Department of Computer Science and Engineering

PES University, Bangalore, India

UE23CS243A: Automata Formal Language and Logic

Prakash C O, Associate Professor, Department of CSE

Ambiguous CFGs and Inherently ambiguous CFLs

Ambiguous Grammar

- A context free grammar is called ambiguous if there exists **more than one leftmost derivation (LMD)** or **more than one rightmost derivation (RMD) for a string** which is generated by grammar. **Or**

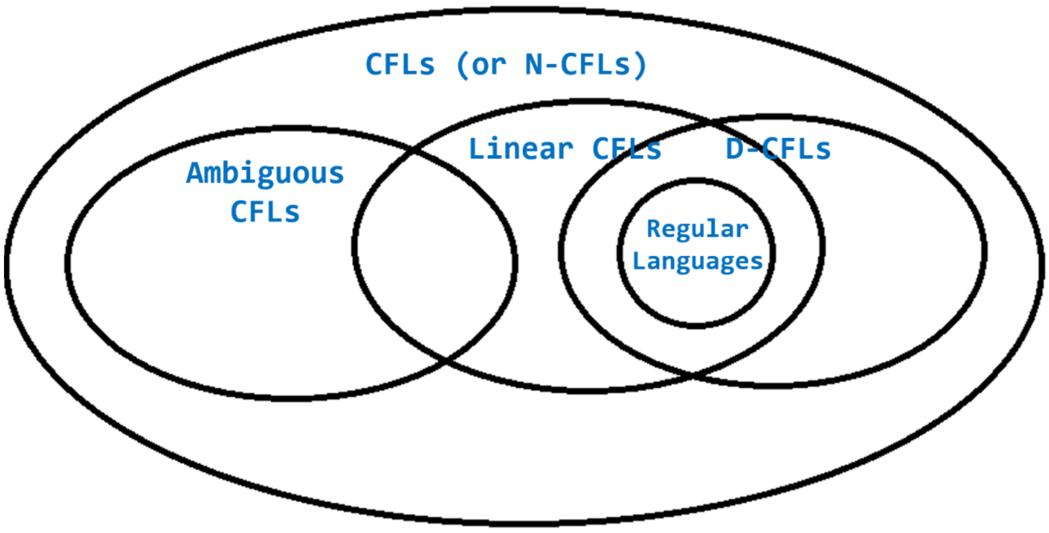
- A context free grammar is called ambiguous if there exists more than one parse tree for a string which is generated by grammar. **Or**

- A CFG is ambiguous if one or more terminal strings have multiple leftmost derivations from the start symbol. Equivalently: multiple rightmost derivations, or multiple parse trees.

Note:

1) **An ambiguous grammar is a context free grammar (CFG)** for which there exists a string that can have more than one leftmost derivation or rightmost derivation or parse tree.
2) Every non-empty context-free language (CFL) admits an ambiguous grammar by introducing a duplicate production rule.
3) **A language that only admits ambiguous grammars** is called an **inherently ambiguous language**.

4) **Deterministic context-free grammars (DCFGs) are always unambiguous**, and are an important subclass of unambiguous grammars; there are non-deterministic unambiguous grammars, however.



Important Notes:

Note-1:

If N parse trees exist for any string w of grammar G, then there will be

- N-corresponding leftmost derivations and
- N-corresponding rightmost derivations.

Note-2:

For ambiguous grammars, more than one leftmost derivation and more than one rightmost derivation exist for at least one string.

Leftmost derivation and rightmost derivation may be same or different.

Note-3:

For unambiguous grammars, a unique leftmost derivation and a unique rightmost derivation exist for all the strings.

Leftmost derivation and rightmost derivation represent the same parse tree.

Example: Consider the unambiguous grammar

E → E + T | T
T → T * F | F
F → (E) | num | id

Let us consider a string w = **`id+id*id`**

Leftmost derivation

E ⇒ E + T
⇒ T + T
⇒ F + T
⇒ id + T
⇒ id + T * F
⇒ id + F * F
⇒ id + id * F
⇒ id + id * id

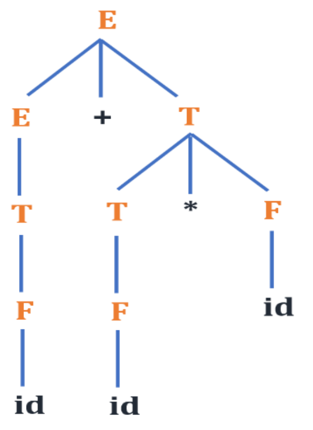
Rightmost derivation

E ⇒ E + T
⇒ E + T * F
⇒ E + T * id
⇒ E + F * id
⇒ E + id * id
⇒ T + id * id
⇒ F + id * id
⇒ id + id * id

The Parse tree for string **id+id*id** is

A parse tree is a labelled tree representation of a derivation.

- **The interior nodes are nonterminals.**
- **The leaf nodes are terminals**



For both the Leftmost derivation and Rightmost derivation, the parse tree is the same.

Note: For unambiguous grammars, it is not possible to get two different parse trees for the same input string either using Leftmost derivations and/or using Rightmost derivations.

Note-4:

Leftmost derivation and rightmost derivation of a string is exactly same in some grammars (i.e., in linear grammars).

In fact, there may exist a grammar in which leftmost derivation and rightmost derivation is exactly same for all the strings.

Example: Consider the linear grammar

$$S \rightarrow aS \mid \varepsilon$$

The language generated by this grammar is **L = { a^n | n ≥ 0 }** and Regex = **a***

All the strings generated from this grammar have their leftmost derivation and rightmost derivation exactly same.

Let us consider a string **w = aaa**.

Leftmost Derivation:

S ⇒ a**S**
⇒ aa**S** (Using S → aS)
⇒ aaa**S** (Using S → aS)
⇒ aaa (Using S → ε)

Rightmost Derivation:

S ⇒ a**S**
⇒ aa**S** (Using S → aS)
⇒ aaa**S** (Using S → aS)
⇒ aaa (Using S → ε)

Clearly, **Leftmost derivation = Rightmost derivation**

Similar is the case for all other strings.

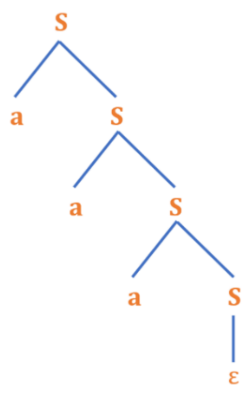
Note-5:

For a given parse tree, we may have its leftmost derivation exactly same as rightmost derivation.

Example: Consider the linear grammar

$$S \rightarrow aS \mid \varepsilon$$

Let us consider a parse tree for the string '**aaa**'



Leftmost Derivation:

S ⇒ a**S**
⇒ aa**S** (Using S → aS)
⇒ aaa**S** (Using S → aS)
⇒ aaa (Using S → ε)

Rightmost Derivation:

S ⇒ a**S**
⇒ aa**S** (Using S → aS)
⇒ aaa**S** (Using S → aS)
⇒ aaa (Using S → ε)

Clearly, for a given parse tree, the leftmost derivation is exactly same as the rightmost derivation.

Note-6:

If for all the strings of a grammar (i.e., in linear grammars), leftmost derivation is exactly same as rightmost derivation, then that grammar may be ambiguous or unambiguous.

Note-7:

There may exist derivations for a string which are neither leftmost nor rightmost.

Example:

Consider the following grammar

S → ABC
A → a
B → b
C → c

Consider a string **w = abc**

Total 6 derivations exist for string w.

The following 3 derivations are neither leftmost nor rightmost

Derivation-01:	**Derivation-02:**	**Derivation-03:**
$S \Rightarrow ABC$	$S \Rightarrow ABC$	$S \Rightarrow ABC$
$\Rightarrow aBC$ (Using $A \to a$)	$\Rightarrow AbC$ (Using $B \to b$)	$\Rightarrow AbC$ (Using $B \to b$)
$\Rightarrow aBc$ (Using $C \to c$)	$\Rightarrow abC$ (Using $A \to a$)	$\Rightarrow Abc$ (Using $C \to c$)
$\Rightarrow$ **abc** (Using $B \to b$)	$\Rightarrow$ **abc** (Using $C \to c$)	$\Rightarrow$ **abc** (Using $A \to a$)

Ambiguous Grammar Examples:

Example 1: $S \to S \mid \varepsilon$

The simplest example is the above ambiguous grammar for the trivial language that consists of only the empty string.

$S \to S \mid \varepsilon$

meaning that the nonterminal S can be derived to either itself again, or to the empty string. Thus, the empty string has leftmost (or rightmost) derivations of length 1, 2, 3, and indeed of any length, depending on how many times the rule $S \to S$ is used.

Consider a string w = **ε**

First Leftmost derivation
$S \Rightarrow \varepsilon$

Second Leftmost derivation
$S \Rightarrow S \Rightarrow S \Rightarrow S \Rightarrow \varepsilon$

This language also has an unambiguous grammar, consisting of a single production rule:

$S \to \varepsilon$

meaning that the unique production can produce only the empty string, which is the unique string in the language.

In the same way, **any grammar for a non-empty language can be made ambiguous by adding duplicates.**

Example 2: Consider the below grammars:

1) **$S \to aS \mid a \mid \varepsilon$**
2) **$S \to Sa \mid a \mid \varepsilon$**
3) **$S \to Sa \mid aS \mid \varepsilon$**

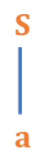
These are all examples of ambiguous grammars for the language $L = \{ a^n \mid n \geq 0 \}$

Consider a string '**a**' for grammar **$S \to aS \mid a \mid \varepsilon$**

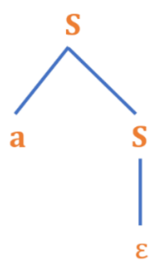
First Leftmost derivation
$S \Rightarrow a$

Second Leftmost derivation
$S \Rightarrow aS$
$\Rightarrow a$

Parse trees:



Parse tree-01



Parse tree-02

Since two different LMDs and two different parse trees exist for the string 'a', so the grammar is ambiguous.

The language L = {a^n | n≥0} of the above grammar also has an unambiguous grammar:

S → aS | ε or **S → Sa | ε**

Here, each string has unique leftmost derivation and unique rightmost derivation.

Why Ambiguity in Grammar is not suitable for compiler construction?

- Ambiguity in grammar is not suitable for compiler construction because it can make it difficult for a compiler to determine which parse tree is correct for a given input string. This is because ambiguous grammar can produce multiple parse trees for the same input string.
- Ambiguous grammar is not parsed by any parsers**.**

Recognizing ambiguous grammars

1) How to find out whether the given grammar is ambiguous?

There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

Here are some ways to determine if a grammar is ambiguous:

- **Check for multiple derivations**

  If a string can be derived from more than one left most derivation or more than one right most derivation, then the grammar is ambiguous.
- **Check for multiple parse trees**

  If an input string can have more than one parse tree, then the grammar is ambiguous

2) How to convert ambiguous grammar to unambiguous grammar?

- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.
- If the grammar has ambiguity, no method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.
- Ambiguous grammar (that generates arithmetic expressions / boolean expressions / regular expressions having operators) can be transformed into unambiguous grammar by using some methods like the definition of operator precedence and associativity, by which only one interpretation will be correct.

Ambiguity removal in Context Free Grammars

Example 1: Arithmetic Expression Grammar

Here is a CFG that generates arithmetic expressions (having addition, subtraction, multiplication, division and exponentiation operators).

E → E + E
E → E - E
E → E * E
E → E / E
E → E ^ E
E → (E)| num | id

What these production rules tell us is that the result of any operation, for example, multiplication, is also an expression (denoted, E).

From the above grammar, string "**id+id-id**" can be derived in two ways using Leftmost derivations:

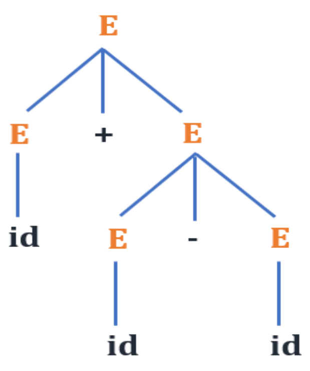
First Leftmost derivation

E ⇒ E + E
⇒ id + E
⇒ id + E - E
⇒ id + id - E
⇒ **id + id - id**

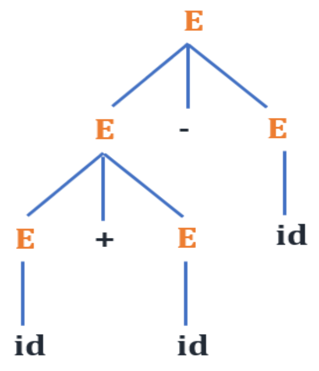
Second Leftmost derivation

E ⇒ E - E
⇒ E + E - E
⇒ id + E - E
⇒ id + id - E
⇒ **id + id - id**

Parse Trees



Parse tree-01



Parse tree-02

From the above grammar, string "**id+id-id**" can be derived in two ways using Rightmost derivations:

First Rightmost derivation

E ⇒ E + E
⇒ E + E - E
⇒ E + E - id
⇒ E + id - id
⇒ **id + id - id**

Second Rightmost derivation

E ⇒ E - E
⇒ E - id
⇒ E + E - id
⇒ E + id - id
⇒ **id + id - id**

Since there are two leftmost derivations/two parse trees/two rightmost derivations for a string "id+id-id", the above CFG is ambiguous.

Removal of Ambiguity in arithmetic expression grammar:

We can remove ambiguity solely on the basis of the following two properties:

1. Precedence

If different operators are used, we will consider the precedence of the operators. The three important characteristics are:

1) **The level at which the production is present denotes the priority of the operator used.**
2) The **production at higher levels** will have **operators with less priority**.
   In the parse tree, the nodes which are at top levels or close to the root node will contain the lower priority operators.
3) The **production at lower levels** will have **operators with higher priority**.
   In the parse tree, the nodes which are at lower levels or close to the leaf nodes will contain the higher priority operators.

2. Associativity

If the same precedence operators are in production, then we will have to consider the associativity.

- **If the associativity is left to right, then we have to prompt a left recursion in the production**. The parse tree will also be left recursive and grow on the left side.
  +, -, *, / are left associative operators.
- **If the associativity is right to left, then we have to prompt the right recursion in the production.** The parse tree will also be right recursive and grow on the right side.
  ^ is a right associative operator.

Arithmetic expression grammar after removing ambiguity

E → E + T
E → E - T (rules are Left recursive because + and – have left association)
E → T
T → T * F
T → T / F (rules are Left recursive because * and / have left association)
T → F
F → D ^ F (rule is right recursive because the exponentiation operator is right associative)
F → D
D → (E) | num | id

Note:

- E - Expression, T - Term, F – Factor, num – number and id – identifier
- In math expressions, terms are the components added or subtracted, factors are the elements multiplied within each term, and coefficients are the numbers multiplying variables.

Example 2: Conditional Boolean expressions grammar

Ambiguous grammar:

```
B → B or B | B and B | not B | E relop E | true |false
```

From the above grammar String '**true *and* false *or* true**' can be derived in two ways:

First Leftmost derivation

B ⇒ B *or* B
⇒ B *and* B *or* B
⇒ true *and* B *or* B
⇒ true *and* false *or* true
⇒ **true *and* false *or* true**

Second Leftmost derivation

B ⇒ B *and* B
⇒ true *and* B
⇒ true *and* B *or* B
⇒ true *and* false *or* B
⇒ **true *and* false *or* true**

Since there are two leftmost derivations for a string 'true *and* false *or* true', **the above CFG is ambiguous.**

Unambiguous grammar: (Ambiguity is removed by considering operator precedence)

B → B *or* C | C
C → C *and* D | D
D → E *relop* E | F
F → *not* F | G
G → (B)| true | false

Note: **E** represents arithmetic expression, ***relop*** is relational operator, ***or*** is logical OR, ***and*** is logical AND.

Example 3: Regular Expression grammar

Ambiguous grammar:

```
R → R+R | RR | R* | R+ | R? | (R) | a | b | c
```

From the above grammar string ‘**a+bc**’ can be derived in two ways:

First Leftmost derivation

R ⇒ R + R
⇒ a + R
⇒ a + RR
⇒ a + bR
⇒ **a + bc**

Second Leftmost derivation

R ⇒ RR
⇒ R + RR
⇒ a + RR
⇒ a + bR
⇒ **a + bc**

Since there are two leftmost derivations for a string ‘a+bc’, the above CFG is ambiguous.

Unambiguous grammar: (Ambiguity is removed by considering operator precedence)

```
R → R+S | S      [Note: + in this rule represents alternation or Union]
S → ST | T
T → T* | T⁺ | T? | U        [Note: + in this rule represents Kleene plus]
U → (R) | a | b | c
```

Only one Leftmost derivation for the string ‘a+bc’:

R ⇒ R + S
⇒ S + S
⇒ T + S
⇒ U + S
⇒ a + S
⇒ a + ST
⇒ a + TT
⇒ a + UT
⇒ a + bT
⇒ a + bU
⇒ a + bc

Example 4: Dangling else

A common example of ambiguity in computer programming languages is the dangling else problem.

In many languages, **the else in an if–then(–else) statement is optional**, which results in nested conditionals having multiple ways of being recognized in terms of the context-free grammar.

Conditionals in two valid forms:

- **if-then form** and
- **if-then-else form**

In a grammar containing the rules

S → if C then S
S → if C then S else S
S → … [other statements rule]
C → … [boolean expressions rule]

some ambiguous phrase structures can appear. The expression

if a then if b then S1 else S2

can be parsed as either

if a then begin if b then S1 end else S2

or as

if a then begin if b then S1 else S2 end

depending on whether the else is associated with the first if or second if.

This is resolved in various ways in different languages.

- Sometimes the grammar is modified so that it is unambiguous, such as
  - by **requiring an endif statement** or
  - by **making else mandatory**

  Unambiguous grammar for if-then-else statements

  S → matched-stmt | open-stmt

  matched-stmt → if C then matched-stmt else matched-stmt | ... [other statements]

  open-stmt → if C then S | if C then matched-stmt else open-stmt

- In other cases, the grammar is left ambiguous, but the ambiguity is resolved by making the overall phrase grammar context-sensitive, such as by associating an else with the nearest if. In this case the phrase grammar is unambiguous, but the context-free grammar is ambiguous.

Exercise:

1) Find out whether the following grammars are ambiguous or not?

a) $S \to aSbS \mid bSaS \mid \varepsilon$ [generates set of strings where the number of a's equals the number of b's]

b) $S \to 1S \mid 11S \mid \varepsilon$ [generates 1*]

c) $S \to AB \mid aB$ [generates $(a)^+ b$]

$A \to a \mid aA$

$B \to b$

Solutions: The following strings have multiple LMDs or parse trees

a) baba or abab

b) 111

c) ab

Inherently ambiguous grammar

- Inherently ambiguous grammar is a context-free grammar (CFG) that cannot be made unambiguous, and the language it generates is also inherently ambiguous.
- Inherently ambiguous grammar is a CFG that can't be made unambiguous, and is used to describe a language that can't be generated by an unambiguous context-free grammar.

  Note: A CFL L is inherently ambiguous if every CFG for L is ambiguous.

Examples

- Consider the languages $L_1 = \{ a^n b^n c^m \mid n \geq 0 \text{ and } m \geq 1 \}$ and $L_2 = \{ a^n b^m c^m \mid m \geq 0 \text{ and } n \geq 1 \}$

  The grammar of language $L = L_1 \cup L_2$ is an example of an inherently ambiguous grammar.

$S \to A \mid B$

$A \to CD$

$C \to aCb \mid \varepsilon$ [generates $a^n b^n \mid n \geq 0$]

$D \to cD \mid c$ [generates $c^m \mid m \geq 1$]

$B \to EF$

$E \to aE \mid a$ [generates $a^n \mid n \geq 1$]

$F \to bFc \mid \varepsilon$ [generates $b^m c^m \mid m \geq 0$]

or

$S \to A \mid B$

$A \to Ac \mid c \mid C$

$C \to aCb \mid \varepsilon$

$B \to aB \mid a \mid D$

$D \to bDc \mid \varepsilon$

Strings that belong to both the languages are **$L_1 \cap L_2$ = { $a^nb^nc^n$ | n≥1}.**

Example string: abc

First Leftmost derivation	**Second Leftmost derivation**
S ⇒ A	S ⇒ B
⇒ CD	⇒ EF
⇒ aCbD	⇒ aF
⇒ abD	⇒ abFc
⇒ **abc**	⇒ **abc**

For every string **$a^nb^nc^n$** where **n**≥1**,** there exists two parse trees either using LMDs or RMDs. **Hence, the grammar is ambiguous.**

This CFG can't be made unambiguous.

Inherently ambiguous languages

- An inherently ambiguous language is a context-free language that cannot be recognized by any unambiguous grammar.
  In other words, some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.
- While some context-free languages (the set of strings that can be generated by a grammar) have both ambiguous and unambiguous grammars, **there exist context-free languages for which no unambiguous context-free grammar can exist. Such languages are called inherently ambiguous.**
- **A Context Free Language L is inherently ambiguous if every CFG for Context Free Language L is ambiguous.**

Proving ambiguity:

- Proving that a language is inherently ambiguous is difficult and undecidable in general (i.e., for complex context-free languages).
- However, there are some methods that can be used to prove the ambiguity of some simple languages, including:
  - Iterations on derivation trees
  - Iterations on semi-linear sets
  - Generating series

Note: There are no inherently ambiguous regular languages.

Examples:

1) Is the Language L = { $a^ib^jc^k$ | i,j,k ≥ 0 and either i = j or j = k} inherently ambiguous?

Solution: Is the grammar of L ambiguous? Why or why not?

Idea: This language is simply the union of **L_1 = { $a^ib^jc^k$ | i,j,k ≥ 0, i = j}** and **L_2 = { $a^ib^jc^k$ | i,j,k ≥ 0, j = k}**.
We can create simple grammars for the separate languages and union them:

S → A | B

For L_1, we simply ensure that the number of a's equals the number of b's:

A → Ac | C | ε
C → aCb | ε

Similarly, for ensuring that the number of b's equals the number of c's:

B → aB | D | ε

D → bDc | ε

This grammar is ambiguous. For string w = **$a^nb^nc^n$** , we may use either A or B production rule to generate w.

2) Find out whether the given grammar is inherently ambiguous or not.

S → aaS | aaaS | ε

Solution:

First, we will prove that the grammar is ambiguous by showing that a string 'aaaaa' has more than one LMDs

First Leftmost derivation	**Second Leftmost derivation**
S ⇒ aaS	S ⇒ aaaS
⇒ aaaaaS	⇒ aaaaaS
⇒ **aaaaa**	⇒ **aaaaa**

Two LMDs for the string "aaaaa" indicates that the grammar is ambiguous.

Try constructing another grammar which is unambiguous.

Regex = **(aa + aaa)*** **or** **(ε + aaa*)**

Language L = { ε, aa, aaa, aaaa, aaaaa, ...}

Grammar:

S → aaA | ε
A → aA | ε

or

S → aA | ε
A → aA | a

This grammar is unambiguous.

Hence, the given grammar is not inherently ambiguous.

Note:

- First prove that the grammar is ambiguous by deriving multiple LMDs or RMDs.
  Then try constructing another grammar which is unambiguous (if possible).

2) Find out whether the given grammar is inherently ambiguous or not.

S → SbS | a

Solution: **L={ a, aba, ababa, abababa, ...}**   Regex = **a(ba)*** or **(ab)*a**

First, we prove that the grammar is ambiguous by showing that a string has multiple LMDs.

Input string = **ababa**

First Leftmost derivation	**Second Leftmost derivation**
S ⇒ SbS	S ⇒ SbS
⇒ SbSbS	⇒ abS
⇒ abSbS	⇒ abSbS
⇒ ababS	⇒ ababS
⇒ ababa	⇒ ababa

Two LMDs for the string "ababa" indicates that the grammar is ambiguous.

Try constructing another grammar which is unambiguous.

S → Aa
A → Aab | ε

or **S → abS | a**

or **S → aB**
B → baB | ε

or **S → Sba | a**

These grammars are unambiguous.

Hence, the given grammar is not inherently ambiguous.

Exercise:

1) Find out whether the given grammar is inherently ambiguous or not.
   $S \rightarrow AB \mid aaaB$
   $A \rightarrow Aa \mid a$
   $B \rightarrow b$
2) Find out whether $L = \{ \{a^n b^n c^m d^m \mid n,m>0\} \cup \{ a^n b^m c^m d^n \mid n,m>=1 \}\}$ is inherently ambiguous or not.

Note:

The language L of our grammar G_1 (below) is not inherently ambiguous, even though the grammar G_1 is ambiguous, but there exist some grammars (i.e., G_2) that are not ambiguous for language L.

Ambiguous Grammar G_1: **$S \rightarrow AS \mid \varepsilon$**
$A \rightarrow A1 \mid 0A1 \mid 01$

Consider a string '**00111**':

LMD1: $S \Rightarrow AS \Rightarrow 0A1S \Rightarrow 0A11S \Rightarrow 00111S \Rightarrow 00111$

LMD2: $S \Rightarrow AS \Rightarrow A1S \Rightarrow 0A11S \Rightarrow 00111S \Rightarrow 00111$

Two LMDs for the string "00111" indicates that the grammar is ambiguous.

Change the ambiguous grammar to force the extra 1's to be generated last.

Unambiguous Grammar G_2: **$S \rightarrow AS \mid \varepsilon$**
$A \rightarrow 0A1 \mid B$
$B \rightarrow B1 \mid 01$

$S \Rightarrow AS \Rightarrow 0A1S \Rightarrow 0B1S \Rightarrow 0B11S \Rightarrow 00111$

Only one LMD for the string "00111". Hence the grammar G_2 is Unambiguous.

Why to Care ambiguity?

- Ambiguity of the grammar implies that at least some strings in its language have different structures (parse trees).
  - Thus, such a grammar is unlikely to be useful for a programming language, because two structures for the same string (program) implies two different meanings (executable equivalent programs) for this program.
  - Common example: the easiest grammars for arithmetic expressions are ambiguous and need to be replaced by more complex, unambiguous grammars.
- An inherently ambiguous language would be absolutely unsuitable as a programming language, because we would not have any way of fixing a unique structure for all its programs.

Note:

1. If a context free grammar G is ambiguous, language generated by grammar L(G) may or may not be ambiguous.
2. It is not always possible to convert ambiguous CFG to unambiguous CFG. Only some ambiguous CFG can be converted to unambiguous CFG.
3. There is no algorithm to convert ambiguous CFG to unambiguous CFG.
4. There always exists a unambiguous CFG corresponding to unambiguous CFL.
5. Deterministic CFL are always unambiguous and are parsed by LR parsers.

Questions:

1) Consider the following context free grammar

$S \rightarrow SS \mid ab \mid ba \mid \varepsilon$

I. G is ambiguous
II. G produces all strings with equal number of a’s and b’s
III. G can be accepted by a deterministic PDA

Which combination below expresses all the true statements about G?

a) I only
b) I and III only
c) II and III only
d) I, II and III

Solution:

I. There are different LMD’s for string abab which can be

LMD1: $S \Rightarrow \underline{S}S \Rightarrow \underline{S}SS \Rightarrow ab\underline{S}S \Rightarrow abab\underline{S} \Rightarrow abab$

LMD2: $S \Rightarrow \underline{S}S \Rightarrow ab\underline{S} \Rightarrow abab$

Even ϵ also has infinite number of derivations.
So the grammar is ambiguous. So, statement I is true.

II. "G produces all strings with equal number of a's and b's", is false since L(G)=(ab+ba)* and this language cannot produce all strings with equal number of a's and b's. For example aabb has equal number of a's and b's but aabb $\notin$ L(G).

III."G can be accepted by a DPDA" is true, since L(G) =(ab+ba)* and this is a regular language, since we have written a regular expression for it. Since every regular language is also a DCFL, a DPDA exists. which accepts this language.

2) Which one of the following statements is FALSE?

A. There exist context-free languages such that all the context-free grammars generating them are ambiguous.
B. An unambiguous context free grammar always has a unique parse tree for each string of the language generated by it.
C. Both deterministic and non-deterministic pushdown automata always accept the same set of languages.
D. A finite set of string from one alphabet is always a regular language.

Solution:

(A) is correct because for ambiguous CFL’s, all CFG corresponding to it are ambiguous.
(B) is also correct as unambiguous CFG has a unique parse tree for each string of the language generated by it.
(C) is false as some languages are accepted by Non – deterministic PDA but not by deterministic PDA.
(D) is also true as finite set of string is always regular.

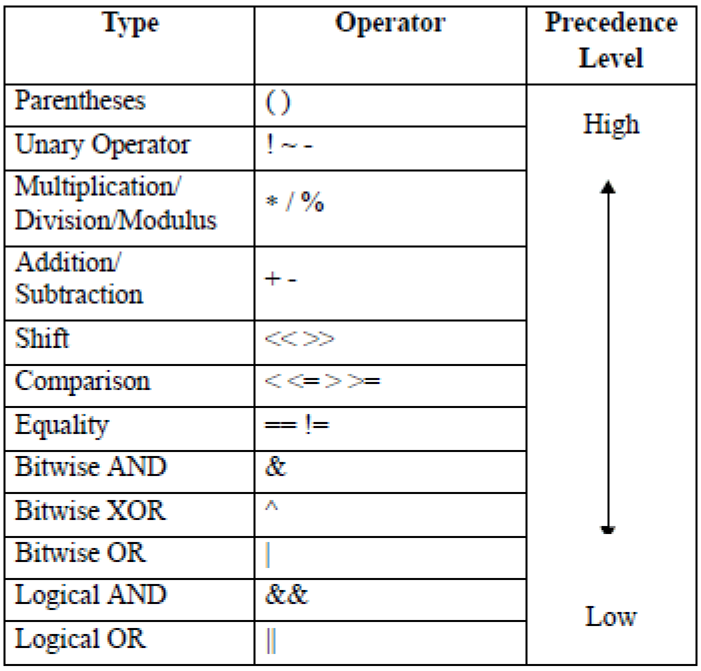
So option (C) is correct option.

Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often, we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Note: CFG Relationship with other Computation Models

- A context-free grammar can be generated by pushdown automata just as regular languages can be generated by finite state machines. Since **all regular languages can be generated by CFGs, all regular languages can too be generated by pushdown automata.**
- **Any language that can be generated using regular expressions can be generated by a context-free grammar.**
- The way to do this is to take the regular language, determine its finite state machine and write production rules that follow the transition functions.

Note: Operators: Order of Precedence



Type	Operator	Precedence Level
Parentheses	( )	High
Unary Operator	! ~ -	
Multiplication/ Division/Modulus	* / %	
Addition/ Subtraction	+ -	
Shift	<< >>	
Comparison	< <= > >=	
Equality	== !=	
Bitwise AND	&	
Bitwise XOR	^	
Bitwise OR	|	
Logical AND	&&	
Logical OR	||	Low