

SYNTAX ANALYSIS

Backus-Naur Form.

- BNF stands for Backus-Naur Form or Backus Normal Form.
- A meta language is a language that is used to describe another language.
- BNF is a meta language (formal notations) for programming languages syntax.
- A production is a rule relating to a pair of strings, say α and β , specifying how one may be transformed into the other. This may be denoted
$$\alpha \rightarrow \beta.$$
- For simple theoretical grammars, upper case letters are used for non-terminals and lower case letters are used for terminals.
- For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions is to use the notations invented by Backus commonly called BNF.
- These notations were first introduced by Backus for describing ALGOL 58.
- These notations were later modified slightly by Peter Naur for the description of ALGOL 60.

Backus-Naur Form.

- In BNF:
 - A non-terminal and terminal are usually given some descriptive names.
 - Non-terminals symbols are written in angle brackets to distinguish it from a terminal symbol.
 - If there are multiple definitions for the same non-terminal symbol, then they can be written as single rule, separated from each by using (|) vertical bar which means logical OR.

Example.

$G = \{N, T, S, P\}$

$N = \{ \langle \text{sentence} \rangle, \langle \text{qualified noun} \rangle, \langle \text{noun} \rangle, \langle \text{pronoun} \rangle, \langle \text{verb} \rangle, \langle \text{adjective} \rangle \}$

$T = \{ \text{the}, \text{man}, \text{girl}, \text{boy}, \text{lecturer}, \text{he}, \text{she}, \text{drinks}, \text{sleeps},$
 $\text{mystifies}, \text{tall}, \text{thin}, \text{thirsty} \}$

$S = \langle \text{sentence} \rangle$

$P = \{$

$\langle \text{sentence} \rangle$	\rightarrow	the $\langle \text{qualified noun} \rangle$ $\langle \text{verb} \rangle$	(1)
		$\langle \text{pronoun} \rangle$ $\langle \text{verb} \rangle$	(2)
$\langle \text{qualified noun} \rangle$	\rightarrow	$\langle \text{adjective} \rangle$ $\langle \text{noun} \rangle$	(3)
$\langle \text{noun} \rangle$	\rightarrow	man girl boy lecturer	(4, 5, 6, 7)
$\langle \text{pronoun} \rangle$	\rightarrow	he she	(8, 9)
$\langle \text{verb} \rangle$	\rightarrow	talks listens mystifies	(10, 11, 12)
$\langle \text{adjective} \rangle$	\rightarrow	tall thin sleepy	(13, 14, 15)

$\}$

- Derive the “The sleepy boy listens” by using the above grammar.

Problems of a CFG.

- Three types of problems are mainly faced in a CFG.
 - Ambiguity.
 - Left Recursion.
 - Common Prefixes.
- Three problems must be removed from a CFG, otherwise the grammar will not work accurately.

Ambiguity.

- An ambiguous grammar is one that:
 - Produces more than one parse trees for the same sentence.
 - Produces more than one leftmost derivations or rightmost derivations for the same sentence.
- A grammar becomes ambiguous when a single non-terminal appears twice or more times on the L.H.S of the production rules in the grammar.
- If more than one parse trees can be produced for a sentence; then the compiler would not be able to generate the code uniquely.

Example.

- Consider the following grammar.

$$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$$
$$\langle \text{id} \rangle \rightarrow A \mid B \mid C$$
$$\begin{aligned} \langle \text{expr} \rangle \rightarrow & \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ & \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ & \mid (\langle \text{expr} \rangle) \\ & \mid \langle \text{id} \rangle \end{aligned}$$

- Now show that this grammar is ambiguous for the sentence $A = B + C * A$.

Parse Trees

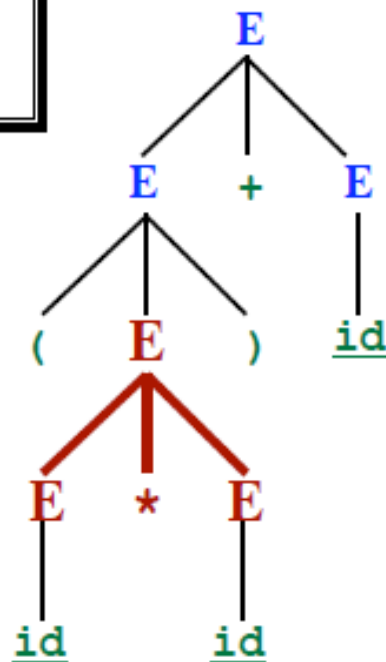
Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (\underline{id} * E) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$



1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Parse Trees

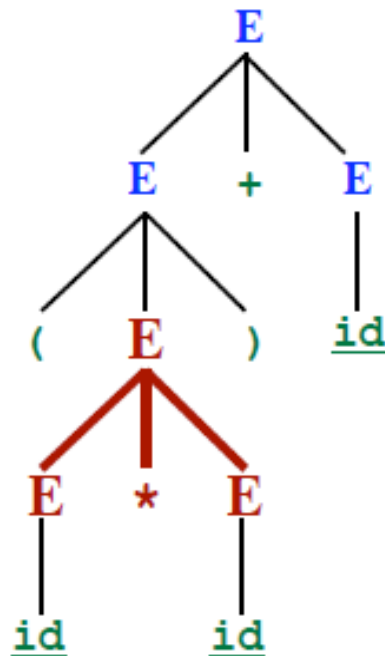
Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$



1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Parse Trees

Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

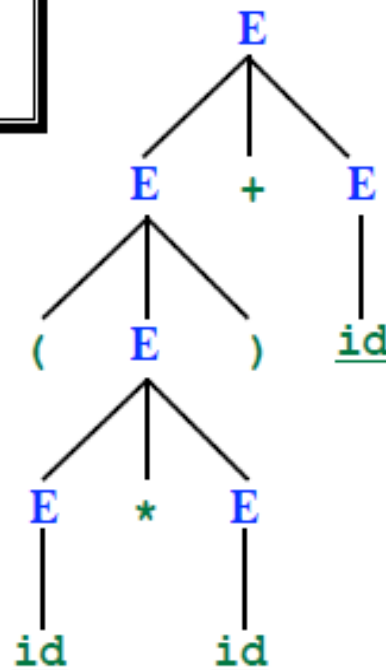
Leftmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (\underline{id} * E) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

Rightmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$



Given a leftmost derivation, we can build a parse tree.
Given a rightmost derivation, we can build a parse tree.

Leftmost Derivation of

(id*id)+id

Rightmost Derivation of

(id*id)+id



Same Parse Tree

Every parse tree corresponds to...

- A single, unique leftmost derivation
- A single, unique rightmost derivation

Ambiguity:

However, one input string may have several parse trees!!!

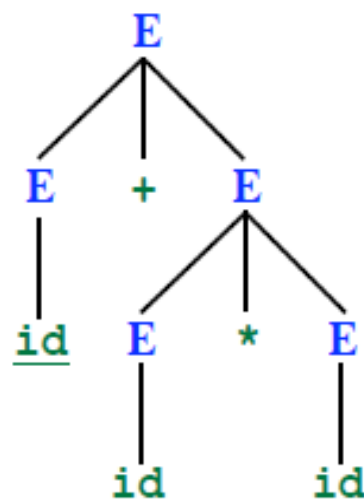
Therefore:

- Several leftmost derivations
- Several rightmost derivations

Ambiguous Grammars

Leftmost Derivation #1

E
 $\Rightarrow E + E$
 $\Rightarrow \underline{id} + E$
 $\Rightarrow \underline{id} + E * E$
 $\Rightarrow \underline{id} + \underline{id} * E$
 $\Rightarrow \underline{id} + \underline{id} * \underline{id}$

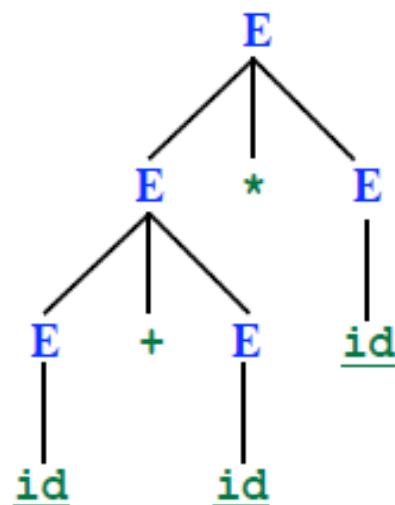


1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Input: $id + id * id$

Leftmost Derivation #2

E
 $\Rightarrow E * E$
 $\Rightarrow E + E * E$
 $\Rightarrow \underline{id} + E * E$
 $\Rightarrow \underline{id} + \underline{id} * E$
 $\Rightarrow \underline{id} + \underline{id} * \underline{id}$



Ambiguous Grammar

More than one Parse Tree for some sentence.

The grammar for a programming language may be ambiguous

Need to modify it for parsing.

Also: Grammar may be left recursive.

Need to modify it for parsing.

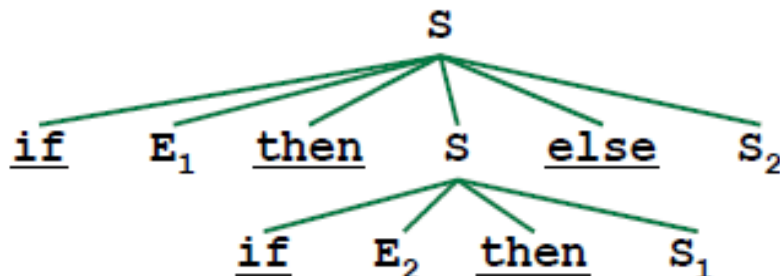
The Dangling “Else” Problem

This grammar is ambiguous!

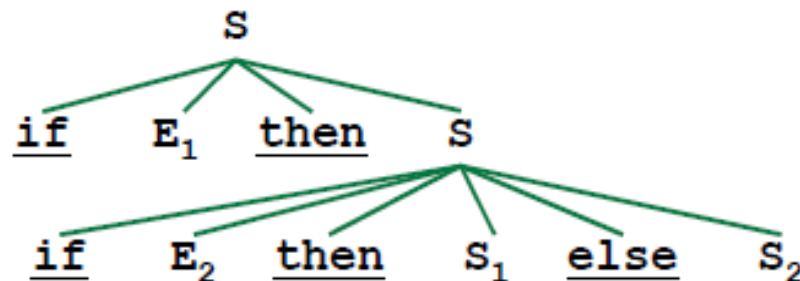
Stmt \rightarrow if Expr then Stmt
 \rightarrow if Expr then Stmt else Stmt
 \rightarrow ...Other Stmt Forms...

Example String: if E_1 then if E_2 then S_1 else S_2

Interpretation #1: if E_1 then (if E_2 then S_1) else S_2



Interpretation #2: if E_1 then (if E_2 then S_1 else S_2)



The Dangling “Else” Problem

Goal: “Match else-clause to the closest if without an else-clause already.”

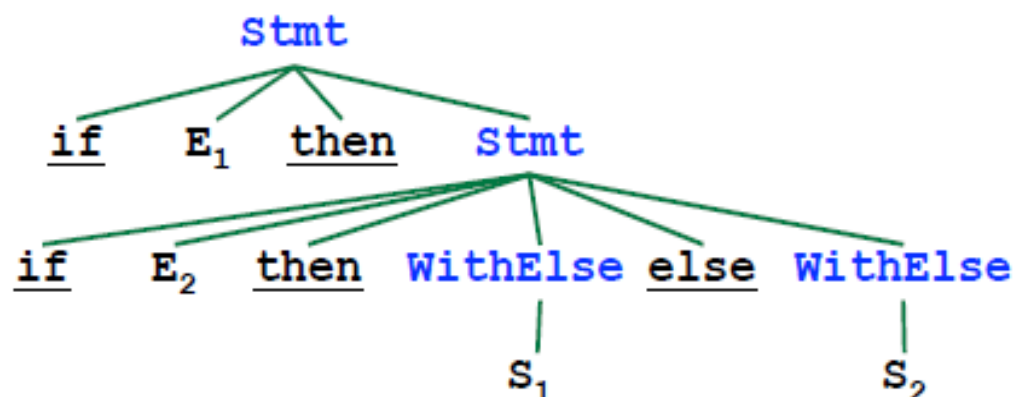
Solution:

Stmt → if Expr then Stmt
→ if Expr then WithElse else Stmt
→ ...Other Stmt Forms...

WithElse → if Expr then WithElse else WithElse
→ ...Other Stmt Forms...

Any Stmt occurring between then and else must have an else.
i.e., the Stmt must not end with “then Stmt”.

Interpretation #2: if E₁ then (if E₂ then S₁ else S₂)



The Dangling “Else” Problem

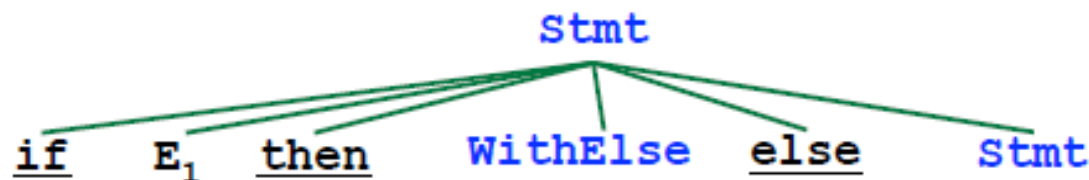
Goal: “Match else-clause to the closest if without an else-clause already.”

Solution:

Stmt → if Expr then Stmt
 → if Expr then WithElse else Stmt
 → ...Other Stmt Forms...
WithElse → if Expr then WithElse else WithElse
 → ...Other Stmt Forms...

Any Stmt occurring between then and else must have an else.
i.e., the Stmt must not end with “then Stmt”.

Interpretation #1: if E₁ then (if E₂ then S₁) else S₂



The Dangling “Else” Problem

Goal: “Match else-clause to the closest if without an else-clause already.”

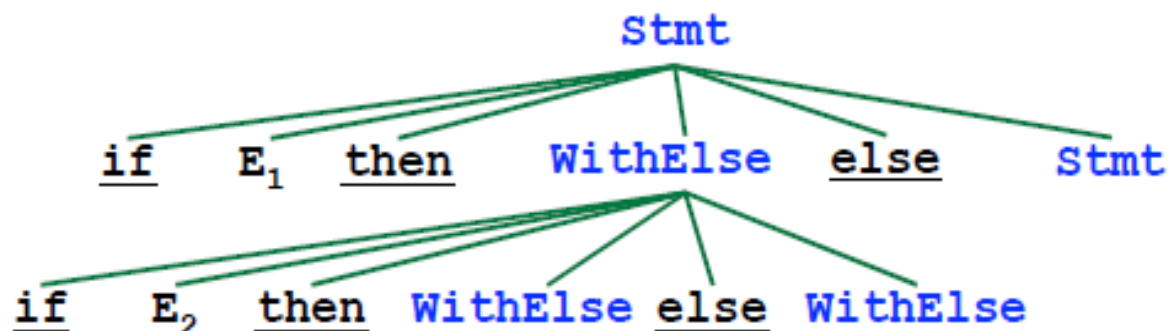
Solution:

Stmt → if Expr then Stmt
 → if Expr then WithElse else Stmt
 → ...Other Stmt Forms...

WithElse → if Expr then WithElse else WithElse
 → ...Other Stmt Forms...

Any Stmt occurring between then and else must have an else.
i.e., the Stmt must not end with “then Stmt”.

Interpretation #1: if E₁ then (if E₂ then S₁) else S₂



The Dangling “Else” Problem

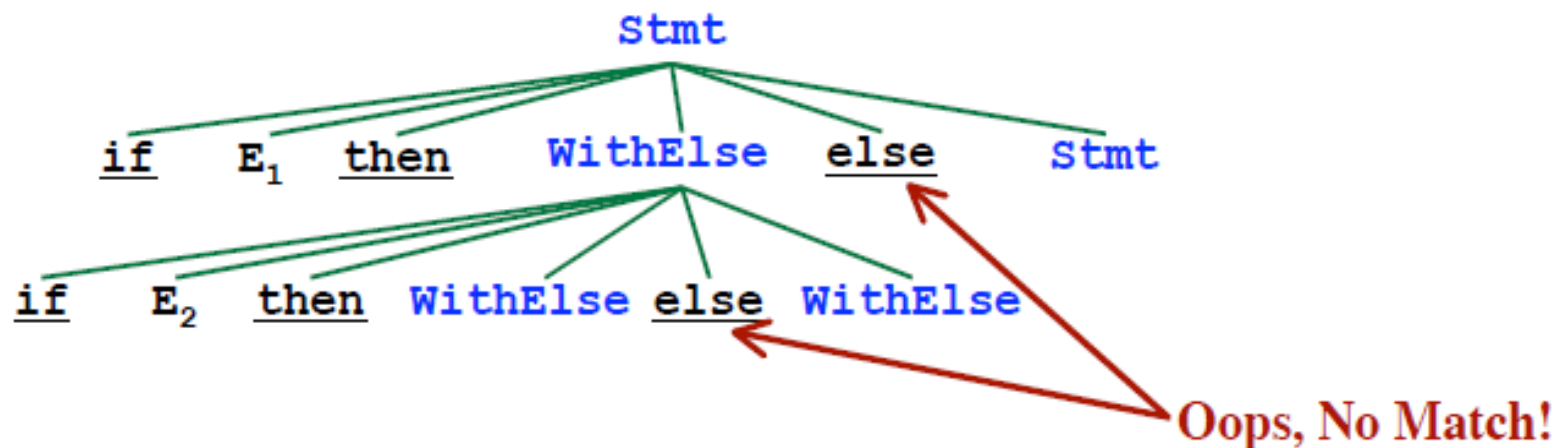
Goal: “Match else-clause to the closest if without an else-clause already.”

Solution:

Stmt → if Expr then Stmt
→ if Expr then WithElse else Stmt
→ ...Other Stmt Forms...
WithElse → if Expr then WithElse else WithElse
→ ...Other Stmt Forms...

Any Stmt occurring between then and else must have an else.
i.e., the Stmt must not end with “then Stmt”.

Interpretation #1: if E₁ then (if E₂ then S₁) else S₂



Null Production

Definition: The production of the form
nonterminal $\rightarrow \epsilon$
is said to be ***null production***.

Example: Consider the following CFG
 $S \rightarrow aA|bB|\epsilon$, $A \rightarrow aa|\epsilon$, $B \rightarrow aS$
Here $S \rightarrow \epsilon$ and $A \rightarrow \epsilon$ are null
productions.

Following is a note regarding the null
productions

Note

If a CFG has a null production, then it is possible to construct another CFG without null production accepting the same language with the exception of the word ϵ *i.e.* if the language contains the word ϵ then the new language cannot have the word ϵ .

Following is a method to construct a CFG without null production for a given CFG

Null Production continued

...

Method: Delete all the Null productions and add new productions *e.g.*

consider the following productions of a certain CFG $X \rightarrow aNbNa$, $N \rightarrow \epsilon$, delete the production $N \rightarrow \epsilon$ and using the production

$X \rightarrow aNbNa$, add the following new productions

$X \rightarrow aNba$, $X \rightarrow abNa$ and $X \rightarrow aba$

Thus the new CFG will contain the following productions $X \rightarrow aNba|abNa|aba|aNbNa$

Note: It is to be noted that $X \rightarrow aNbNa$ will still be included in the new CFG.

Nullable Production

Definition: A production is called ***nullable production*** if it is of the form

$N \rightarrow \epsilon$

or

there is a derivation that starts at N and leads to ϵ *i.e.*

$N_1 \rightarrow N_2, N_2 \rightarrow N_3, N_3 \rightarrow N_4, \dots, N_n \rightarrow \epsilon,$

where N, N_1, N_2, \dots, N_n are non terminals.

Following is an example

Example

Consider the following CFG

$S \rightarrow AA|bB, A \rightarrow aa|B, B \rightarrow aS | \epsilon$

Here $S \rightarrow AA$ and $A \rightarrow B$ are nullable productions, while $B \rightarrow \epsilon$ is null a production.

Following is an example describing the method to convert the given CFG containing null productions and nullable productions into the one without null productions

Example

Consider the following CFG

$$S \rightarrow XaY|YY|aX|Z YX$$
$$X \rightarrow Za|bZ|ZZ|Yb$$
$$Y \rightarrow Ya|XY|\epsilon$$
$$Z \rightarrow aX|YYY$$

It is to be noted that in the given CFG, the productions $S \rightarrow YY$, $X \rightarrow ZZ$, $Z \rightarrow YYY$ are Nullable productions, while $Y \rightarrow \epsilon$ is Null production.

Example continued ...

Here the method of removing null productions, as discussed earlier, will be used along with replacing nonterminals corresponding to nullable productions like nonterminals for null productions are replaced.

Thus the required CFG will be

$$S \rightarrow XaY|Xa|aY|a|YY|Y|aX|Z YX|YX|ZX|ZY$$
$$X \rightarrow Za|a|bZ|b|ZZ|Z|Yb$$
$$Y \rightarrow Ya|a|XY|X|Y$$
$$Z \rightarrow aX|a|YYY|YY|Y,$$

Following is another example

Example

Consider the following CFG

$S \rightarrow XY, X \rightarrow Zb, Y \rightarrow bW$

$Z \rightarrow AB, W \rightarrow Z, A \rightarrow aA|bA|\epsilon$

$B \rightarrow Ba|Bb|\epsilon.$

Here $A \rightarrow \epsilon$ and $B \rightarrow \epsilon$ are null productions, while $Z \rightarrow AB, W \rightarrow Z$ are nullable productions. The new CFG after, applying the method, will be

Example continued ...

$S \rightarrow XY$

$X \rightarrow Zb|b$

$Y \rightarrow bW|b$

$Z \rightarrow AB|A|B$

$W \rightarrow Z$

$A \rightarrow aA|a|bA|b$

$B \rightarrow Ba|a|Bb|b$

Note

While adding new productions all Nullable productions should be handled with care. All Nullable productions will be used to add new productions, but only the Null production will be deleted.

Unit production

Unit production: The productions of the form nonterminal \rightarrow one nonterminal, is called the ***unit production***. Following is an example showing how ***to eliminate the unit productions from a given CFG.***

Unit production continued

...

Example: Consider the following CFG

$S \rightarrow A|bb,$

$A \rightarrow B|b,$

$B \rightarrow S|a$

Separate the unit productions from the nonunit productions as shown below

unit prods. nonunit prods.

$S \rightarrow A$

$S \rightarrow bb$

$A \rightarrow B$

$A \rightarrow b$

$B \rightarrow S$

$B \rightarrow a$

Example continued ...

$S \rightarrow A$ gives $S \rightarrow b$ (using $A \rightarrow b$)

$S \rightarrow A \rightarrow B$ gives $S \rightarrow a$ (using $B \rightarrow a$)

$A \rightarrow B$ gives $A \rightarrow a$ (using $B \rightarrow a$)

$A \rightarrow B \rightarrow S$ gives $A \rightarrow bb$ (using $S \rightarrow bb$)

$B \rightarrow S$ gives $B \rightarrow bb$ (using $S \rightarrow bb$)

$B \rightarrow S \rightarrow A$ gives $B \rightarrow b$ (using $A \rightarrow b$)

Thus the new CFG will be

$S \rightarrow a|b|bb, A \rightarrow a|b|bb, B \rightarrow a|b|bb.$

Which generates the finite language $\{a,b,bb\}.$