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∈ L(G).

•Analytical aspect: Given a CFG G and strings w, how do you decide if w∈ 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

  A program that determines if a string $w \in L(G)$ 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*.

CFG: Parsing

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:

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

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

Yes because there is the following derivation:

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

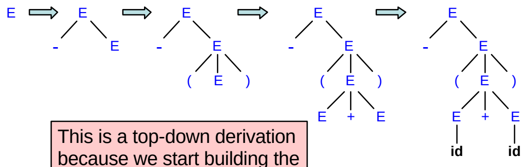
CFG: Parsing

Parse Tree: Example 1

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

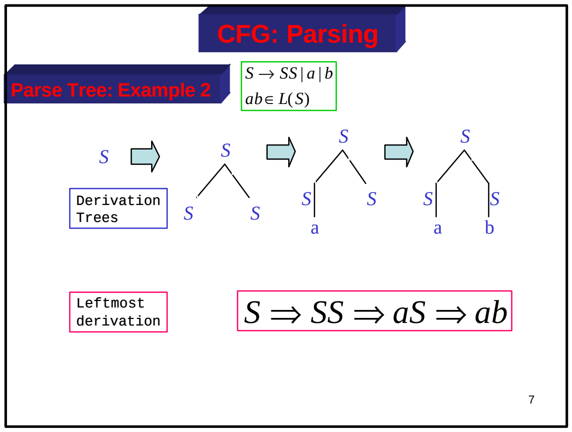
Lets examine this derivation:

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



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

parse tree



CFG: Parsing

Parse Tree: Example 2

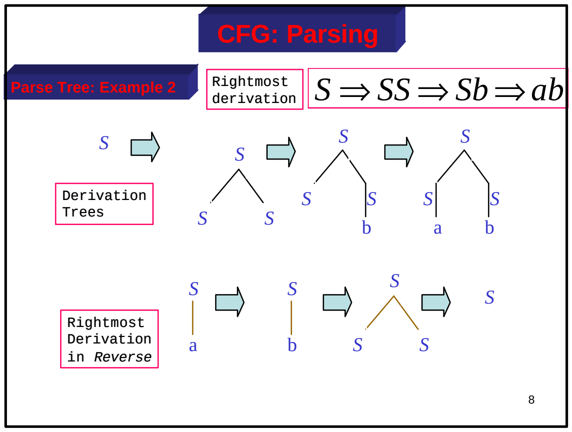
$S \rightarrow SS \mid a \mid b$

$ab \in L(S)$

Derivation Trees

Leftmost derivation

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



CFG: Parsing

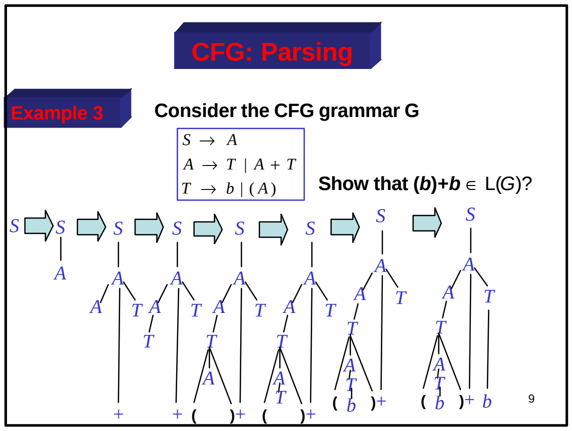
Parse Tree: Example 2

Rightmost derivation

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

Derivation Trees

Rightmost Derivation in *Reverse*



CFG: Parsing

Example 3

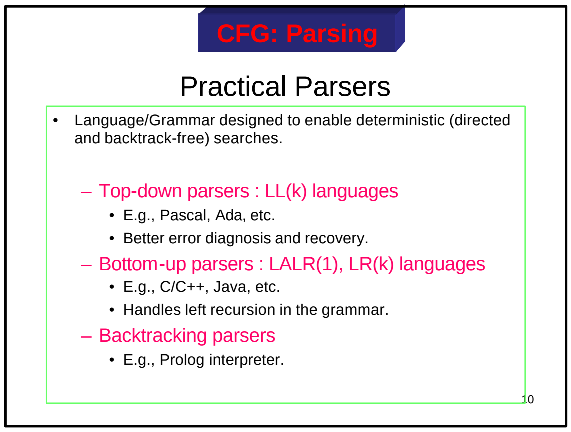
Consider the CFG grammar G

$S \rightarrow A$

$A \rightarrow T \mid A + T$

$T \rightarrow b \mid (A)$

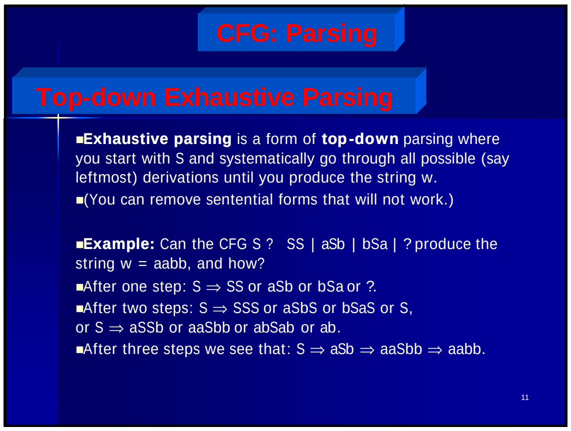
Show that (*b*)+*b* $\in$ L(G)?



CFG: Parsing

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.

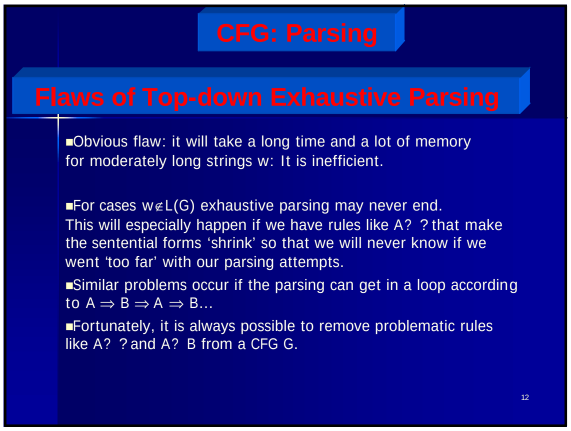


CFG: Parsing

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 ? SS | aSb | bSa | ? produce the string w = aabb, and how?
- After one step: S ⇒ SS or aSb or bSa or ?.
- After two steps: S ⇒ SSS or aSbS or bSaS or S, or S ⇒ aSSb or aaSbb or abSab or ab.
- After three steps we see that: S ⇒ aSb ⇒ aaSbb ⇒ aabb.

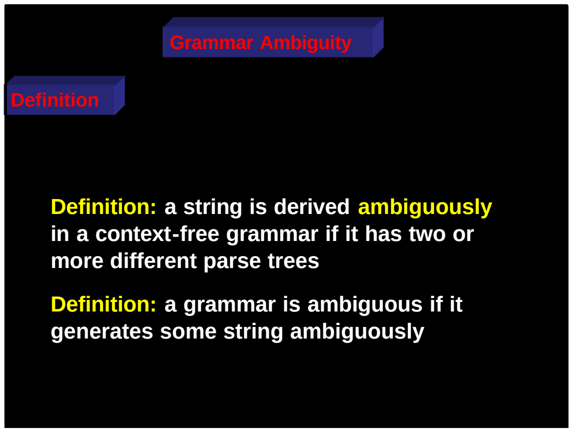


CFG: Parsing

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∉L(G) exhaustive parsing may never end. This will especially happen if we have rules like A? ? 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 ⇒ B ⇒ A ⇒ B…
- Fortunately, it is always possible to remove problematic rules like A? ? and A? B from a CFG G.

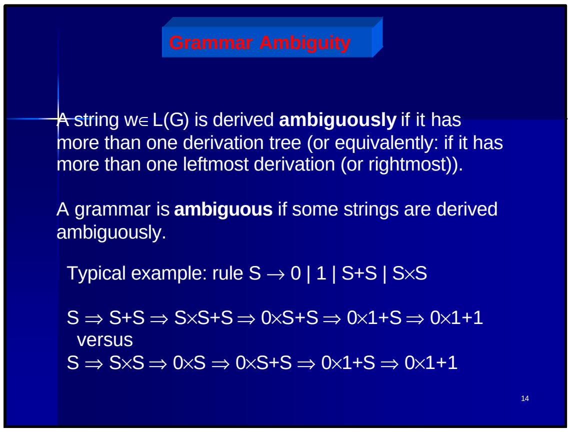


Grammar Ambiguity

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∈ 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)).

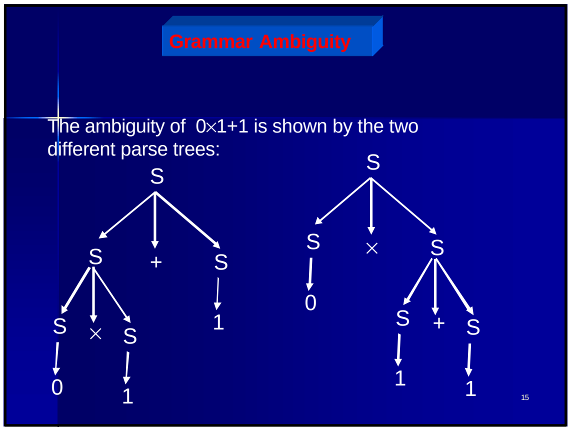
A grammar is **ambiguous** if some strings are derived ambiguously.

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

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

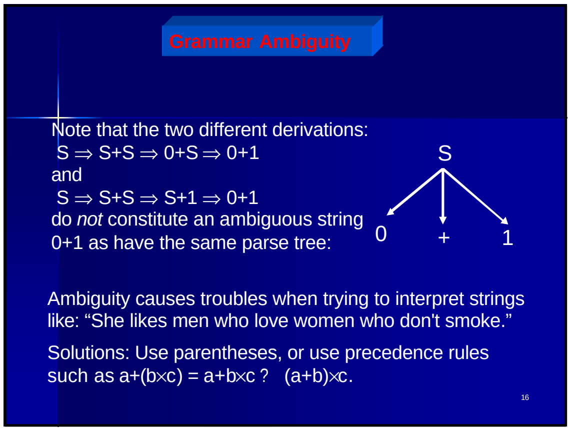
versus

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



Grammar Ambiguity

The ambiguity of 0×1+1 is shown by the two different parse trees:



Grammar Ambiguity

Note that the two different derivations:

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

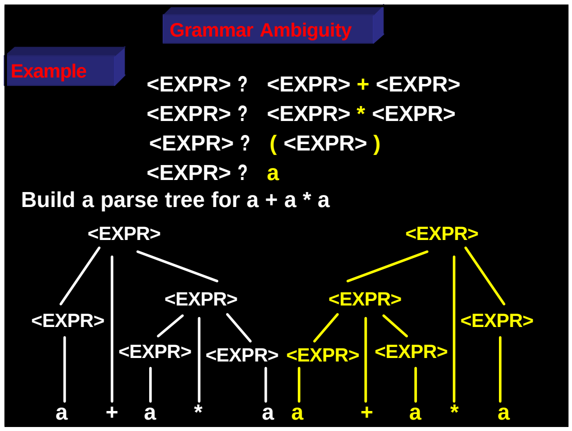
and

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

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×c) = a+b×c ? (a+b)×c.



Grammar Ambiguity

Example

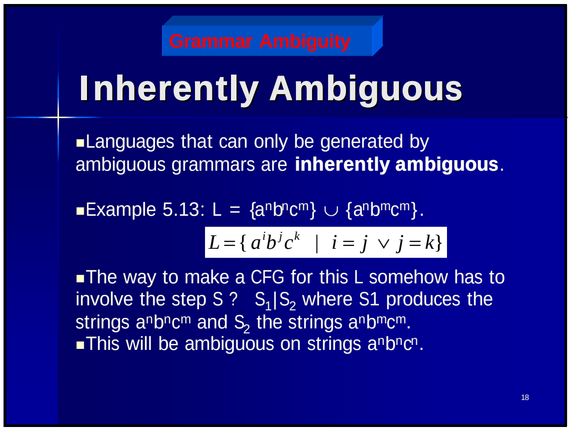
<EXPR> ? <EXPR> + <EXPR>

<EXPR> ? <EXPR> * <EXPR>

<EXPR> ? (<EXPR>)

<EXPR> ? a

Build a parse tree for a + a * a



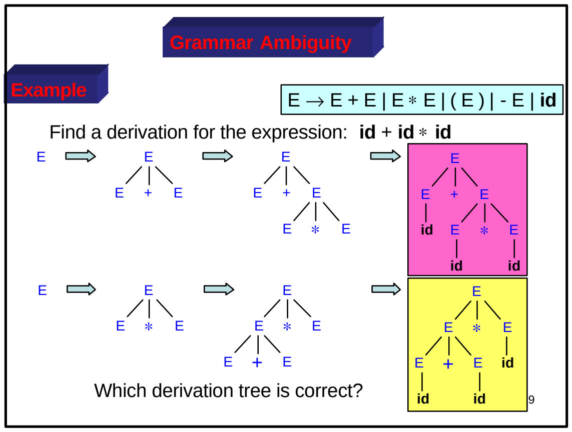
Grammar Ambiguity

Inherently Ambiguous

- Languages that can only be generated by ambiguous grammars are **inherently ambiguous**.
- Example 5.13: L = $\{a^nb^nc^m\} \cup \{a^nb^mc^m\}$.

$\boxed{L=\{\,a^ib^jc^k \;\mid\; i=j \,\vee\, j=k\}}$

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



Grammar Ambiguity

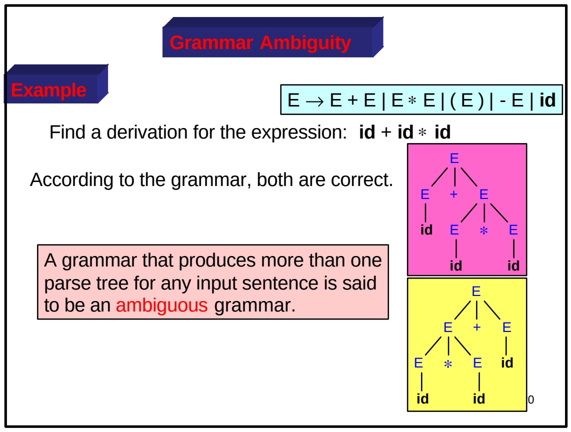
Example

E → E + E | E ∗ E | (E) | - E | **id**

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

Which derivation tree is correct?

9



Grammar Ambiguity

Example

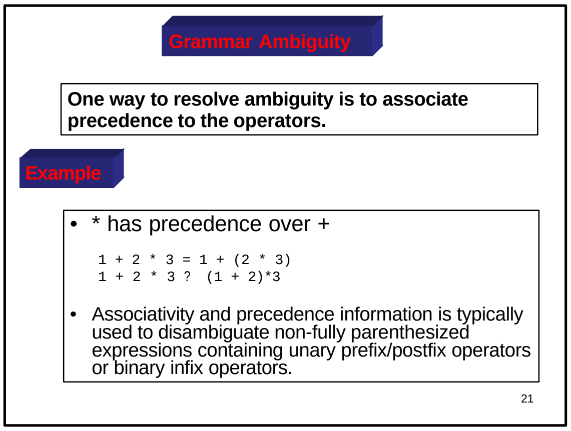
E → E + E | E ∗ E | (E) | - E | **id**

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.

0



Grammar Ambiguity

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

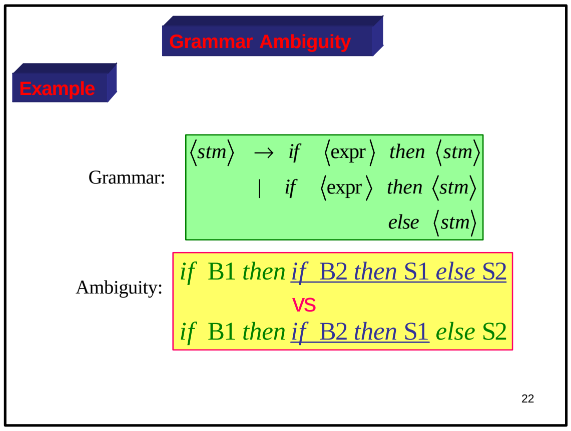
Example

- * 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.

21



Grammar Ambiguity

Example

Grammar:

$$\begin{aligned} \langle stm \rangle &\rightarrow \textit{if}\ \langle \text{expr} \rangle\ \textit{then}\ \langle stm \rangle \\ &\mid\ \textit{if}\ \langle \text{expr} \rangle\ \textit{then}\ \langle stm \rangle\ \textit{else}\ \langle stm \rangle \end{aligned}$$

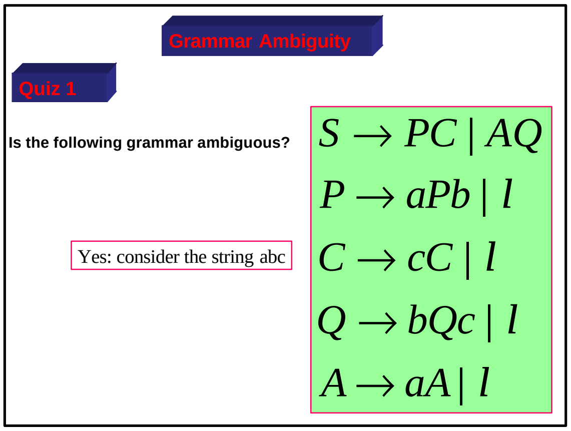
Ambiguity:

if B1 *then* *if* B2 *then* S1 *else* S2

VS

if B1 *then* *if* B2 *then* S1 *else* S2

22



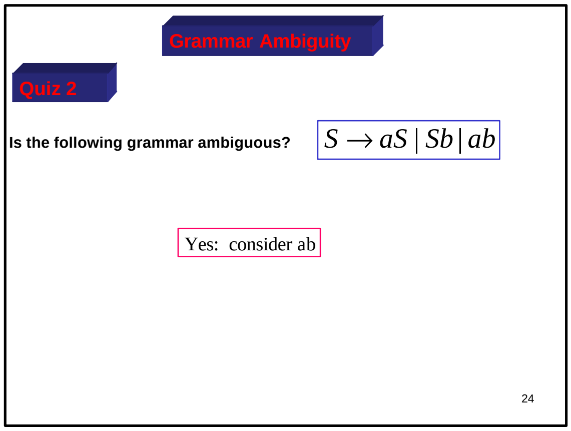
Grammar Ambiguity

Quiz 1

Is the following grammar ambiguous?

$$\begin{aligned} &S \rightarrow PC \mid AQ \\ &P \rightarrow aPb \mid l \\ &C \rightarrow cC \mid l \\ &Q \rightarrow bQc \mid l \\ &A \rightarrow aA \mid l \end{aligned}$$

Yes: consider the string abc



Grammar Ambiguity

Quiz 2

Is the following grammar ambiguous?

$S \rightarrow aS \mid Sb \mid ab$

Yes: consider ab

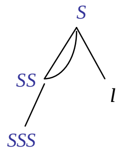
24

Grammar Ambiguity

Quiz

Is the following grammar ambiguous? $S \rightarrow SS \mid l$

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 ? ax with
A∈ V, a∈ T, x∈ V* and any pair (A,a) occurs at most once.

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

•Example: Take the s-grammar S ? aS|bSS|c with aabcc:
S ⇒ aS ⇒ aaS ⇒ aabSS ⇒ aabcS ⇒ aabcc.

Quiz: is the grammar S ? 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

Even though we can’t get every grammar into right-linear form, or *in general* even get rid of ambiguity, there is an especially simple form that general CFG’s can be converted into:

Chomsky Normal Form

Definition 6.4: A CFG is in **Chomsky normal form** if and only if all production rules are of the form
A → BC
or A → x
with variables A,B,C∈ V and x∈ T.
(Sometimes rule S→? is also allowed.)
CFGs in CNF can be parsed in time O(|w|^3).

Named after Noam Chomsky who in the 60s made seminal contributions to the field of theoretical linguistics. (cf. Chomsky hierarchy of languages).

Chomsky Normal Form CNF

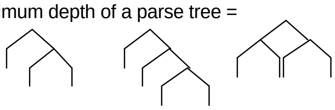
Noam Chomsky came up with an especially simple type of context free grammars which is able to capture all context free languages.

Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.

Chomsky Normal Form CNF

Significance of CNF

- Length of derivation of a string of length *n* in CNF = (2*n*-1)
  (*Cf.* Number of nodes of a strictly binary tree with *n*-leaves)
- Maximum depth of a parse tree = $n \lceil \log_2 n \rceil + 1$
- Minimum depth of a parse tree =



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:

$A \to BC$ (dyadic variable productions)

$A \to a$ (unit terminal productions)

$S \to l$ (? for empty string sake only)

where $B, C \in V - \{S\}$

Where *S* is the start variable, *A,B,C* are variables and *a* is a terminal. Thus epsilons 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.

Chomsky Normal Form CNF

CFG→ CNF: Construction

- Obtain an equivalent grammar that does not contain λ-rules, chain rules, and useless variables.
- Apply following conversion on rules of the form: $A \to bBcC$

⇩

$A \to PQ \qquad P \to b$

$Q \to BR \qquad R \to WC$

$W \to c$

Chomsky Normal Form CNF

CFG→ CNF: Construction

Converting a general grammar into Chomsky Normal Form works in four steps:

1. Ensure that the start variable doesn't appear on the right hand side of any rule.
2. Remove all ?-rules productions, except from start variable.
3. Remove unit variable productions of the form *A* → *B* where *A* and *B* are variables.
4. Add variables and dyadic variable rules to replace any longer non-dyadic or non-variable productions

Chomsky Normal Form CNF

CFG→ CNF: Example 1

Let's see how this works on the following example grammar:

S→? | a | b | aSa | bSb

Chomsky Normal Form CNF

CFG→ CNF: Example 1

1. Start Variable

Ensure that start variable doesn't appear on the right hand side of any rule.

S'→S

S→? | a | b | aSa | bSb

Chomsky Normal Form CNF

CFG→ CNF: Example 1

2. Remove ?-rules

Remove all epsilon productions, except from start variable.

S'→S | ?

S→ ? | a | b | aSa | bSb | aa | bb

Chomsky Normal Form CNF

CFG→ CNF: Example 1

3. Remove variable units

Remove unit variable productions of the form *A → B.*

S'→S | ? | a | b | aSa | bSb | aa | bb

S→ ? | a | b | aSa | bSb | aa | bb

Chomsky Normal Form CNF

CFG→ CNF: Example 1

4. Longer production rules

Add variables and dyadic variable rules to replace any longer productions.

S'→ ? | a | b | aSa | bSb | aa | bb AB|CD|AA|CC

S→a | b | aSa | bSb | aa | bb AB|CD|AA|CC

A→ a

B→ SA

C→ b

D→SC

Chomsky Normal Form CNF

CFG→ CNF: Example 1

5. Result

CFG

S→? | a | b | aSa | bSb

CNF

S'→ ? | a | b | AB | CD | AA | CC

S→ a | b | AB | CD | AA | CC

A→ a

B→ SA

C→ b

D→SC

CFG→ CNF: Example 2

1. Add a new start variable S_0 (and add the rule S_0 ? S)

2. Remove all A ? ? rules (where A is not S_0)

For each occurrence of A on right hand side of a rule, add a new rule with the occurrence deleted

If we have the rule B ? A, add B ? ?, unless we have previously removed B ? ?

3. Remove unit rules A ? B

Whenever B ? w appears, add the rule A ? w unless this was a unit rule previously removed

S_0 ? S

S ? 0S1

S ? T#T

S ? T

T ? ?

S ? T#

S ? #T

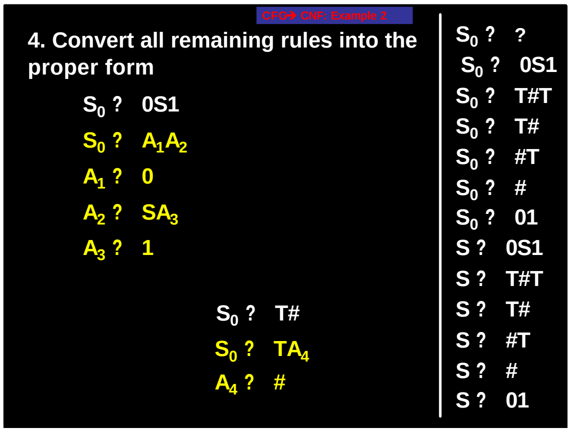
S ? #

S ? ?

S ? 01

S_0 ? 0S1

S_0 ? ?



CFG→ CNF: Example 2

4. Convert all remaining rules into the proper form

S_0 ? 0S1

S_0 ? A_1A_2

A_1 ? 0

A_2 ? SA_3

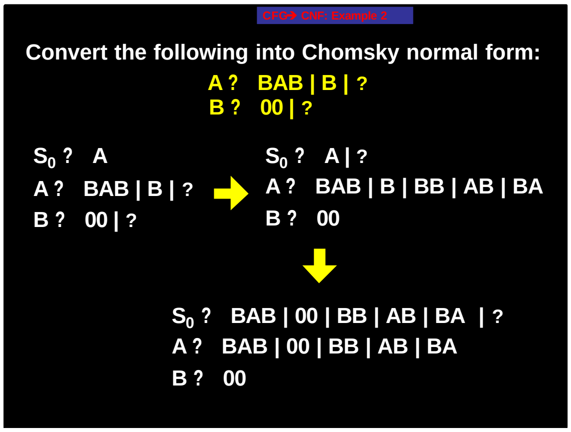
A_3 ? 1

S_0 ? T#

S_0 ? TA_4

A_4 ? #

S_0 ? ?
S_0 ? 0S1
S_0 ? T#T
S_0 ? T#
S_0 ? #T
S_0 ? #
S_0 ? 01
S ? 0S1
S ? T#T
S ? T#
S ? #T
S ? #
S ? 01



CFG→ CNF: Example 2

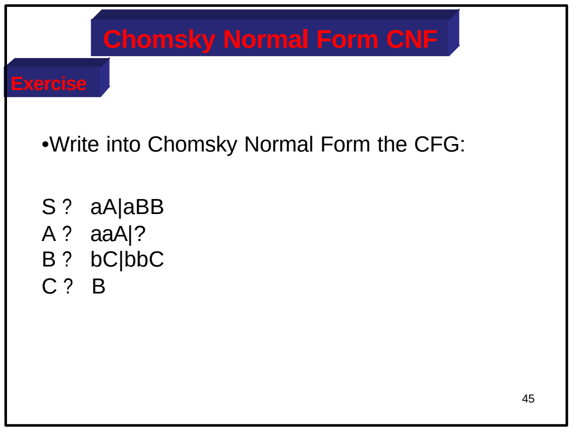
Convert the following into Chomsky normal form:

A ? BAB | B | ?
B ? 00 | ?

S_0 ? A
A ? BAB | B | ?
B ? 00 | ?

S_0 ? A | ?
A ? BAB | B | BB | AB | BA
B ? 00

S_0 ? BAB | 00 | BB | AB | BA | ?
A ? BAB | 00 | BB | AB | BA
B ? 00



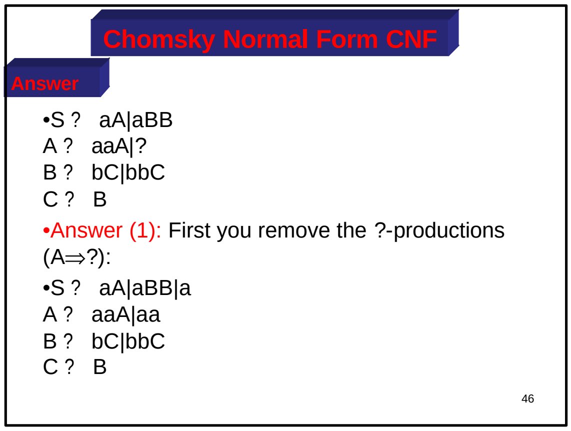
Chomsky Normal Form CNF

Exercise

- Write into Chomsky Normal Form the CFG:

S ? aA|aBB
A ? aaA|?
B ? bC|bbC
C ? B

45

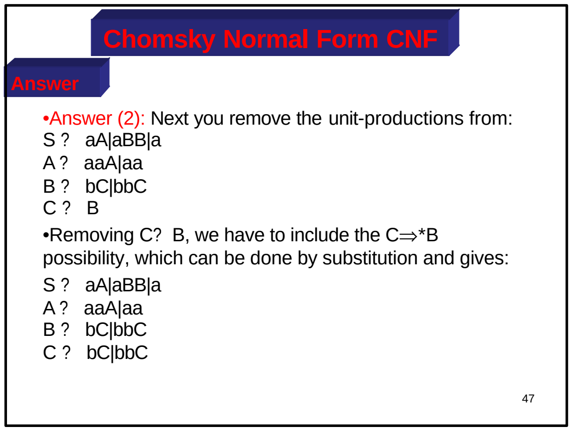


Chomsky Normal Form CNF

Answer

- S ? aA|aBB
A ? aaA|?
B ? bC|bbC
C ? B
- Answer (1): First you remove the ?-productions (A⇒?):
- S ? aA|aBB|a
A ? aaA|aa
B ? bC|bbC
C ? B

46

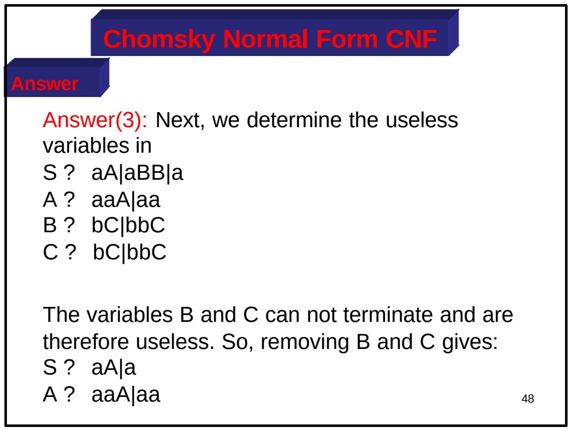


Chomsky Normal Form CNF

Answer

- Answer (2): Next you remove the unit-productions from:
S ? aA|aBB|a
A ? aaA|aa
B ? bC|bbC
C ? B
- Removing C? B, we have to include the C⇒*B possibility, which can be done by substitution and gives:
S ? aA|aBB|a
A ? aaA|aa
B ? bC|bbC
C ? bC|bbC

47



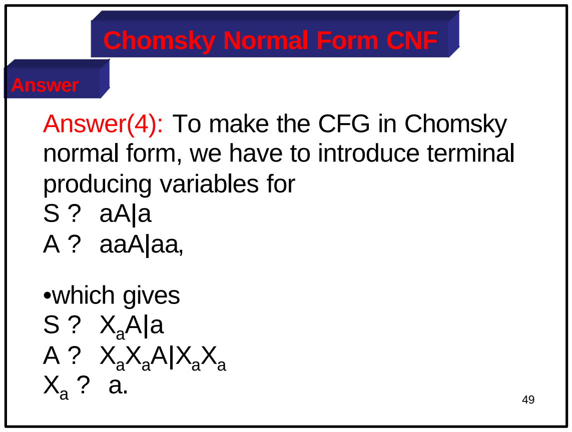
Chomsky Normal Form CNF

Answer

Answer(3): Next, we determine the useless variables in
S ? aA|aBB|a
A ? aaA|aa
B ? bC|bbC
C ? bC|bbC

The variables B and C can not terminate and are therefore useless. So, removing B and C gives:
S ? aA|a
A ? aaA|aa

48



Chomsky Normal Form CNF

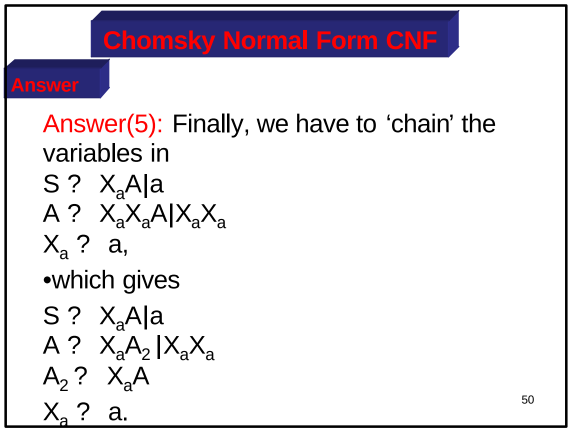
Answer

Answer(4): To make the CFG in Chomsky normal form, we have to introduce terminal producing variables for

S ? aA|a
A ? aaA|aa,

•which gives

S ? X_aA|a
A ? X_aX_aA|X_aX_a
X_a ? a.



Chomsky Normal Form CNF

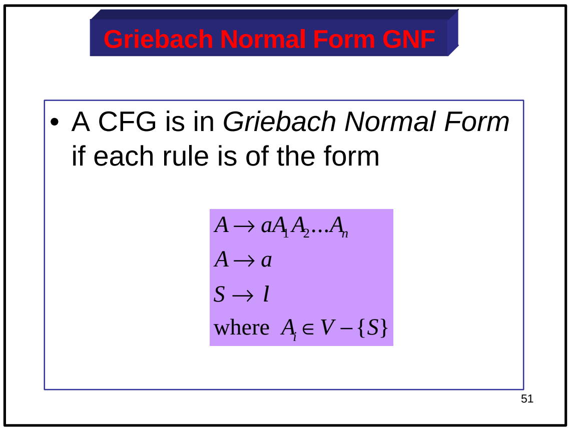
Answer

Answer(5): Finally, we have to 'chain' the variables in

S ? X_aA|a
A ? X_aX_aA|X_aX_a
X_a ? a,

•which gives

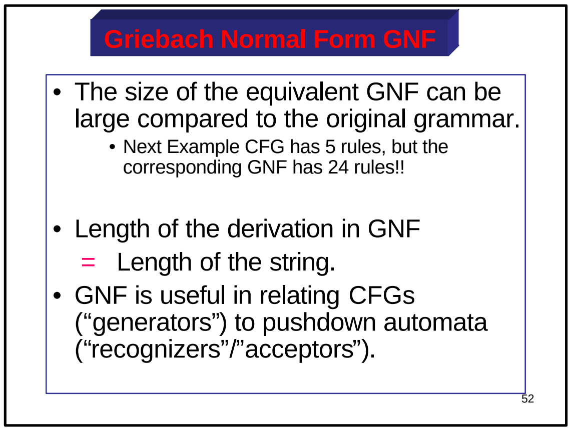
S ? X_aA|a
A ? X_aA_2 |X_aX_a
A_2 ? X_aA
X_a ? a.



Griebach Normal Form GNF

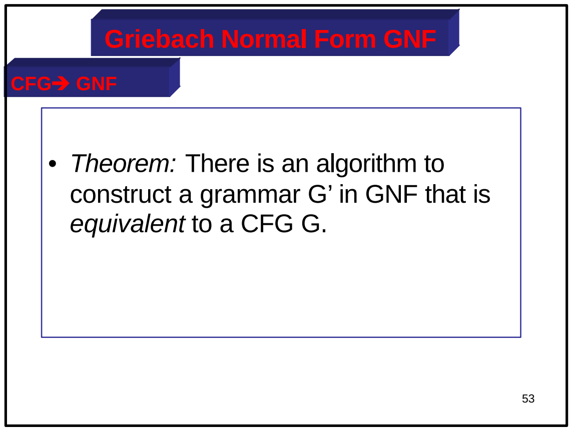
- A CFG is in *Griebach Normal Form* if each rule is of the form

$$A \rightarrow aA_1A_2 \ldots A_n$$
$$A \rightarrow a$$
$$S \rightarrow l$$
$$\text{where } A_i \in V - \{S\}$$



Griebach Normal Form GNF

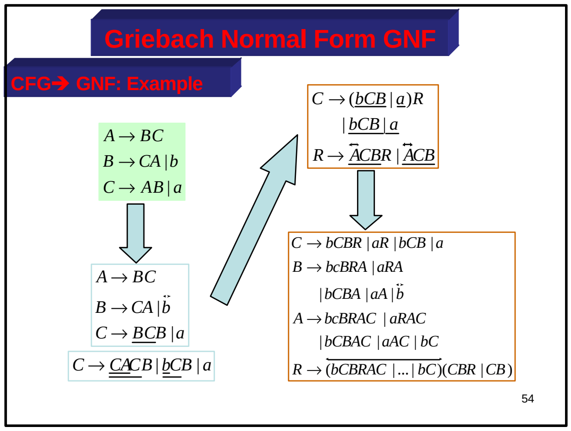
- The size of the equivalent GNF can be large compared to the original grammar.
  - Next Example CFG has 5 rules, but the corresponding GNF has 24 rules!!
- Length of the derivation in GNF
  = Length of the string.
- GNF is useful in relating CFGs ("generators") to pushdown automata ("recognizers"/"acceptors").



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.



Griebach Normal Form GNF

CFG➔ GNF: Example

$$A \rightarrow BC$$
$$B \rightarrow CA \mid b$$
$$C \rightarrow AB \mid a$$

$$A \rightarrow BC$$
$$B \rightarrow CA \mid \overleftrightarrow{b}$$
$$C \rightarrow \underline{BC}B \mid a$$

$$C \rightarrow \underline{\underline{CA}}CB \mid \underline{b}CB \mid a$$

$$C \rightarrow (\underline{bCB} \mid \underline{a})R$$
$$\mid \underline{bCB \mid a}$$
$$R \rightarrow \underline{\overleftrightarrow{A}CB}R \mid \underline{\overleftrightarrow{A}CB}$$

$$C \rightarrow bCBR \mid aR \mid bCB \mid a$$
$$B \rightarrow bcBRA \mid aRA$$
$$\mid bCBA \mid aA \mid \overleftrightarrow{b}$$
$$A \rightarrow bcBRAC \mid aRAC$$
$$\mid bCBAC \mid aAC \mid bC$$
$$R \rightarrow \overleftrightarrow{(bCBRAC \mid \ldots \mid bC)}(CBR \mid CB)$$

Context Sensitive Grammar

An even more general form of grammars exists. In general, a non-context free grammar is one in which whole mixed variable/terminal substrings are replaced at a time. For example with $\Sigma = \{a,b,c\}$ consider:

S → **?** | *ASBC*	*aB* → *ab*
A → *a*	*bB* → *bb*
CB → *BC*	*bC* → *bc*
	cC → *cc*

For technical reasons, when length of LHS always ≤ length of RHS, these general grammars are called ***context sensitive***.

Context Sensitive Grammar (CSG)

Example

Find the language generated by the CSG:

S → ? | *ASBC*
A → *a*
CB → *BC*
aB → *ab*
bB → *bb*
bC → *bc*
cC → *cc*

Context Sensitive Grammar (CSG)

Example

Answer is $\{a^n b^n c^n\}$.

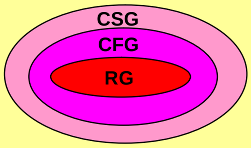
In a future class we'll see that this language is not context free. Thus perturbing context free-ness by allowing context sensitive productions expands the class.

Relations between Grammars

So far we studied 3 grammars:

1. Regular Grammars (RG)
2. Context Free Grammars (CFG)
2. Context Sensitive Grammars (CSG)

The relation between these 3 grammars is as follow:



Grammar Applications

Programming Languages

Programming languages are often defined as Context Free Grammars in **Backus-Naur Form** (**BNF**).

Example:

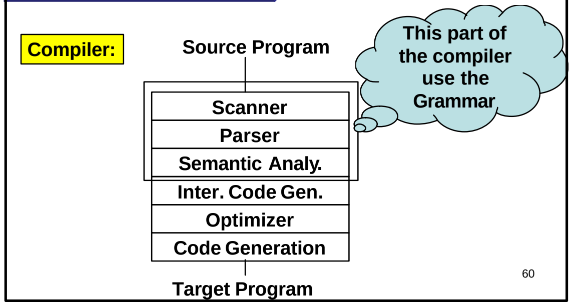
```
<if_statement> ::= IF <expression><then_clause><else_clause>
<expression> ::= <term> | <expression>+<term>
<term> ::= <factor>|<term>*<factor>
```

The variables as indicated by <a variable name>
The arrow ?   is replaces by ::=
Here, `IF`, + and * are terminals.

"Syntax Checking" is checking if a program is an element of the CFG of the programming language.

Grammar Applications

Compiler Syntax Analysis

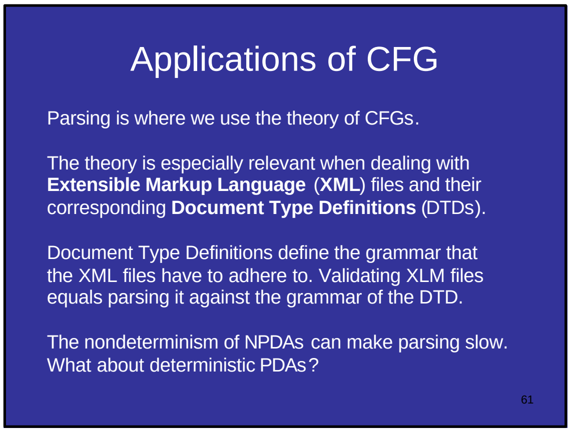

Compiler:

Source Program

Scanner
Parser
Semantic Analy.
Inter. Code Gen.
Optimizer
Code Generation

Target Program

This part of the compiler use the Grammar



Applications of CFG

Parsing is where we use the theory of CFGs.

The theory is especially relevant when dealing with **Extensible Markup Language** (**XML**) files and their corresponding **Document Type Definitions** (DTDs).

Document Type Definitions define the grammar that the XML files have to adhere to. Validating XLM files equals parsing it against the grammar of the DTD.

The nondeterminism of NPDAs can make parsing slow. What about deterministic PDAs?

61