# AUTOMATA FORMAL LANGUAGES AND LOGIC


## Lecture notes on

## Parsing & Ambiguity

**Prepared by:**

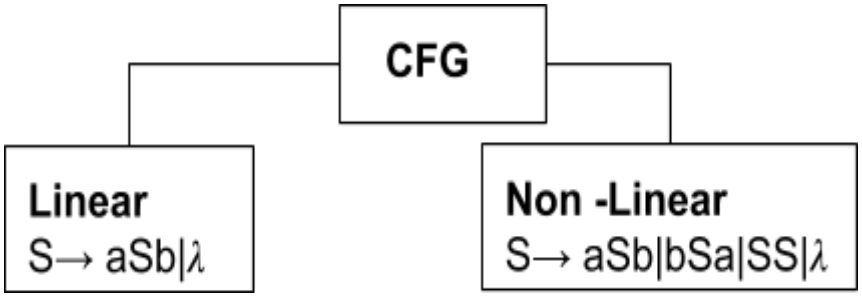
**Prof.Sangeeta V I**
**Assistant Professor**

**Department of Computer Science & Engineering**
**PES UNIVERSITY**
**(Established under Karnataka Act No.16 of 2013)**
**100-ft Ring Road, BSK III Stage, Bangalore – 560 085**

## Table of Contents

**Examples Solved :**

## 1. Derivation




A Context Free Grammar can be linear (only one non-terminal on the RHS) or nonlinear (more than one non-terminal on the RHS).

For the CFG (linear grammar) S→ aSb|$\lambda$ ,we will derive the string w=aabb

S⇒aSb (using the production rule and replacing S with S→ aSb)

S⇒aaSbb (using the production rule and replacing S with S→ aSb)

S⇒aa$\lambda$bb (using the production rule and replacing S with S→ $\lambda$)

S⇒aabb

### 1.1 Sentence and Sentential Form

A **sentential form** is any string derivable from the start symbol.
**Sentential form ∈ (V ∪ T)***
In the derivation of the string aabb **:aSb,aaSbb,aa$\lambda$bb,aabb** are sentential forms.

A **sentence** is a sentential form consisting only of terminals,i.e **Sentence∈T***. So here "**aabb"** is the sentence.

## 1.2 Leftmost Derivation(LMD) and Rightmost Derivation(RMD)

For the CFG (non-linear grammar) S→ aSb|bSa|SS|$\lambda$ ,we will derive the string w=aabb.

1. Which productions to use at each step?

   S→ SS

2. Which non terminal to be expanded first?

   S→ **S**S

If we choose to expand the left most variable on the RHS always it is called left most derivation.

A **leftmost derivation(LMD)** is the one in which we replace/expand the leftmost variable in a production body(RHS) by one of its production bodies first, and then work our way from left to right.

S→ S**S**

If we choose to expand the left most variable on the RHS always it is called left most derivations.

A **rightmost derivation(RMD)** is one in which we replace/expand the rightmost variable by one of its production bodies first, and then work our way from right to left.ne of its production bodies first, and then work our way from left to right.

For the linear grammar we do not have leftmost and rightmost derivations as we have only one non terminal on the RHS and hence there is only one derivation. So, the LMD and RMD occurs only in non-linear grammar.

For the CFG ,S→ aSb|bSa|SS|$\lambda$ ,we will derive the string w=aabb using leftmost derivation (LMD).

For deriving the string "aabb",to start with which production do we use?

S $\Rightarrow^{lm}$ SS (using S→ aSb)
S $\Rightarrow^{lm}$ aSbS (using S→ $\lambda$)
S $\Rightarrow^{lm}$ abS (using S→ bSa)
S $\Rightarrow^{lm}$ abbSa (using S→ $\lambda$)
Sabba

For the CFG ,S→ aSb|bSa|SS|$\lambda$ ,we will derive the string w=aabb using leftmost derivation (LMD).

S $\Rightarrow^{rm}$ SS (using S→ bSa)

S $\Rightarrow^{rm}$SbSa(using S→ $\lambda$)

S $\Rightarrow^{rm}$Sba (using S→asb)

S $\Rightarrow^{rm}$aSbba (using S→ $\lambda$)

S $\Rightarrow^{rm}$abba

## 2. Ambiguous Grammar

A grammar G=(V,T,P,S) is ambiguous if and only if there exists at least one string w,w$\in$L(G) such that there exists two or more parse trees.

**Example :**

Consider the language which generates an arithmetic expression.

G=(V,T,P,S)

V={E} T={+,*,id} S is the start symbol.

P ={

E→ E+E

E→ E-E

E→ E*E

E→ E/E

E→ (E)

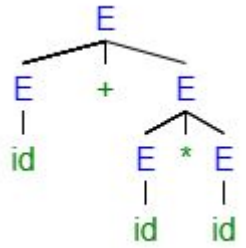
E→ id

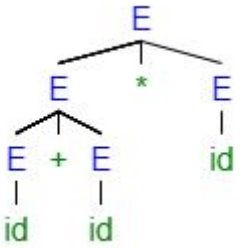
E→ num }

Let's derive the string **id+id*id**

| **LMD 1:** | **LMD 2:** |
|---|---|
| E⇒E+E | E⇒E*E |
| E⇒id+E | E⇒E+E*E |
| E⇒id+E*E | E⇒id+E*E |
| E⇒id+id*E | E⇒id+id*E |
| E⇒id+id*id | E⇒id+id*id |

**Parse tree for LMD1:** **Parse tree for LMD2:**




- We got two parse trees using LMD with yield as id+id*id.
- The **yield** of the parse tree is the string that results when we concatenate the leaves from left to right .
- The two different parse trees for the same string implies different semantics for the same string.This behaviour is highly undesirable from the point of view of the compiler.
- Ambiguity can be a problem in things like programming languages where we want agreement between the programmer and compiler over what happens.
- Hence,whenever we consider the design of any programming language ,we ensure that all the valid programs are unambiguous ,resulting in exactly one parse tree.

The grammar for generating arithmetic expressions was ambiguous as we got two parse trees by applying LMD twice for the string id+id*id.

If we want to evaluate string 3+4*5 ,

Parse tree 1:
Gives the right answer =3+20=23.
Here ,4*5 is evaluated first and then on the result we apply + with 3.
Parse tree 2:
Gives the wrong answer =7*5=35.
Here ,3+4 is evaluated first and then on the result we apply * with 5.

Semantics changed due to the grammar being ambiguous.
Hence,we must remove the ambiguity in the grammar.

Now there are two questions.

1. **How to find out whether the given grammar is ambiguous?**
   There is no algorithm for detecting whether an arbitrary grammar is ambiguous. .We must go by trial and error. Look for a string 'w' whether we get more than one parse tree. Some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.
2. **How to convert ambiguous grammar to unambiguous grammar?**
   There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.

Let's work on the first one.

**Note:**For a given L we can have multiple grammars deriving that language.

**Example 1:**

Find out whether the given grammar is ambiguous or not?

1. S→ aS|Sa|$\lambda$
2. S→ aSbS|bSaS|$\lambda$
3. R→ R+R|RR|R*|a|b|c
4. S→ 1S|11S|$\lambda$
5. S→ AB|aaB
   A→ a|Aa
   B→ b

**Answers**,strings which prove the grammar is ambiguous,for which we can construct more than one parse trees :

1. aa
2. abab
3. a+bc
4. 111
5. aab

## 3. Grammar ambiguity and Language Ambiguity

Grammar ambiguity and language ambiguity are different.
We can have multiple grammar generating a given language.

**For example:**
$L=\{a^n|n>=0\}$
Different grammars generating L:
G1={aS|$\lambda$}
G2={Sa|$\lambda$}
G1 and G2 are unambiguous grammars.

G3={aS|a|$\lambda$}
G4={Sa|a|$\lambda$}
G5={Sa|aS|$\lambda$}
For the string "a" we get multiple parse trees in G3,G4 and G5.Hence,G3,G4 and G5 are ambiguous grammars.

But the **language $L=\{a^n|n>=0\}$ is unambiguous.**

### 3.1. Inherently ambiguous language:

- Language is ambiguous or inherently ambiguous language when every grammar that derives that language is ambiguous.
- Language becomes unambiguous if there exists at least one grammar(deriving L)which is unambiguous.

**A formal language is said to be ambiguous if every grammar for the language is ambiguous.**

**Example 2:**

Show that union of $\{a^nb^nc^m|n>=0,m>=1\}$ $\{a^nb^mc^m|n>=1,m>=0\}$ is inherently ambiguous.

$L=\{a^nb^nc^m\} \cup \{a^nb^mc^m\}$

$S1 \rightarrow Ac$ $S2 \rightarrow aB$

$A \rightarrow aAb|\lambda$ $B \rightarrow bBc|\lambda$

$S \rightarrow S1|S2$

- Strings that belong to both the languages are $L1 \cap L2=\{a^nb^nc^n\}$
- For every string in $a^nb^nc^n$ there exists two parse trees(either using LMD or RMD).

Example:w="abc"


- We get two parse trees for the string "abc".Hence ,the grammar is ambiguous.
- To generate this language we have only this grammar .No other grammar can generate L.
- Hence ,the language L is ambiguous or we can say that the given language L is inherently ambiguous.

**Example 3:**

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

S→ SaS|b

Note:

- ➔ First prove the grammar is ambiguous.
- ➔ Then try constructing another grammar which is unambiguous((if possible) .

S→ SaS|b

L={bab,babab,bababab,.....}

- ➔ First we prove the grammar is ambiguous by showing that a string has more than one parse tree. .
- Consider the string ”babab”,to prove the grammar is ambiguous.


Two parse trees for the string “babab” indicates the grammar is ambiguous.

- ➔ Try constructing another grammar which is unambiguous.
- The language is regular ,we can write the regular expression b(ab)+

S→ bA

A→ abA|ab

Now,the grammar is unambiguous.

Parse tree for string “bab” and “babab”

**Example 4:**

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

S→ aaS|aaaS|λ

➔ First we prove the grammar is ambiguous by showing that a string has more than one parse tree. .

- Parse tree for the string "aaaaa"


Two parse trees for the string "aaaaa" indicates the grammar is ambiguous.

➔ Try constructing another grammar which is unambiguous.

L=(aa+aaa)*

S→ aaA|λ

A→ aA|λ

This grammar is unambiguous.

**Example 5:**

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

S→ AB|aaB

A→ a|Aa

B→ b

➔ First we prove the grammar is ambiguous by showing that a string has more than one parse tree.

- Parse tree for the string "aab"


Two parse trees for the string "aab" indicates the grammar is ambiguous.

➔ Try constructing another grammar which is unambiguous

L={ab,aab,aaaaaaaab,.....}

L=aa*b

S→ aAb

A→ aA|λ

This grammar is unambiguous.

## 4. Removing ambiguity.

Some of the grammars can be converted to unambiguous grammar using disambiguity rules.

### 4.1 Reasons for ambiguity:

1. Associativity rule not taken care.
2. Precedence rule not taken care.

If we take care of these two rules ,our grammar will become unambiguous,

1. Enforcing associativity.
   Consider the grammar: E→ E+E | E*E | id,
   String generated by grammar : id+id+id
   When there are two operators on either side of the operand,which operator to associate the operand with ?If we associate with the left side operator it is called left associative otherwise right associative.
   If the operator has to be left associative the grammar should be left recursive.
   Left most symbol in RHS is the same as the symbol in LHS.
   Allow the grammar to grow towards right or left,depending on whether the grammar is left associative(left recursive) or right associative(right recursive).

2. Enforcing precedence.
   Highest precedence operator must come at a lower level,so that it is evaluated first.

Ambiguous grammar:
E→ E+E | E*E | id
Unambiguous grammar :
E→ E+T/T (E→ T ,If there is no +, we can directly to go T)
T→ T*F/F
F→ id

**Example 6:**
Eliminate ambiguity in following grammar:
B→ B or B | B and B| not B | True |False

“or” ,”and” left associative, not is non associative(unary operator).
Lowest precedence operator is close to start symbol and highest precedence operator is far from the start symbol .

Unambiguous grammar :
B→ B or F | F
F→ F and G | G
G→ not G|True | False

**Example 7:**
Eliminate ambiguity in following grammar:
R→ R+R|RR|R*|a|b|c

Highest to lowest precedence order:
Kleen closure(*)
Concatenation(.)
Union (+)

Unambiguous grammar :
R→ R+S|S
S→ S.T|U
U→ U*|a|b|c
The variable name can be anything of your choice.