PRINCIPLES OF COMPILER DESIGN - SYNTAX ANALYSIS(PARSING)

Individual Assignment 02

QUESTION- 39
Name: Abrham Abebaw
Course Code: SEng4031
Id: BDU1504862
Section: B

Table of Contents

QUESTION 39: What is ambiguity in a grammar?

Ambiguous Grammar

1. Introduction

Context-Free Grammars (CFGs) are a fundamental concept in **syntax analysis** and **compiler design**. They provide a formal method to describe the **structure of a language**, whether it is a natural language or a programming language.

CFGs define **how symbols can be combined** to form valid sequences, called **strings**. These production rules are essential for **parsing**, the process through which a compiler verifies whether an input string adheres to the grammatical rules of the language.

2. Classification of CFGs

CFGs can be categorized based on the **derivation trees** (also called **parse trees**) they generate. The two main categories are:

2.1 Ambiguous Grammars

- Ambiguous grammar is one in which a **single string can have more than one derivation tree**.
- Each derivation tree represents a different hierarchical interpretation of the string.
- Ambiguity can lead to **multiple interpretations**, which may cause **unclear or inconsistent program behavior**.
- Ambiguous grammar makes parsing **more complex** and **less predictable** in compiler design.

2.2 Unambiguous Grammars

- An unambiguous grammar ensures that **every valid string has exactly one derivation tree**.
- Guarantees a **single, consistent interpretation** for each string.
- Unambiguous grammar makes **parsing simpler, more predictable, and reliable**.

3. Formal Definition

A context free grammar is formally represented as:

$$G = (V, T, P, S)$$

Where:

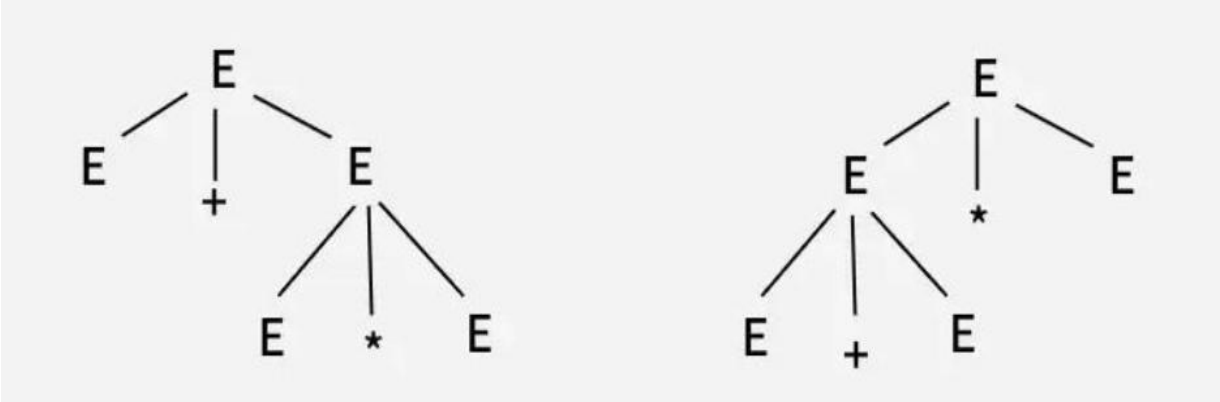
- **V** = Set of variables or non-terminal symbols
- **T** = Set of terminal symbols
- **P** = Set of production rules
- **S** = Start symbol

Ambiguity occurs if there exists at least one string in the language that can be generated in **more than one way**, producing **multiple parse trees**.

4. Understanding Ambiguity in CFGs

- Ambiguity arises when the grammar allows a string to be generated through **different sequences of rule applications**.
- It can be observed through **Leftmost Derivations (LMDs)** or **Rightmost Derivations (RMDs)**:
    - **LMD**: Expands the leftmost non-terminal first.
    - **RMD**: Expands the rightmost non-terminal first.
- If a string has multiple derivations that result in **different parse trees**, the grammar is ambiguous.
- Each parse tree shows a **different hierarchical structure**, even though the terminal string is the same.

Example 1. Let us consider this grammar: E-> E +E | E*E| id, We can create 2 parse tree from this grammar to obtain a string **id + id * id.**



Both the above parse trees are derived from the same grammar rules, but both parse trees are different. Hence the grammar is ambiguous.

Example 2. Let us now consider the following grammar:

```
Set of alphabets ? = {0,…,9, +, *, (, )}

E -> I
E -> E + E
E -> E * E
E -> (E)
I -> ? | 0 | 1 | … | 9
```

From the above grammar String **3*2+5** can be derived in 2 ways:

```
I) First leftmost derivation
leftmost derivation
        E=>E*E
E=>E+E
         =>I*E
=>E*E+E
         =>3*E+E
=>I*E+E
         =>3*I+E
=>3*E+E
         =>3*2+E
=>3*I+E
         =>3*2+I
=>3*2+I
         =>3*2+5
=>3*2+5
```

Following are some examples of ambiguous grammar:

- S-> aS |Sa| ?
- E-> E +E | E*E| id
- A -> AA | (A) | a
- S -> SS|AB, A -> Aa|a , B -> Bb|b

Whereas following grammars are unambiguous:

- S -> (L) | a, L -> LS | S
- S -> AA, A -> aA , A -> b

5. Key Points

- Ambiguity occurs when a grammar does **not clearly specify how parts of a string should be grouped or structured**.
- Ambiguous grammars can create **confusion in interpretation**, which is problematic in programming languages.
- Detecting and resolving ambiguity is **essential in compiler construction** to ensure that programs are interpreted correctly.
- Unambiguous grammars are **highly preferred** because they simplify parsing and make compilers more **predictable and reliable**.

6. Removal of Ambiguity in Grammar

To remove ambiguity from a grammar, follow these simple steps:

1. **Simplify production rules**: Break down complex rules into smaller and simpler ones. This way, the grammar won't allow multiple interpretations for the same string.
2. **Set precedence and associativity**: For things like math operations, make sure the order in which they are applied is clear. For example, define that multiplication happens before addition, or how to group operations (left-to-right or right-to-left).
3. **Fix left recursion**: Left recursion occurs when a rule refers to itself in a way that causes infinite loops. To fix it, change the rules so that recursion happens at the end, not at the start.
4. **Factor out common parts**: If two rules start the same way, combine the common part. For example, instead of having rules like $\mathbf{A \rightarrow \alpha\beta}$ and $\mathbf{A \rightarrow \alpha\gamma}$, make it $\mathbf{A \rightarrow \alpha A'}$ and $\mathbf{A' \rightarrow \beta \mid \gamma}$.

7. Inherent Ambiguity

A Context-Free Language (CFL) is said to be **inherently ambiguous** if every possible grammar for the language is ambiguous. This means no matter how you write the grammar for the language, there will always be strings in that language that can be parsed in more than one way.

Example:

Consider the language **L** defined as:

$L = \{a^n b^n c^m d^m : n \geq 1, m \geq 1 \} \cup \{ a^n b^m c^m d^n : n \geq 1, m \geq 1 \}$

A grammar for this language could be:

- **S → AB | C**
- **A → aAb | ab**
- **B → cBd | cd**
- **C → aCd | aDd**
- **D → bDc | bc**

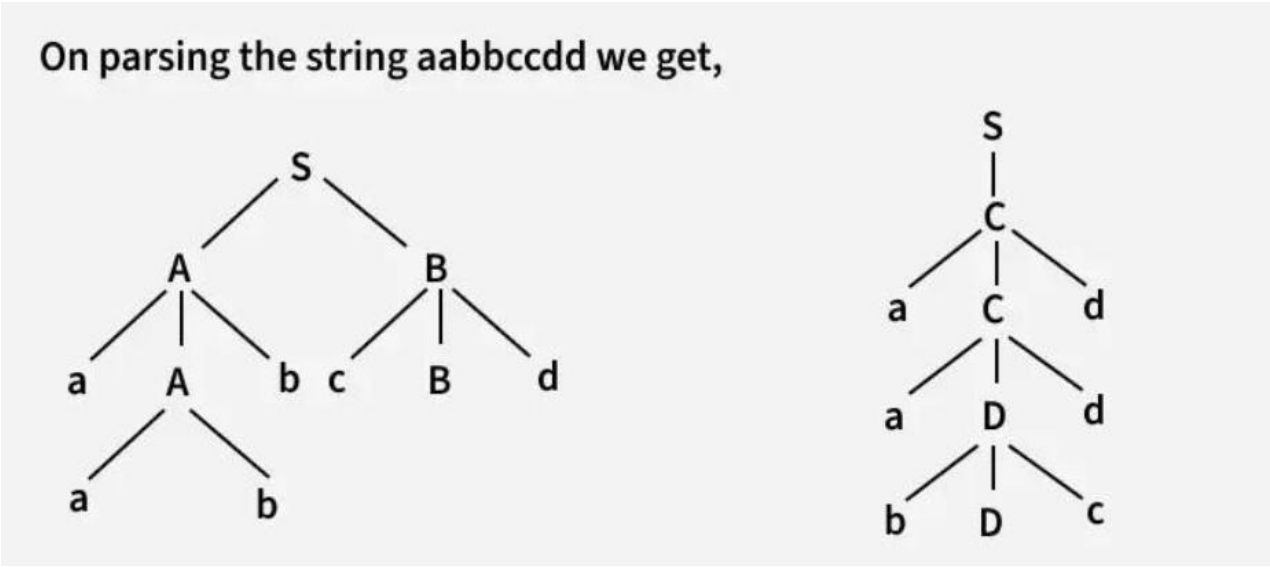
Let's see how the string **aabbccdd** can be parsed. This string has two possible leftmost derivations:

Using **S → AB**:

- **S ⇒ AB**
- **A → aAb ⇒ aAbB**
- **A → ab ⇒ aabbB**
- **B → cBd ⇒ aabbcBd**
- **B → cd ⇒ aabbccdd**

Using **S → C**:

- **S ⇒ C**
- **C → aCd ⇒ aCd**
- **C → aDd ⇒ aaDdd**
- **D → bDc ⇒ aabDcdd**

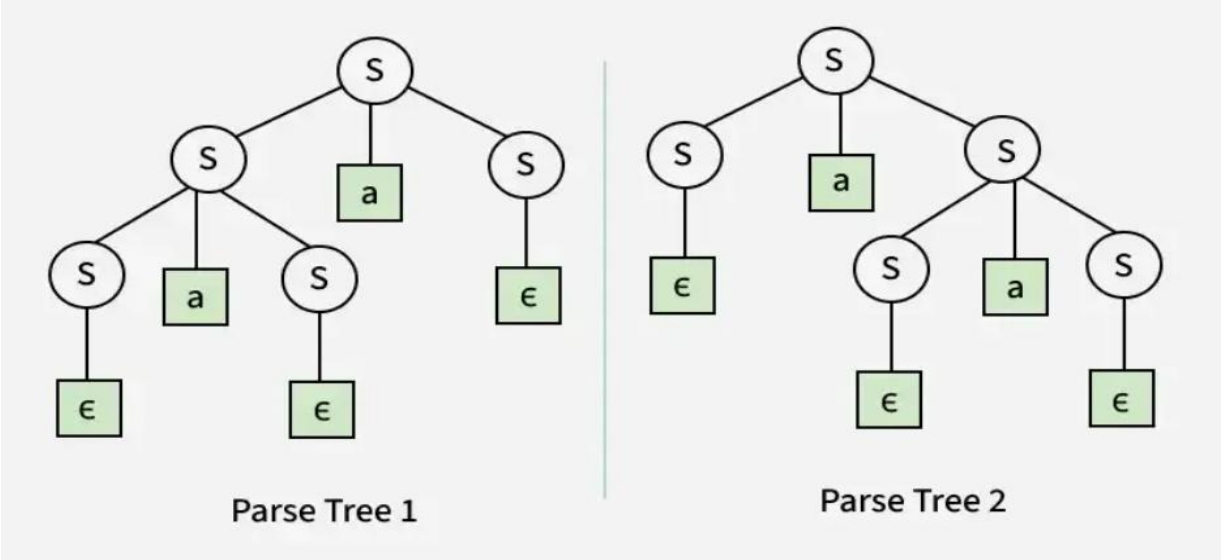


Both derivations result in the same string **aabbccdd**, but they come from different rules and paths, showing that there is more than one way to parse the string.
Since there are multiple ways to parse the string **aabbccdd** and this pattern holds for all grammars of **L**, we can conclude that **L** is inherently ambiguous. This means no grammar for **L** can avoid ambiguity for all strings in the language.

Important Points on Ambiguous Grammar

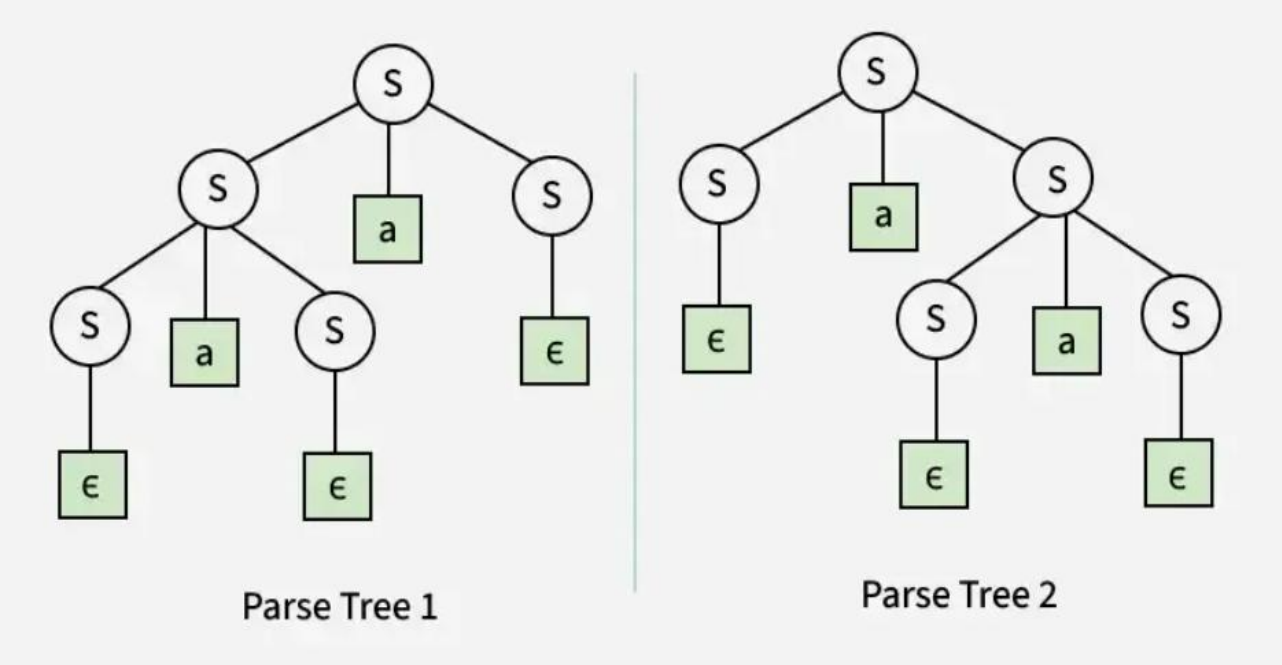
1) A grammar is **ambiguous** if it contains both **left recursion** and **right recursion**. This combination can lead to multiple ways of deriving the same string, creating more than one parse tree.

Example: **S → SaS | ε**
Here, the grammar has both **left recursion** (S → SaS) and **right recursion** (S → ε), making it ambiguous. This allows multiple derivations for the same string, such as **{aa}**, leading to more than one parse tree.



2) Even if a grammar does not have both **left** and **right recursion**, it can still be **ambiguous**. The absence of recursion does not guarantee that the grammar is unambiguous.

Example: S → aB | ab , A → AB | a , B → Abb | b
In this example, there is no left or right recursion, but the grammar is still ambiguous. For the string **{ab}**, we can derive it in multiple ways, resulting in more than one parse tree. This shows that even without recursion, a grammar can still be ambiguous.



From the above example, we can see that even if both left and right recursion are not present in grammar, the grammar can be ambiguous.