings,  $L(G)$ , that  $G$  generates, and
- $G$ 's ***strong generative capacity***, defined to be the set of parse trees that  $G$  generates.

## Two vertical strips of colorful, abstract, and patterned fabric or textile designs. The left strip features a dark, textured background with vibrant, multi-colored patterns. The right strip is a lighter, more intricate design with a mix of floral, geometric, and abstract motifs in various colors.



Two vertical strips of colorful, abstract, and patterned fabric or textile designs. The left strip features a dark, textured background with vibrant, multi-colored patterns. The right strip is a lighter, more intricate design with a mix of floral, geometric, and abstract motifs in various colors.

# Derivations of The Smart Cat

- A left-most derivation is:

$S \Rightarrow NP VP \Rightarrow \text{the } Nominal VP \Rightarrow \text{the } Adjs N VP \Rightarrow$   
 $\text{the } Adj N VP \Rightarrow \text{the smart } N VP \Rightarrow \text{the smart cat } VP \Rightarrow$   
 $\text{the smart cat } V NP \Rightarrow \text{the smart cat smells } NP \Rightarrow$   
 $\text{the smart cat smells } Nominal \Rightarrow \text{the smart cat smells } N \Rightarrow$   
 $\text{the smart cat smells chocolate}$

- A right-most derivation is:

$S \Rightarrow NP VP \Rightarrow NP V NP \Rightarrow NP V Nominal \Rightarrow NP V N \Rightarrow$   
 $NP V \text{ chocolate} \Rightarrow NP \text{ smells chocolate} \Rightarrow$   
 $\text{the } Nominal \text{ smells chocolate} \Rightarrow$   
 $\text{the } Adjs N \text{ smells chocolate} \Rightarrow$   
 $\text{the } Adjs \text{ cat smells chocolate} \Rightarrow$   
 $\text{the } Adj \text{ cat smells chocolate} \Rightarrow$   
 $\text{the smart cat smells chocolate}$

# Derivation is Not Necessarily Unique

## The is True for Regular Languages Too

### Regular Expression

$(a \cup b)^* a (a \cup b)^*$

choose a from  $(a \cup b)$

choose a from  $(a \cup b)$

choose a

choose a

choose a from  $(a \cup b)$

choose a from  $(a \cup b)$

### Regular Grammar

$S \rightarrow a$

$S \rightarrow bS$

$S \rightarrow aS$

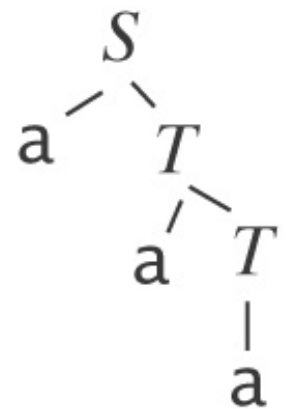
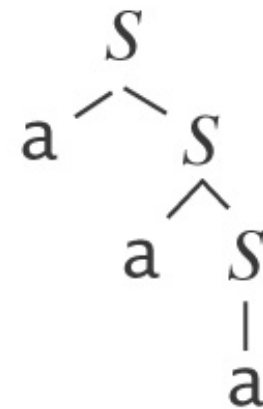
$S \rightarrow aT$

$T \rightarrow a$

$T \rightarrow b$

$T \rightarrow aT$

$T \rightarrow bT$







# Ambiguity

A grammar is ***ambiguous*** iff there is at least one string in  $L(G)$  for which  $G$  produces more than one parse tree.

For most applications of context-free grammars, this is a problem.

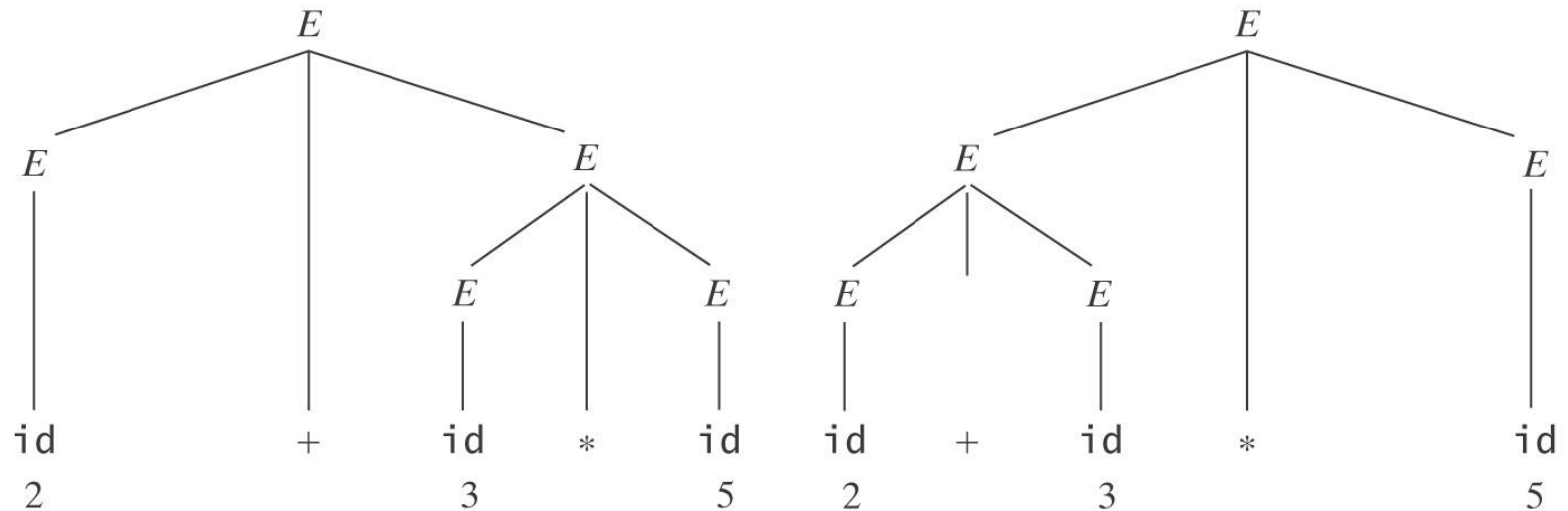
# An Arithmetic Expression Grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \text{id}$

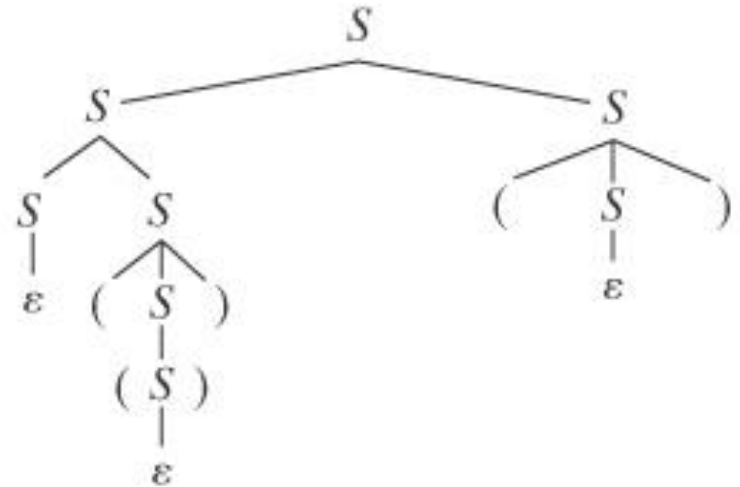
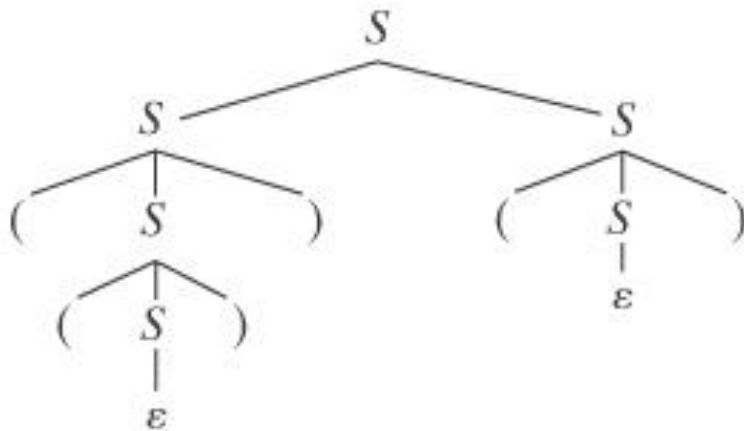


# Even a Very Simple Grammar Can be Highly Ambiguous

$S \rightarrow \varepsilon$

$S \rightarrow SS$

$S \rightarrow (S)$





# Inherent Ambiguity

Some languages have the property that every grammar for them is ambiguous. We call such languages *inherently ambiguous*.

Example:

$$L = \{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}.$$

# Inherent Ambiguity

$$L = \{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}.$$

One grammar for  $L$  has the rules:

$$S \rightarrow S_1 \mid S_2$$

$$\begin{aligned} S_1 &\rightarrow S_1 c \mid A & /* \text{Generate all strings in } \{a^n b^n c^m\}. \\ A &\rightarrow a A b \mid \varepsilon \end{aligned}$$

$$\begin{aligned} S_2 &\rightarrow a S_2 \mid B & /* \text{Generate all strings in } \{a^n b^m c^m\}. \\ B &\rightarrow b B c \mid \varepsilon \end{aligned}$$

Consider any string of the form  $a^n b^n c^n$ .

$L$  is inherently ambiguous.



# Inherent Ambiguity

Both of the following problems are undecidable:

- Given a context-free grammar  $G$ , is  $G$  ambiguous?
- Given a context-free language  $L$ , is  $L$  inherently ambiguous?



# But We Can Often Reduce Ambiguity

We can get rid of:

- $\varepsilon$  rules like  $S \rightarrow \varepsilon$ ,
- rules with symmetric right-hand sides, e.g.,

$$S \rightarrow SS$$

$$E \rightarrow E + E$$

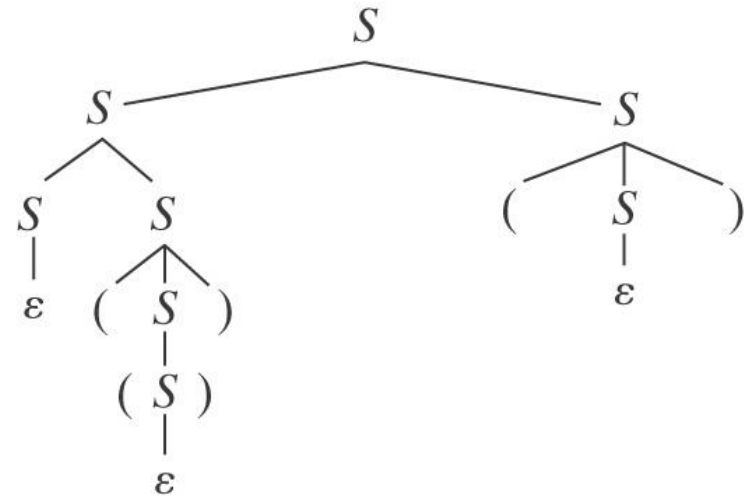
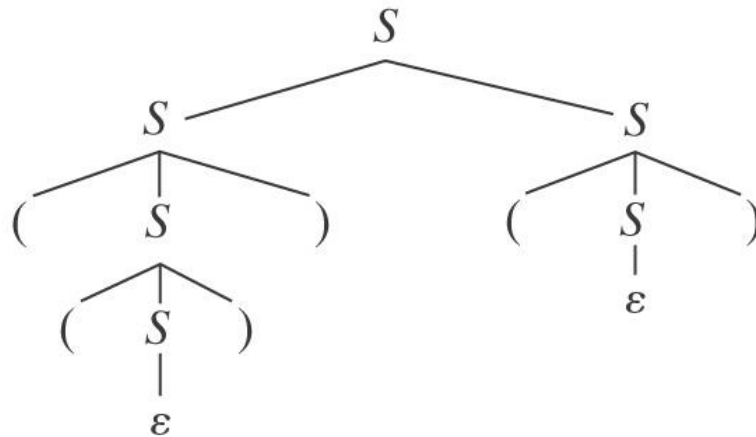
- rule sets that lead to ambiguous attachment of optional postfixes.

# A Highly Ambiguous Grammar

$S \rightarrow \varepsilon$

$S \rightarrow SS$

$S \rightarrow (S)$







# Resolving the Ambiguity with a Different Grammar

The biggest problem is the  $\varepsilon$  rule.

A different grammar for the language of balanced parentheses:

$$S^* \rightarrow \varepsilon$$

$$S^* \rightarrow S$$

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

# Nullable Variables

Examples:

$$S \rightarrow aTa$$

$$T \rightarrow \varepsilon$$

$$S \rightarrow aTa$$

$$T \rightarrow AB$$

$$A \rightarrow \varepsilon$$

$$B \rightarrow \varepsilon$$

# Nullable Variables

A variable  $X$  is **nullable** iff either:

- (1) there is a rule  $X \rightarrow \varepsilon$ , or
- (2) there is a rule  $X \rightarrow PQR\dots$  and  $P$ ,  $Q$ ,  $R$ , ... are all nullable.

So compute  $N$ , the set of nullable variables, as follows:

1. Set  $N$  to the set of variables that satisfy (1).
2. Until an entire pass is made without adding anything to  $N$  do
  - Evaluate all other variables with respect to (2).
  - If any variable satisfies (2) and is not in  $N$ , insert it.

# A General Technique for Getting Rid of $\varepsilon$ -Rules

Definition: a rule is **modifiable** iff it is of the form:

$$P \rightarrow \alpha Q \beta, \text{ for some nullable } Q.$$

*removeEps*( $G$ : cfg) =

1. Let  $G' = G$ .
2. Find the set  $N$  of nullable variables in  $G'$ .
3. Repeat until  $G'$  contains no modifiable rules that haven't been processed:  
    Given the rule  $P \rightarrow \alpha Q \beta$ , where  $Q \in N$ , add the rule  $P \rightarrow \alpha \beta$   
    if it is not already present and if  $\alpha \beta \neq \varepsilon$  and if  $P \neq \alpha \beta$ .
4. Delete from  $G'$  all rules of the form  $X \rightarrow \varepsilon$ .
5. Return  $G'$ .

$$L(G') = L(G) - \{\varepsilon\}$$

# An Example

$G = (\{S, T, A, B, C, a, b, c\}, \{a, b, c\}, R, S), R =$   
 $\{ S \rightarrow aTa$   
 $T \rightarrow ABC$   
 $A \rightarrow aA \mid C$   
 $B \rightarrow Bb \mid C$   
 $C \rightarrow c \mid \varepsilon \}$

# What If $\varepsilon \in L$ ?

*atmostoneEps*( $G$ : cfg) =

1.  $G'' = \text{removeEps}(G)$ .
2. If  $S_G$  is nullable then /\* i. e.,  $\varepsilon \in L(G)$  \*/
  - 2.1 Create in  $G''$  a new start symbol  $S^*$ .
  - 2.2 Add to  $R_{G''}$  the two rules:  
$$S^* \rightarrow \varepsilon$$
$$S^* \rightarrow S_G.$$
3. Return  $G''$ .

# But There is Still Ambiguity

$$S^* \rightarrow \varepsilon$$

$$S^* \rightarrow S$$

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

What about  $()()()$  ?



# But There is Still Ambiguity

$$S^* \rightarrow \varepsilon$$

$$S^* \rightarrow S$$

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

What about  $()()()$  ?





# But There is Still Ambiguity

$$S^* \rightarrow \varepsilon$$

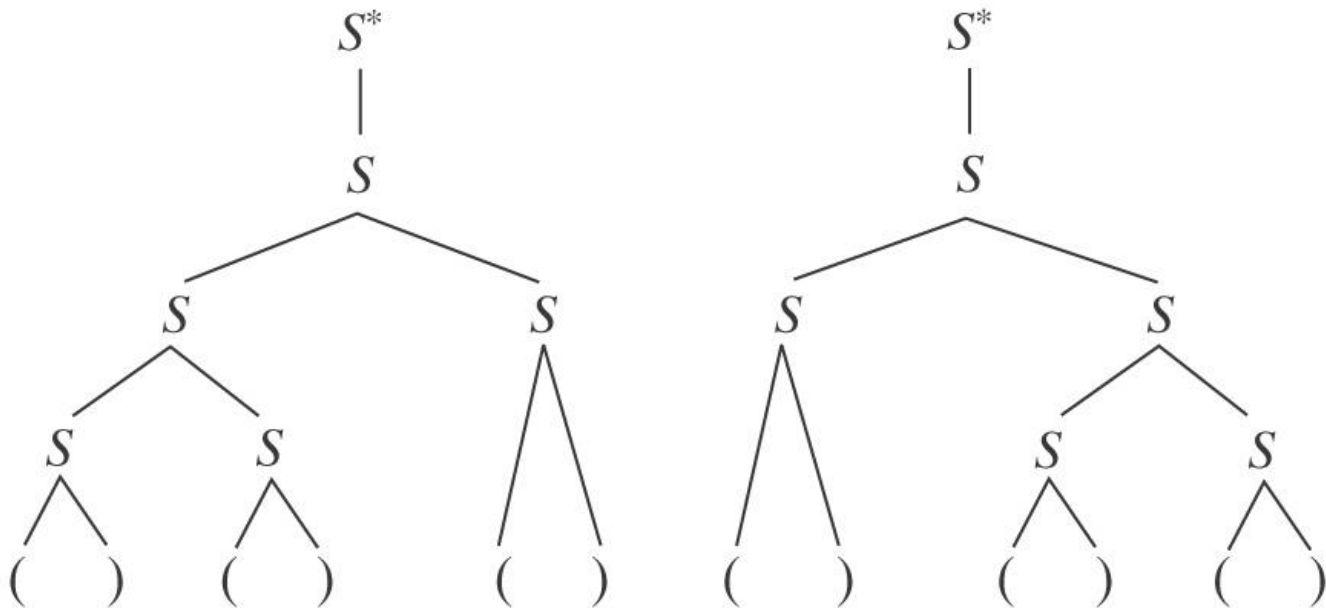
$$S^* \rightarrow S$$

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

What about  $()()()$  ?



# Eliminating Symmetric Recursive Rules

$$S^* \rightarrow \varepsilon$$

$$S^* \rightarrow S$$

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

Replace  $S \rightarrow SS$  with one of:

$$S \rightarrow SS_1$$

/\* force branching to the left

$$S \rightarrow S_1S$$

/\* force branching to the right

So we get:

$$S^* \rightarrow \varepsilon$$

$$S^* \rightarrow S$$

$$S \rightarrow SS_1$$

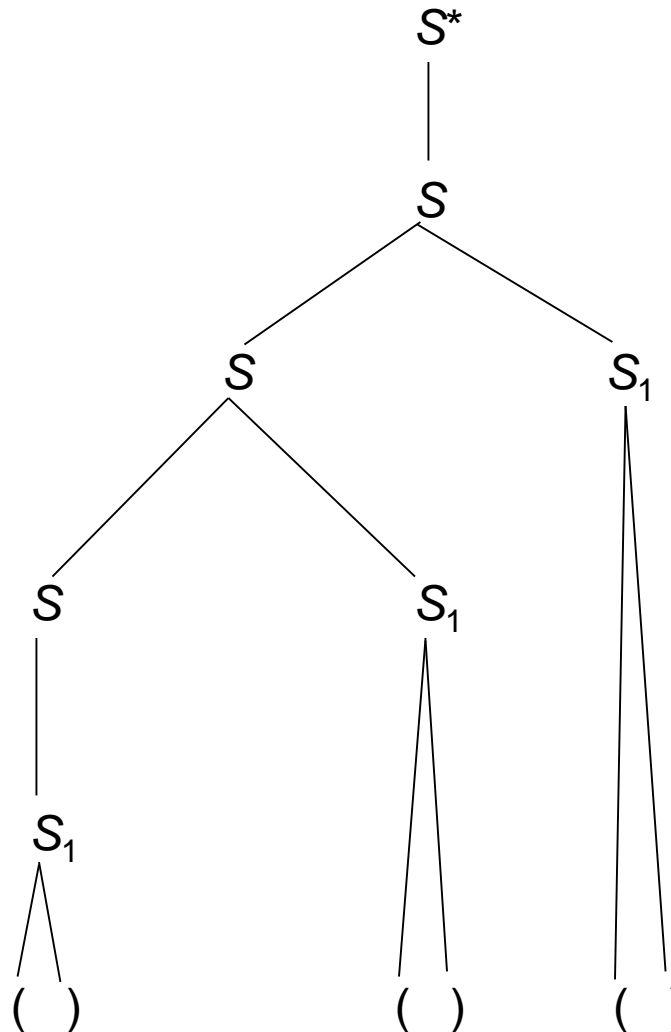
$$S \rightarrow S_1$$

$$S_1 \rightarrow (S)$$

$$S_1 \rightarrow ()$$

# Eliminating Symmetric Recursive Rules

So we get:

$$S^* \rightarrow \varepsilon$$
$$S^* \rightarrow S$$
$$S \rightarrow SS_1$$
$$S \rightarrow S_1$$
$$S_1 \rightarrow (S)$$
$$S_1 \rightarrow ()$$


# Arithmetic Expressions

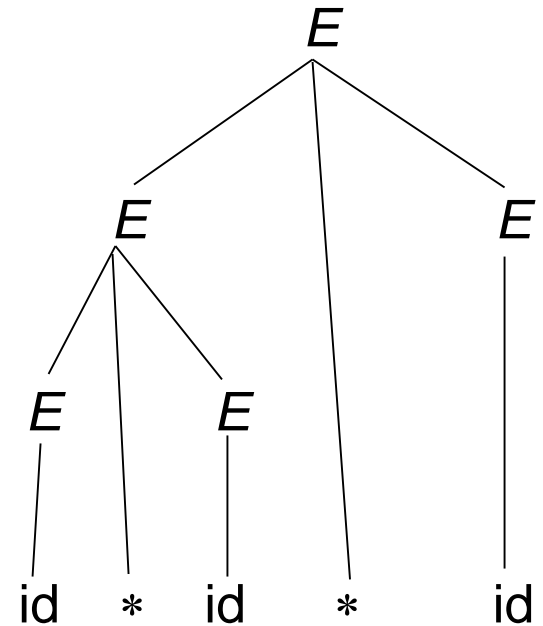
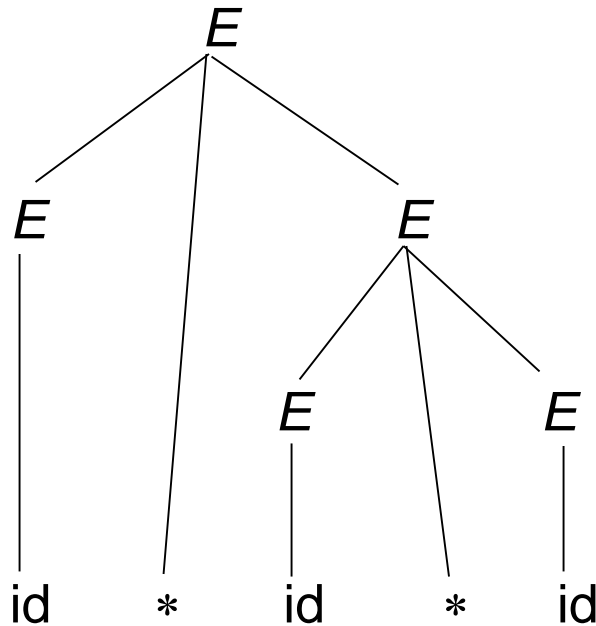
$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{id} \}$$

## Problem 1: Associativity



# Arithmetic Expressions

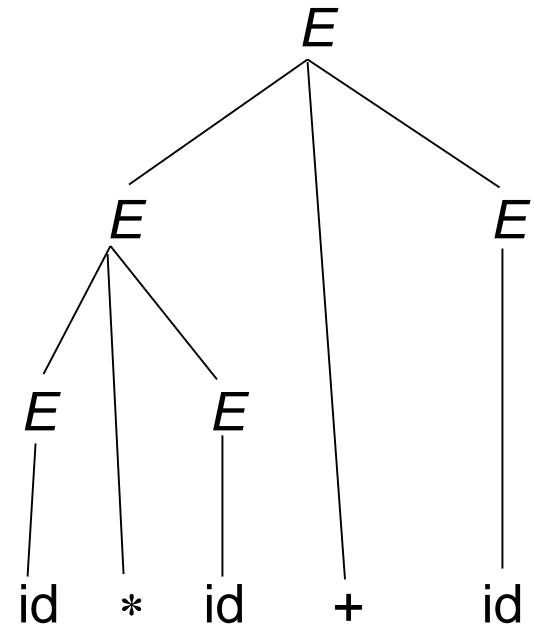
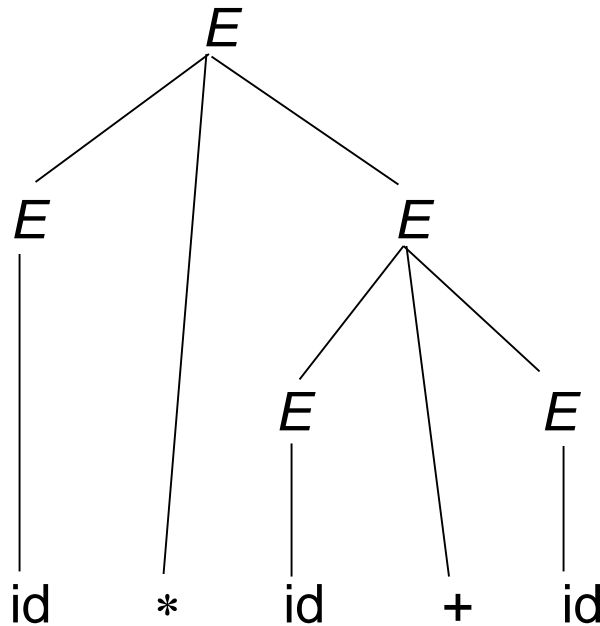
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \text{id} \}$

## Problem 2: Precedence



# Arithmetic Expressions - A Better Way

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow \text{id}$$

Examples:

`id + id * id`

`id * id * id`

# Arithmetic Expressions - A Better Way

$E \rightarrow E + T$

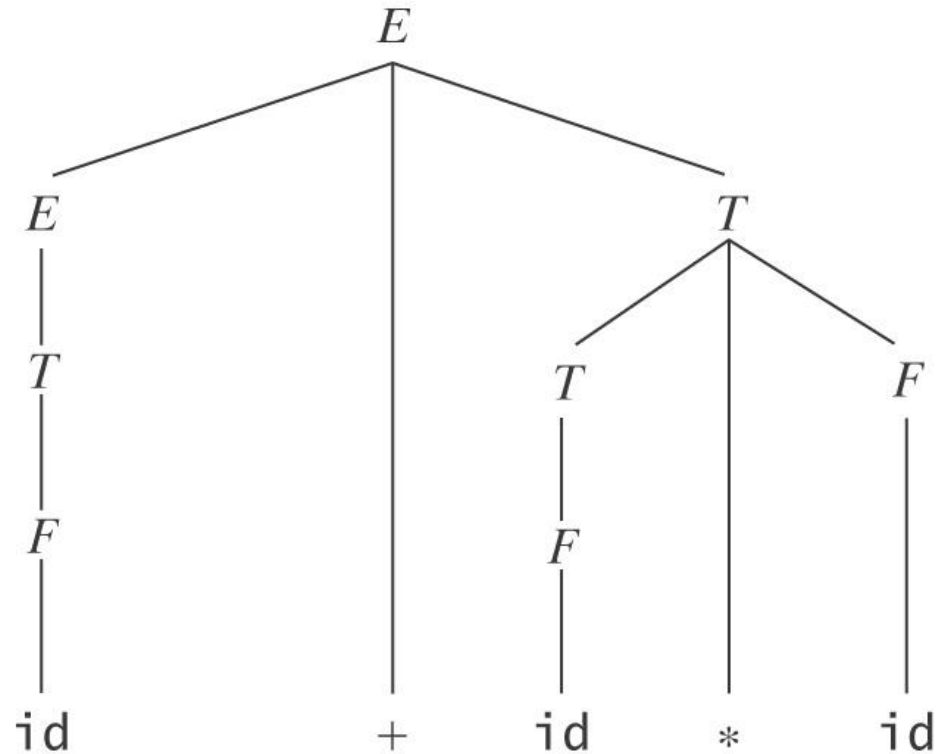
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{id}$



# The Language of Boolean Logic

$G = (V, \Sigma, R, E)$ , where

$$V = \{\wedge, \vee, \neg, \Rightarrow, (, ), \text{id}, E, \quad\},$$

$$\Sigma = \{\wedge, \vee, \neg, \Rightarrow, (, ), \text{id}\},$$

$$R = \{ E \rightarrow E \Rightarrow E_1$$

$$E \rightarrow E_1$$

$$E_1 \rightarrow E_1 \vee E_2$$

$$E_1 \rightarrow E_2$$

$$E_2 \rightarrow E_2 \wedge E_3$$

$$E_2 \rightarrow E_3$$

$$E_3 \rightarrow \neg E_3$$

$$E_3 \rightarrow E_4$$

$$E_4 \rightarrow (E)$$

$$E_4 \rightarrow \text{id}$$



# Boolean Logic isn't Regular

Suppose BL were regular. Then there is a  $k$  as specified in the Pumping Theorem.

Let  $w$  be a string of length  $2k + 1$  of the form:

$$w = \underbrace{((( ((( \text{id} ) ) ) ) ) ) ) }_k x y$$

$$y = ({}^p \text{ for some } p > 0$$

Then the string that is identical to  $w$  except that it has  $p$  additional '('s at the beginning would also be in BL. But it can't be because the parentheses would be mismatched. So BL is not regular.



# Ambiguous Attachment

The dangling else problem:

`<stmt> ::= if <cond> then <stmt>`

`<stmt> ::= if <cond> then <stmt> else <stmt>`

Consider:

`if cond1 then if cond2 then st1 else st2`

# Ambiguous Attachment

The dangling else problem:

$\langle \text{stmt} \rangle ::= \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle ::= \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

Consider:

$\text{if } \text{cond}_1 \text{ then } \underline{\text{if } \text{cond}_2 \text{ then } \text{st}_1 \text{ else } \text{st}_2}$

# Ambiguous Attachment

The dangling else problem:

$\langle \text{stmt} \rangle ::= \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle$

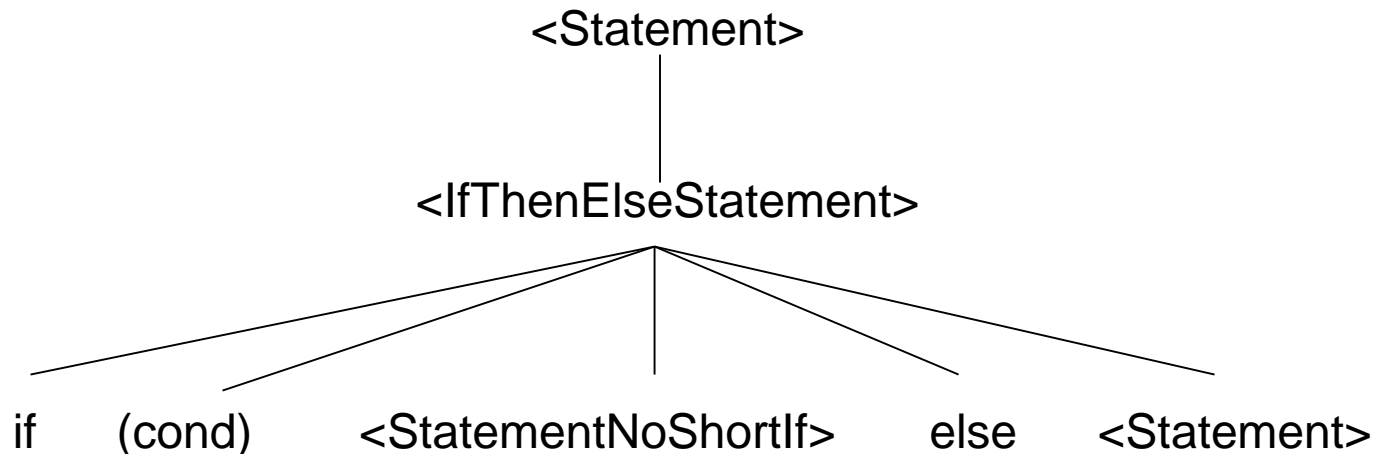
$\langle \text{stmt} \rangle ::= \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

Consider:

$\text{if } \text{cond}_1 \text{ then } \underline{\underline{\text{if } \text{cond}_2 \text{ then } \text{st}_1 \text{ else } \text{st}_2}}$

# The Java Fix

$\langle \text{Statement} \rangle ::= \langle \text{IfThenStatement} \rangle \mid \langle \text{IfThenElseStatement} \rangle \mid$   
 $\quad \langle \text{IfThenElseStatementNoShortIf} \rangle$   
 $\langle \text{StatementNoShortIf} \rangle ::= \langle \text{block} \rangle \mid$   
 $\quad \langle \text{IfThenElseStatementNoShortIf} \rangle \mid \dots$   
 $\langle \text{IfThenStatement} \rangle ::= \text{if} ( \langle \text{Expression} \rangle ) \langle \text{Statement} \rangle$   
 $\langle \text{IfThenElseStatement} \rangle ::= \text{if} ( \langle \text{Expression} \rangle )$   
 $\quad \langle \text{StatementNoShortIf} \rangle \text{ else } \langle \text{Statement} \rangle$   
 $\langle \text{IfThenElseStatementNoShortIf} \rangle ::=$   
 $\quad \text{if} ( \langle \text{Expression} \rangle ) \langle \text{StatementNoShortIf} \rangle$   
 $\quad \text{else } \langle \text{StatementNoShortIf} \rangle$





# Java Audit Rules Try to Catch These

From the **CodePro Audit Rule Set**:

## **Dangling Else**

**Severity:** Medium

### **Summary**

Use blocks to prevent dangling else clauses.

### **Description**

This audit rule finds places in the code where else clauses are not preceded by a block because these can lead to dangling else errors.

### **Example**

```
if (a > 0) if (a > 100) b = a - 100; else b = -a;
```

# Proving that $G$ is Unambiguous

A grammar  $G$  is unambiguous iff every string derivable in  $G$  has a single leftmost derivation.

$$S^* \rightarrow \varepsilon \quad (1)$$

$$S^* \rightarrow S \quad (2)$$

$$S \rightarrow SS_1 \quad (3)$$

$$S \rightarrow S_1 \quad (4)$$

$$S_1 \rightarrow (S) \quad (5)$$

$$S_1 \rightarrow () \quad (6)$$

- $S^*$ :
- $S_1$ : If the next two characters to be derived are  $()$ ,  $S_1$  must expand by rule (6). Otherwise, it must expand by rule (5).

# The Proof, Continued

$$S^* \rightarrow \varepsilon \quad (1)$$

$$S^* \rightarrow S \quad (2)$$

$$S \rightarrow SS_1 \quad (3)$$

$$S \rightarrow S_1 \quad (4)$$

$$S_1 \rightarrow (S) \quad (5)$$

$$S_1 \rightarrow () \quad (6)$$

The siblings of  $m$  is the smallest set that includes any matched set  $p$  adjacent to  $m$  and all of  $p$ 's siblings.

Example:

$$\begin{array}{cccc} ( & \underline{()} & \underline{()} & ) & \underline{()} & \underline{()} \\ & 1 & 2 & & 3 & 4 \\ \hline & & & & 5 & \end{array}$$

The set  $()$  labeled 1 has a single sibling, 2. The set  $(())$  labeled 5 has two siblings, 3 and 4.



# The Proof, Continued

$$S^* \rightarrow \varepsilon \quad (1)$$

$$S^* \rightarrow S \quad (2)$$

$$S \rightarrow SS_1 \quad (3)$$

$$S \rightarrow S_1 \quad (4)$$

$$S_1 \rightarrow (S) \quad (5)$$

$$S_1 \rightarrow () \quad (6)$$

- $S$ :
  - $S$  must generate a matched set, possibly with siblings.
  - So the first terminal character in any string that  $S$  generates is (. Call the string that starts with that ( and ends with the ) that matches it,  $s$ .
  - $S_1$  must generate a single matched set with no siblings.
  - Let  $n$  be the number of siblings of  $s$ . In order to generate those siblings,  $S$  must expand by rule (3) exactly  $n$  times before it expands by rule (4).

# The Proof, Continued

$$S^* \rightarrow \varepsilon \quad (1)$$

$$S^* \rightarrow S \quad (2)$$

$$S \rightarrow SS_1 \quad (3)$$

$$S \rightarrow S_1 \quad (4)$$

$$S_1 \rightarrow (S) \quad (5)$$

$$S_1 \rightarrow () \quad (6)$$

- $S$ :

$$\frac{(((\ ))(\ )) \ () \ () \ ((\ ))}{s}$$

$s$  has 3 siblings.

$S$  must expand by rule (3) 3 times before it uses rule (4).

Let  $p$  be the number of occurrences of  $S_1$  to the right of  $S$ .

If  $p < n$ ,  $S$  must expand by rule (3).

If  $p = n$ ,  $S$  must expand by rule (4).

# Going Too Far

$S \rightarrow NP VP$

$NP \rightarrow \text{the Nominal} \mid \text{Nominal} \mid \text{ProperNoun} \mid NP PP$

$\text{Nominal} \rightarrow N \mid \text{Adjs } N$

$N \rightarrow \text{cat} \mid \text{girl} \mid \text{dogs} \mid \text{ball} \mid \text{chocolate} \mid$   
 $\text{bat}$

$\text{ProperNoun} \rightarrow \text{Chris} \mid \text{Fluffy}$

$\text{Adjs} \rightarrow \text{Adj Adjs} \mid \text{Adj}$

$\text{Adj} \rightarrow \text{young} \mid \text{older} \mid \text{smart}$

$VP \rightarrow V \mid V NP \mid VP PP$

$V \rightarrow \text{like} \mid \text{likes} \mid \text{thinks} \mid \text{hits}$

$PP \rightarrow \text{Prep } NP$

$\text{Prep} \rightarrow \text{with}$

- Chris likes the girl with the cat.
- Chris shot the bear with a rifle.



# Going Too Far

- Chris likes the girl with the cat.
- Chris shot the bear with a rifle.

# Going Too Far

- Chris likes the girl with the cat.

- Chris shot the bear with a rifle.



- Chris shot the bear with a rifle.



# Comparing Regular and Context-Free Languages

## Regular Languages

- regular exprs.  
or
- regular grammars
- recognize

## Context-Free Languages

- context-free grammars
- parse

# A Testimonial

Also, you will be happy to know that I just made use of the context-free grammar skills I learned in your class! I am working on Firefox at IBM this summer and just found an inconsistency between how the native Firefox code and a plugin by Adobe parse SVG path data elements. In order to figure out which code base exhibits the correct behavior I needed to trace through the grammar

<http://www.w3.org/TR/SVG/paths.html#PathDataBNF>. T

hanks to your class I was able to determine that the bug is in the Adobe plugin. Go OpenSource!