Context-Free Grammar Introduction

Definition – A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T = NULL.**
- **P** is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rules **P** does have any right context or left context.
- **S** is the start symbol.

Example

- The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
- The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
- The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

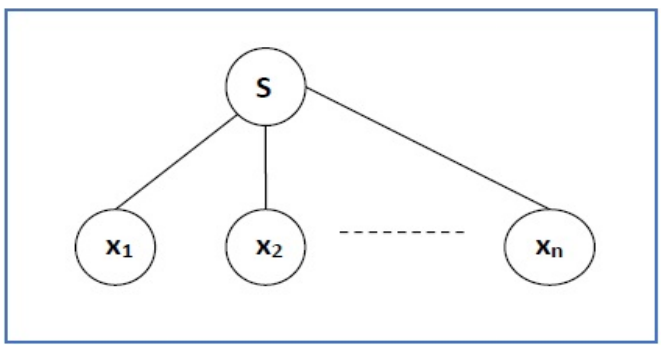
Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

- **Root vertex** – Must be labeled by the start symbol.
- **Vertex** – Labeled by a non-terminal symbol.
- **Leaves** – Labeled by a terminal symbol or ε.

If $S \rightarrow x_1 x_2 \ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows –



There are two different approaches to draw a derivation tree −

Top-down Approach −

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

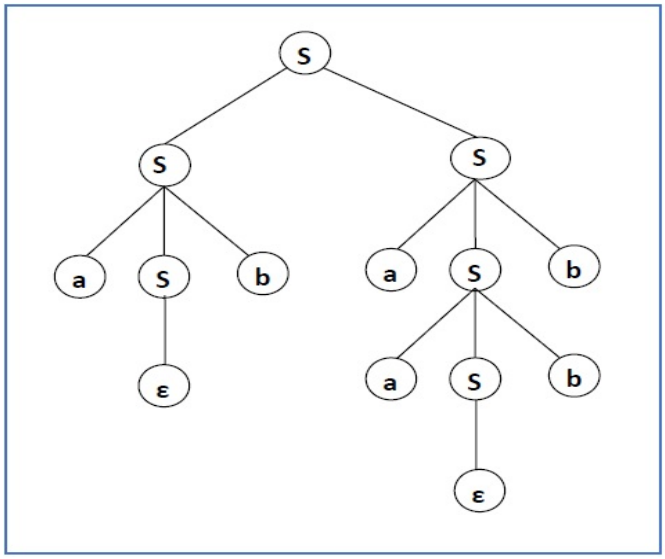
Example

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is abaabb

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



Sentential Form and Partial Derivation Tree

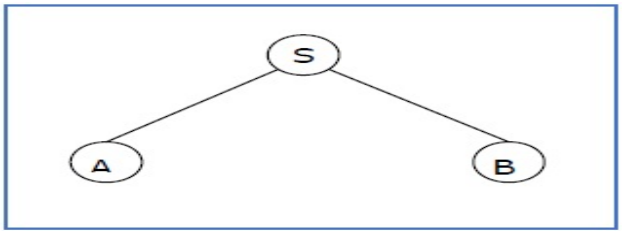
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are –

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following –



If a partial derivation tree contains the root S, it is called a **sentential form**. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String

- **Leftmost derivation** – A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** – A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

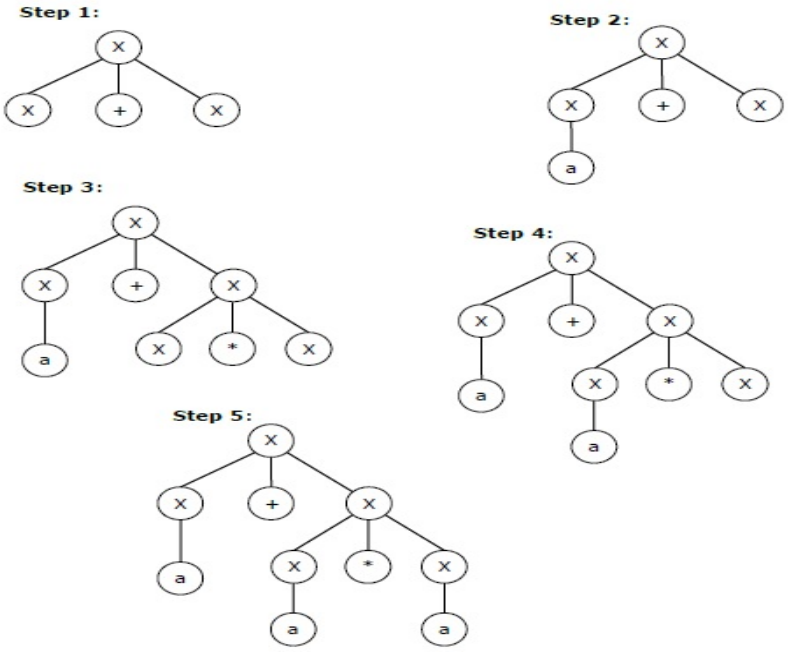
X → X+X | X*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** may be –

X → X+X → a+X → a + X*X → a+a*X → a+a*a

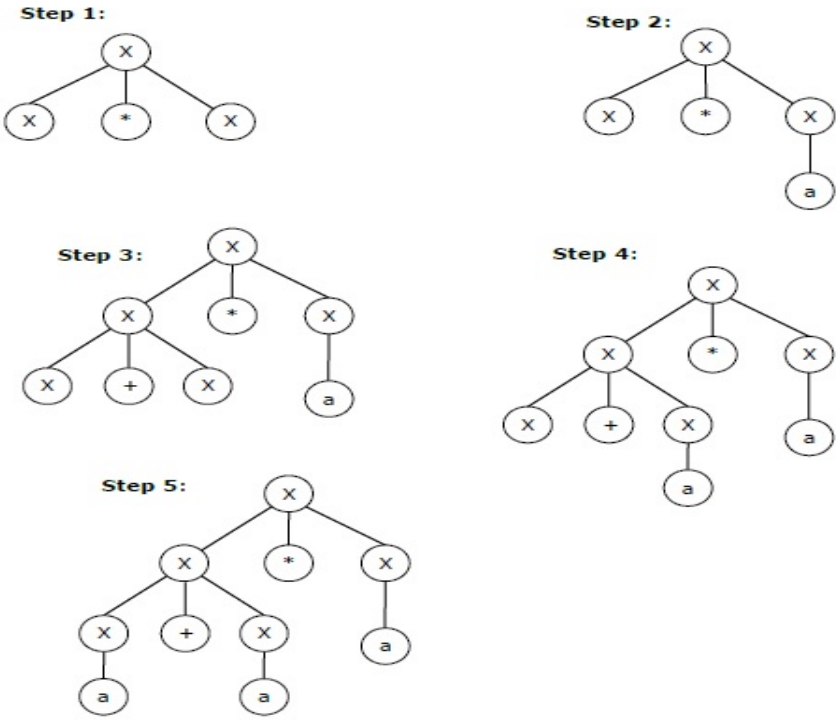
The stepwise derivation of the above string is shown as below –



The rightmost derivation for the above string **"a+a*a"** may be –

X → X*X → X*a → X+X*a → X+a*a → a+a*a

The stepwise derivation of the above string is shown as below –



Left and Right Recursive Grammars

In a context-free grammar **G**, if there is a production in the form **X → Xa** where **X** is a non-terminal and **a** is a string of terminals, it is called a **left recursive production**. The grammar having a left recursive production is called a **left recursive grammar**.

And if in a context-free grammar **G**, if there is a production is in the form **X → aX** where **X** is a non-terminal and **a** is a string of terminals, it is called a **right recursive production**. The grammar having a right recursive production is called a **right recursive grammar**.

Derivation Tree in Automata Theory

In automata theory, we use derivation trees or parse trees to derive language from grammars. In this chapter, we will see the concept of derivation trees in the context of context-free-grammars. Derivation tree is a fundamental tool in understanding and representing the structure of strings generated by context-free grammars.

What are Derivation Tree?

A Derivation Tree, also known as a **Parse Tree**, is a visual representation of the process by which a context-free grammar generates a particular string. It provides a hierarchical breakdown of the string. This illustrates the sequence of production rules applied to derive the string from the grammar's start symbol.

Key Elements of a Derivation Tree

Before learning how to construct a derivation tree, let's understand their essential components:

- **Root Vertex** – The root vertex represents the starting point of the derivation process and is always labeled with the grammar's start symbol.
- **Vertices (Internal Nodes)** – These nodes represent the non-terminal symbols of the grammar, serving as intermediate stages in the derivation.
- **Leaves** – These nodes represent the terminal symbols of the grammar, forming the final string derived from the grammar. They can also be labeled with the empty symbol, ε, if the grammar allows for empty productions.

Example of Context-Free Grammar

Let's consider the following context-free grammar G –

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

Where,

- V = {S, A, B} (Variables or Non-terminal symbols)
- T = {0, 1} (Terminal symbols)
- P = {S → 0B, A → 1AA | ε, B → 0AA} (Production Rules)
- S = S (Start Symbol)

This grammar defines a language where strings begin with a '0', then followed by a combination of '0's and '1's.

Derivation Tree Construction

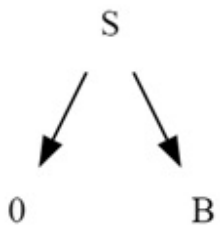
To illustrate the formation of a derivation tree, let us examine the derivation of the string "001" from the grammar G:

Root Vertex

We begin by placing the start symbol 'S' as the root vertex of the tree.

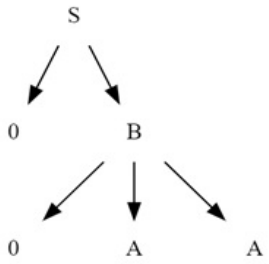
Applying Production Rules

Looking at the production rules, we see that 'S' can be replaced by "0B".



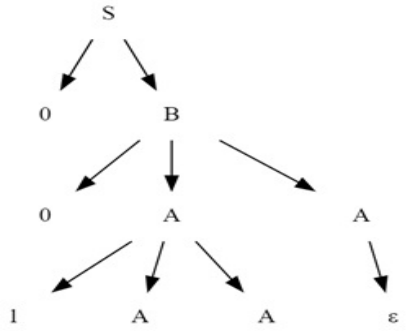
Continuing the Derivation

The rightmost vertex is 'B', and the production rule for 'B' is "0AA". Applying this rule, we obtain −

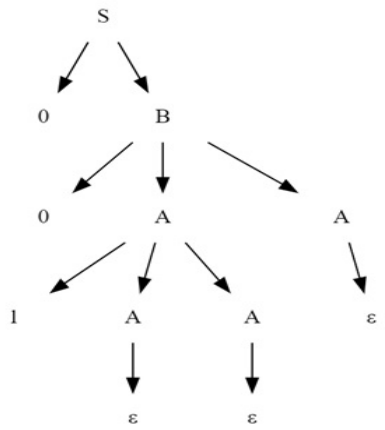


Reaching Terminal Symbols

Now, we have two 'A' variables. Since the production rule for 'A' allows for the empty string 'ε', we can replace the right most 'A's with 'ε', and left A to 1AA.



Now from the final non-terminals A to ε to get final tree.



This final structure represents the derivation tree for the string "001" from grammar G.

Left Derivation Tree and Right Derivation Tree

There are two primary methods for constructing derivation trees: The left derivation and right derivation. These methods dictate the order in which production rules are applied to non-terminal symbols within the sentential form.

- **Left Derivation Tree** – A Left Derivation Tree is generated by consistently applying production rules to the leftmost variable in each step of the derivation. This method uses expanding the non-terminal symbols on the left side of the sentential form.
- **Right Derivation Tree** – A Right Derivation Tree is obtained by applying production rules to the rightmost variable in each step. This method uses expanding non-terminal symbols on the right side of the sentential form.

Example of Left and Right Derivation Trees

Let's consider a new grammar and generate both left and right derivation trees to understand the difference in their construction.

Grammar –

$$S \rightarrow aAS|aSS|\varepsilon S \rightarrow aAS|aSS|\varepsilon$$

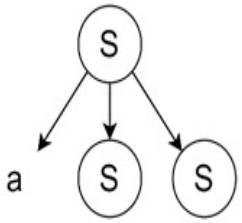
$$A \rightarrow SbA|baA \rightarrow SbA|ba$$

String to derive: "aabbaa"String to derive: "aabbaa"

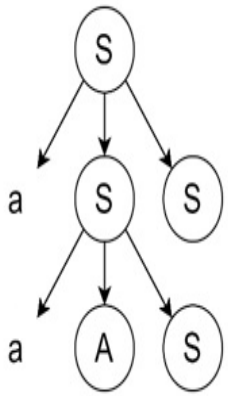
Left Derivation Tree

For the left derivation tree, we start from S.

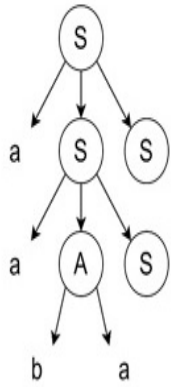
- **Applying Leftmost Derivation** – We apply the production rule **'S → aSS'** to the leftmost 'S'.



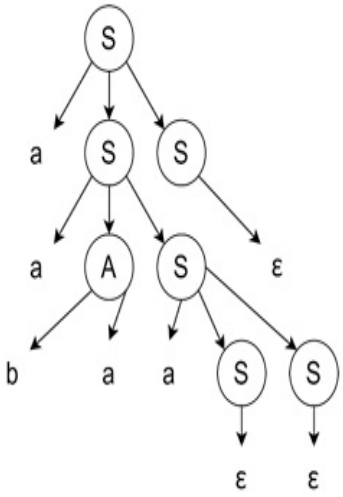
- **Expanding Leftmost Variable** – We expand the leftmost 'S' using the rule 'S → aAS'.



- **Continuing Leftmost Derivation** – We continue expanding the leftmost 'A' using the rule 'A → ba'.



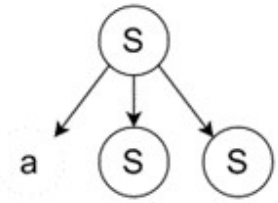
- **Completing the Derivation** – We expand the next leftmost S to aSS, and from there remaining S to 'S → ε'.



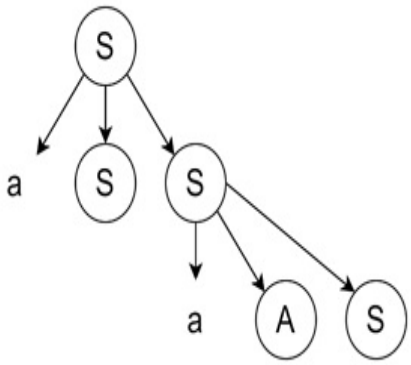
Right Derivation Tree

Like left most derivation, we start from the start symbol 'S'.

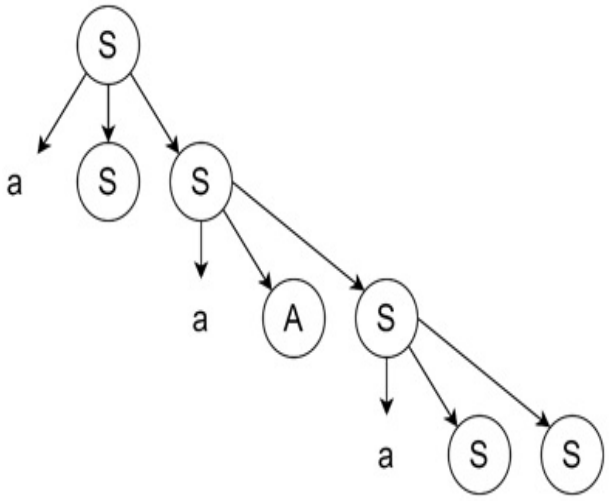
- **Applying Rightmost Derivation** – We apply the production rule 'S → aSS' to the rightmost 'S'.



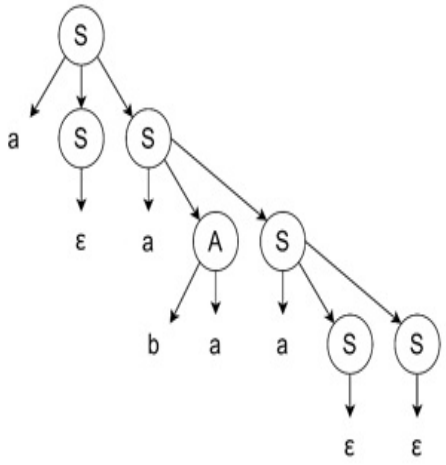
- **Expanding Rightmost Variable** – We expand the rightmost 'S' using the rule 'S → aAS'.



- **Continuing Rightmost Derivation** – We expand the rightmost 'S' using the rule 'S → aSS'.



- **Completing the Derivation** – We expand the remaining 'A' and 'S' from the right to left, the last two S will produce ε. Then A will bring 'ba' and the leftmost S will bring ε again.



Conclusion

Derivation trees provide a method of visualizing the structure of strings generated by context-free grammars. These are essential in several cases including understanding the concepts CFG derivation, parsing in compiler design, etc. In this chapter, we explained the concepts in detail with step by step examples for a clear understanding.

Parse Tree in Automata Theory

Parsing is an important part in context of automata and compiler design perspective. Parsing refers to the process of analyzing a string of symbols, either in natural language or programming language, according to the rules of a formal grammar. This is done by the concept called parser, which determines if a given string belongs to a language defined by a specific grammar.

The parser often generates a graphical representation of the derivation process. In this chapter, we will explain the concept of parsing and parse trees through examples.

Basics of Context-Free Grammar (CFG)

Before getting the idea of parse trees, it is essential to understand Context-Free Grammar (CFG). A CFG consists of a set of production rules that describe all possible strings in a given formal language. These production rules define how the symbols in the language can be combined and transformed to generate valid strings.

A Context-Free Grammar G is defined by four components –

- V: A finite set of variables (non-terminal symbols).
- T: A finite set of terminal symbols.
- S: A start symbol, which is a special non-terminal symbol from V.
- P: A finite set of production rules, each having a form $A \rightarrow \alpha$ $A \rightarrow \alpha$, where A is a non-terminal and α is a string consisting of terminals and/or non-terminals.

Parse Trees and Their Role in Parsing

A parse tree, also known as derivation tree is a graphical representation of the derivation of a string according to the production rules of a CFG. It shows how a start symbol of a CFG can be transformed into a terminal string, using applying production rules in a sequence.

Each node in a parse tree represents a symbol of the grammar, while the edges represent the application of production rules.

- **Tree Nodes** – The nodes in a parse tree represent either terminal or non-terminal symbols of the grammar.
- **Tree Edges** – The edges represent the application of production rules leading from one node to another.

Types of Parsers

Parsers can be broadly categorized into two types based on how they construct the parse tree −

- **Top-down Parsers** − These parsers build the parse tree starting from the root and proceed towards the leaves. They typically perform a leftmost derivation. They expand the leftmost non-terminal at each step.
- **Bottom-up Parsers** − These parsers start from the leaves and move towards the root, performing a rightmost derivation in reverse order.

Example of Parse Tree Construction

Let us consider an example of a parse tree construction using a simple grammar −

Given grammar GG −

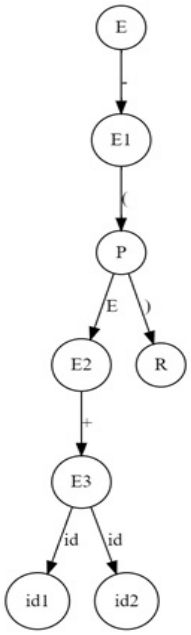
$$E \rightarrow E+E|E \times E|(E)||-E|idE \rightarrow E+E|E \times E|(E)||-E|id$$

Deriving the String "-(id + id)"

To determine whether the string **"-(id + id)"** is a valid sentence in this grammar, we can construct a parse tree as follows −

- Start with EE.
- Apply the production $E \rightarrow -EEE \rightarrow -EE$ to get the intermediate string "\mathrm{-E}".
- Apply $E \rightarrow (E)E \rightarrow (E)$ to derive "$-(E)-(E)$".
- Apply $E \rightarrow E+EE \rightarrow E+E$ to derive "$-(E+E)-(E+E)$".
- Finally, apply $E \rightarrow idE \rightarrow id$ to get the final string "$-(id+id)-(id+id)$".

Here is the corresponding **parse tree** −



This representation shows the hierarchical structure of the derivation, with the root representing the start symbol and the leaves representing the terminal symbols of the string.

Another Example of Parse Tree

Consider a different grammar for a second example –

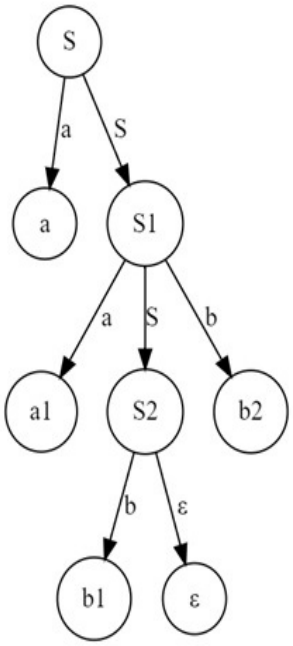
Given grammar GG

$$S \rightarrow SS|aSb|\varepsilon S \rightarrow SS|aSb|\varepsilon$$

To derive the string **"aabb"**, we proceed as follows –

- Start with SS.
- Apply S→aSbS→aSb to get "aSbaSb".
- Apply S→asbS→asb again within the non-terminal SS to derive "aaSbbaaSbb".
- Finally, apply S→εS→ε to replace the last SS, resulting in "aabbaabb".

The **corresponding parse tree** can be represented like below –



Ambiguity in Parse Trees

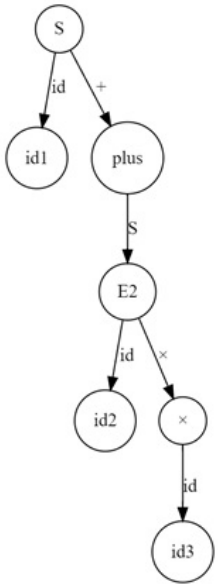
A grammar is said to be ambiguous if there exists at least one string that can be derived in more than one way. So there are multiple distinct parse trees. Ambiguity poses challenges when parsing as it can lead to multiple interpretations of the same string.

For instance, consider the following grammar −

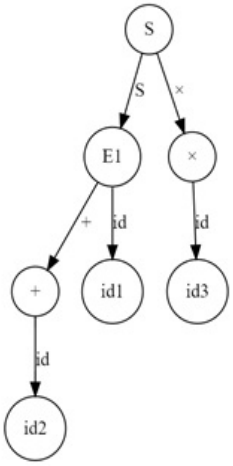
$$S \rightarrow S+S|S \times S|id S \rightarrow S+S|S \times S|id$$

The string "id + id × id" can have two different parse trees depending on the order of operations −

First Parse Tree



Second Parse Tree



Conclusion

Parse trees are a basic concept in Theory of Computation and Compiler Design. Parse trees are used to parse context-free languages. They provide a visual representation of how a string is derived from the grammars start symbol. It offers an insight into the structure and hierarchy of the language.

Ambiguity in Context-Free Grammars

If a context free grammar **G** has more than one derivation tree for some string **w ∈ L(G)**, it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

- If the grammar is not ambiguous then we call it unambiguous grammar.
- If the grammar has ambiguity then it is good for compiler construction.
- No method can automatically detect and remove the ambiguity, but we can remove the ambiguity by re-writing the whole grammar without ambiguity.

Example 1

Check whether the grammar G with production rules –

X → X+X | X*X |X| a

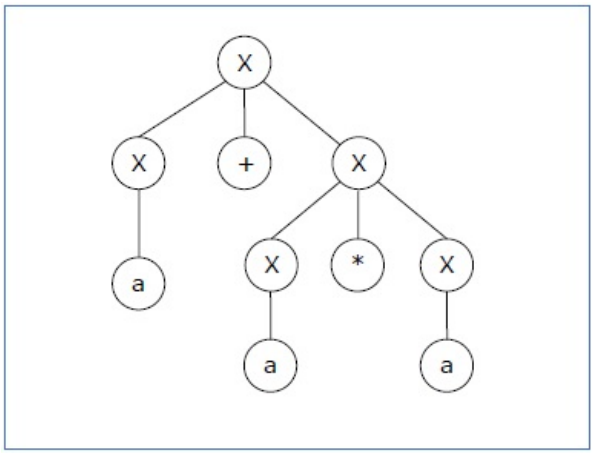
is ambiguous or not.

Solution

Lets find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

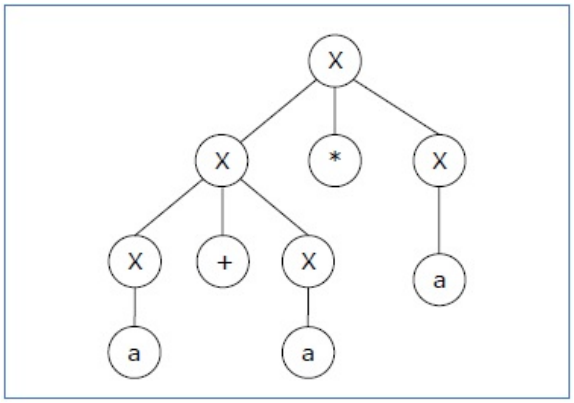
Derivation 1 – X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 –



Derivation 2 – X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 −



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

Example 2

Let us consider a grammar with production rules, as shown below −

$$E=IE=I$$

$$E=E+EE=E+E$$

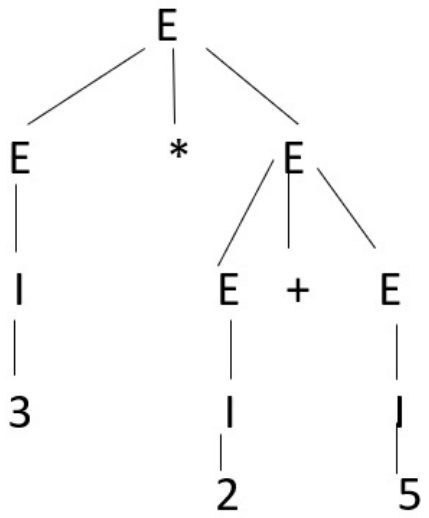
$$E=E\times EE=E\times E$$

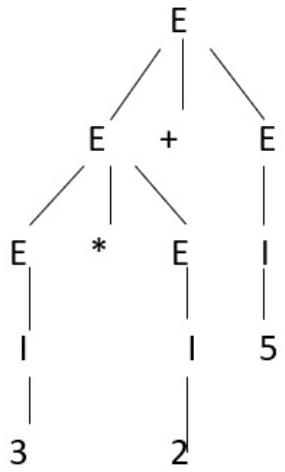
$$E=(E)E=(E)$$

$$E=\epsilon|0|1|2|3\ldots9E=\epsilon|0|1|2|3\ldots9$$

Let's consider a string "3*2+5"

If the above grammar generates two parse trees by using Left most derivation (LMD) then, we can say that the given grammar is ambiguous grammar.





Since there are two parse trees for a single string, then we can say the given grammar is ambiguous grammar.

Example 3

Consider another example,

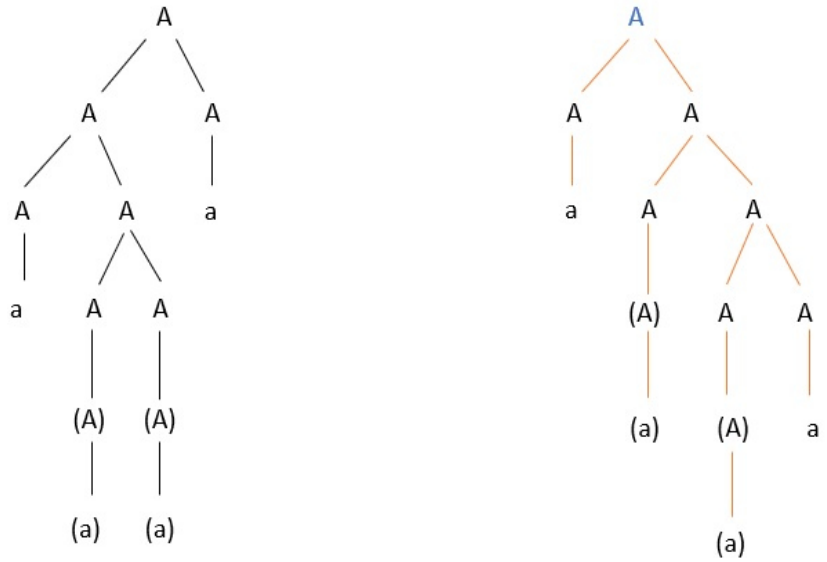
Check whether the grammar is ambiguous or not.

$$A \rightarrow AAA \rightarrow AA$$

$$A \rightarrow (A)A \rightarrow (A)$$

$$A \rightarrow aA \rightarrow a$$

For the string "a(a)(a)a" the above grammar can generate two parse trees, as given below –



Context-Free Grammar vs Regular Grammar

Regular Grammar

A regular grammar is formally represented as a four-part system, written as: **(V, T, P, S)**.

Where,

- **V** – Represents the set of non-terminals. These are like placeholders or variables in our grammar, typically denoted by uppercase letters.
- **T** – Represents the set of terminals. These are the actual input symbols of our language, often denoted by lowercase letters.
- **P** – Represents the set of production rules. These rules dictate how we can replace non-terminals with other symbols (terminals or non-terminals) to generate strings in our language.

- **S** – Represents the start symbol, a special non-terminal from which all derivations begin.

Context-Free Grammar

Just like regular grammar, a context-free grammar is also represented as a four-part system: (V, T, P, S). Here,

- **V** – Represents the non-terminals (uppercase letters).
- **T** – Represents the terminals (lowercase letters).
- **P** – Represents the production rules, although the rules for writing these productions differ between context-free and regular grammars (more on that later).
- **S** – Represents the start symbol, the starting point of our derivations.

The definitions seem almost identical! The key difference lies in the production rules, which we'll discuss in detail further on.

Parsing

Regular Grammar and Parsing – Here's a crucial difference, the regular grammars are not suitable for parsing. Remember how parsing involves breaking down an expression into its individual tokens (like we did with parse trees) Regular grammars lack the power to handle this process effectively.

Context-Free Grammar and Parsing – On the other hand, context-free grammars are perfectly suited for parsing. This capability makes them incredibly useful for analyzing the structure of languages.

This fundamental difference in parsing ability has significant implications for the applications of these grammars.

Automata Construction

Regular Grammars – We use regular grammars to construct Deterministic Finite Automata (DFAs). DFAs are computational models used to recognize patterns within strings.

Context-Free Grammars – We use context-free grammars to construct Pushdown Automata (PDAs). PDAs are more powerful than DFAs and can handle more complex languages. We'll explore PDAs in detail in future articles.

Representing Programming Languages

Regular Grammars – While useful for simpler tasks, regular grammars cannot fully represent the syntax of any programming language. Their limited expressive power restricts their use in this domain.

Context-Free Grammars – Context-free grammars, with their ability to be parsed and their more flexible production rules, can fully represent the syntax of any programming language. This feature makes them essential for compiler construction, where understanding the structure of programming languages is paramount.

Chomsky Hierarchy and Subsets

Chomsky Hierarchy – The Chomsky Hierarchy classifies grammars based on their generative power.

- Regular grammars are classified as Type 3 grammars.
- Context-free grammars are classified as Type 2 grammars.

Subsets – Importantly, every regular grammar can be considered a context-free grammar. In other words, regular grammars form a subset of context-free grammars.

However, the reverse is not true. Not every context-free grammar can be simplified into a regular grammar. This difference in expressiveness highlights the broader capabilities of context-free grammars.

Production Rules

Regular Grammar Production Rules – The production rules in regular grammars are highly restricted. They follow one of these forms –

- A → a:A non-terminal (A) can be replaced by a single terminal (a).
- A → aB: non-terminal (A) can be replaced by a terminal (a) followed by another non-terminal (B).
- A → Ba: non-terminal (A) can be replaced by a non-terminal (B) followed by a terminal (a).

Context-Free Grammar Production Rules – Context-free grammars offer much more flexibility in their production rules. The right-hand side of a production rule can be –

- A single terminal: A → a
- A single non-terminal: A → B
- A combination of terminals and non-terminals: A → aBcD

This flexibility in combining terminals and non-terminals gives context-free grammars their power to represent complex language structures.

Conclusion

In this chapter, we highlighted the differences between context-free grammar or CFG with regular grammars. Context-free grammars are more powerful than regular grammars due to their flexible production rules and parsing ability. They can represent programming language syntax and be used in compiler construction.

Applications of Context-Free Grammar

For a little recap, let us see the definitions of **context-free grammar**. Just like other grammars, we can define a context-free grammar using a four-tuple structure. These four components are –

- **V** – This represents the set of variables which are also known as non-terminals. Think of them as placeholders that can be replaced by other symbols.
- **T** – This is the set of symbols used in the grammar. We often call these terminals because they represent the final, indivisible elements of our language.
- **P** – This set of the production rules. These rules define how we can replace variables with other variables and terminals.
- **S** – Lastly, we have the start symbol. This specific non-terminal acts as the initial point from which we begin building strings in our language.

Importance of Production Rules

The defining characteristic of a CFG lies within its production rules. These rules are like to the following structure –

- **LHS** – Left-hand side, consisting of a single variable.
- **RHS** – Right-hand side, potentially consisting of:
  - Epsilon (ε), representing an empty string
  - A single terminal symbol
  - A string of terminals and/or variables

There is the flexibility of RHS. This distinguishes CFGs from regular grammars, which impose restrictions on the RHS, limiting it to a single terminal followed by a variable, a single variable followed by a terminal, or just a single terminal.

Let us take an **example** for a clear understanding. Consider a CFG designed to generate strings with at least two 'a's over the alphabet {a, b} –

```
V = {S, T}
T = {a, b}
P = {
  S -> TaTaT
  T -> aT | bT |
}
Start Symbol = S
```

In this example, 'S' and 'T' are variables, 'a' and 'b' are terminals, and the production rules define how variables can be replaced. For instance, the rule 'S → TaTaT'.

Applications of Context-Free Grammar

Let us see the applications of context-free grammars.

Programming Language Design and Implementation

CFGs are fundamental to defining the syntax of programming languages. The Backus-Naur Form (BNF) and Extended Backus-Naur Form (EBNF) notations, widely used for specifying programming language grammars. These are essentially context-free grammars. These grammars states the structure of programs, keeping the language's rules.

Compiler Construction: Parsing and Syntax Analysis

Compilers translate high-level code into machine-executable instructions. Compilers heavily rely on CFGs during the parsing phase. Parsing involves analyzing the syntactic structure of the input code based on the language's grammar.

CFGs are used for the construction of parsers, which are algorithms responsible for –

- **Lexical Analysis** – Grouping characters into meaningful units or tokens.
- **Syntax Analysis** – Verifying if the sequence of tokens conforms to the grammar rules.
- **Building Parse Trees** – Creating tree-like representations of the code's grammatical structure.

Markup Languages and Data Representation – Markup languages like HTML and XML, used for web development and data representation, they rely on CFG-based principles to define their structure. These languages use tags to define elements and attributes, and their hierarchical organization can be represented and validated using CFGs.

Natural Language Processing (NLP) – Natural language is inherently ambiguous and more complex than formal languages we learn in automata theory. The context-free grammars still play a significant role in NLP as well.

- **Grammar Formalisms** – CFGs are used in computational linguistics to model the grammatical structure of natural language, even if they can't capture all nuances.
- **Parsing Natural Language** – While challenging, parsing natural language with CFGs helps with tasks like identifying parts of speech and understanding sentence structure, these are crucial for applications like machine translation and sentiment analysis.

Conclusion

In this chapter, we explored the core idea of context-free grammars, starting from its definition and a simple example. Then, we highlighted a class of different application areas where contextfree grammars can be used. In the subsequent chapters of this tutorial, we will see context free grammars in further detail.

Left Recursion (LR) and Left Factoring (LF)

Context-free grammars play a vital role in in compilers and programming languages. CFGs are used to define how valid programs should be organized. We use recursion in parsing, whereby examining a sequence of tokens to its grammatical structure, is greatly dependent on these grammars.

In this chapter, we will cover two important characteristics of CFG which are **left recursion** and **left factoring**. Let us understand them through examples.

Left Recursion

A context-free grammar is said to be left recursive if it contains a production rule where the non-terminal on the left-hand side of the rule also appears as the first symbol on the right-hand side. In other words, the grammar is trying to define a non-terminal in terms of itself, creating a recursive loop.

This can be represented formally as –

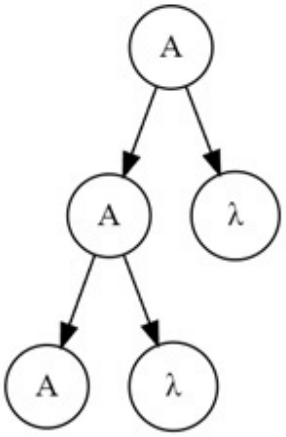
$$A \rightarrow A\alpha|\beta A \rightarrow A\alpha|\beta$$

Where –

- **A** is a non-terminal symbol.
- **α** represents a sequence of terminals and/or non-terminals.
- **β** represents another sequence of terminals and/or non-terminals.

The most important part here is the presence of A on both sides of the production rule, with it appearing first on the right-hand side.

To visualize this, consider the following **parse tree** –



It is generated by a left-recursive grammar. As the grammar recursively expands the non-terminal 'A' on the left, the tree grows indefinitely downwards on the left side. This continuous expansion makes it unsuitable for top-down parsing, as the parser could get trapped in an infinite loop, trying to expand 'A' repeatedly.

Problem of Left Recursion for Top-Down Parsing

As we have seen, the top-down parsing works by starting with the start symbol of the grammar and attempting to derive the input string by applying production rules.

When encountering a left-recursive rule, the parser keeps expanding the same non-terminal, leading to an infinite loop. This inability to handle left recursion directly is a significant drawback of top-down parsing methods.

Eliminating Left Recursion

To solve this we can eliminate immediate left recursion from a grammar without altering the language it generates. The general approach involves introducing a new non-terminal and rewriting the recursive rules.

Let's illustrate this with our previous example –

$$A \rightarrow A\alpha | \beta A \rightarrow A\alpha | \beta$$

We can eliminate the left recursion by introducing a new non-terminal 'A'` and rewriting the rule as follows –

$$A \rightarrow \beta A' A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \varepsilon A' \rightarrow \alpha A' | \varepsilon$$

In this **transformed grammar** –

- A now derives the non-recursive part 'β' followed by the new non-terminal A'.
- A' handles the recursion, deriving either ɑ A' (continuing the recursion) or ϵ (empty string), which terminates the recursion.

Let us see this through an example for a better view

Consider a simplified arithmetic expression grammar –

$$E \rightarrow E+T | T E \rightarrow E+T | T$$

$$T \rightarrow T*F | F T \rightarrow T*F | F$$

$$F \rightarrow (E) | id F \rightarrow (E) | id$$

This grammar contains left recursion in the rules for both `E` and `T`. Let's eliminate it –

- Eliminating Left Recursion in `E`:

$$E \rightarrow TE' E \rightarrow TE'$$

$$E' \rightarrow +TE' | \varepsilon E' \rightarrow +TE' | \varepsilon$$

- Eliminating Left Recursion in `T`:

$$T \rightarrow FT' T \rightarrow FT'$$

$$T' \rightarrow *FT' | \varepsilon T' \rightarrow *FT' | \varepsilon$$

The final transformed grammar, free from left recursion, becomes –

$$E \rightarrow TE' E \rightarrow TE'$$

$$E' \rightarrow +TE' | \varepsilon E' \rightarrow +TE' | \varepsilon$$

$$T \rightarrow FT' T \rightarrow FT'$$

$$T' \rightarrow *FT' | \varepsilon T' \rightarrow *FT' | \varepsilon$$

$$F \rightarrow (E) | id F \rightarrow (E) | id$$

Left Factoring

After the left recursion, let us see the idea of left factoring. While left recursion presents a parsing challenge, left factoring is a desirable property for top-down parsing. It involves restructuring the grammar to eliminate common prefixes in production rules.

Importance of left factoring

When a grammar has multiple production rules for a non-terminal with a common prefix, the parser faces ambiguity during the parsing process. It has to choose between these rules without knowing which one will ultimately lead to a successful parse.

Left factoring helps in resolving this ambiguity by postponing the decision point, making the parsing process more efficient.

Performing Left Factoring

The process of left factoring involves identifying the longest common prefix (α) in the alternatives for a non-terminal and rewriting the rules to factor out this common prefix.

Consider a grammar with the following rules –

$$A \rightarrow \alpha\beta 1 | \alpha\beta 2 | \ldots | \alpha\beta n | \gamma A \rightarrow \alpha\beta 1 | \alpha\beta 2 | \ldots | \alpha\beta n | \gamma$$

Here, 'α' is the longest common prefix in the first 'n' alternatives for non-terminal 'A'.

We can left factor this as follows –

$$A \rightarrow \alpha A' | \gamma A \rightarrow \alpha A' | \gamma$$

$$A \rightarrow \beta 1 | \beta 2 | \ldots | \beta n A \rightarrow \beta 1 | \beta 2 | \ldots | \beta n$$

In this **factored grammar** −

- A now derives either the common prefix 'α' followed by the new non-terminal 'A'` or the alternative 'γ'.
- A'` encapsulates the different options that were originally prefixed by 'α'. This defers the decision of which alternative to choose until more of the input has been processed.

Let us understand this through an **example**. Let's consider a **simple grammar** −

$$S \rightarrow \text{if E then S, else S} S \rightarrow \text{if E then S, else S}$$

$$S \rightarrow \text{if E then S} S \rightarrow \text{if E then S}$$

$$S \rightarrow A S \rightarrow A$$

Here, the first two rules for 'S' have a common prefix, "if E then". We can apply left factoring to obtain −

$$S \rightarrow \text{if E then S S' | A} S \rightarrow \text{if E then S S' | A}$$

$$S' \rightarrow \text{else S | } \varepsilon S' \rightarrow \text{else S | } \varepsilon$$

This factored grammar is more suitable for top-down parsing as it avoids the initial choice between the first two rules.

Conclusion

Left recursion and left factoring are two important concepts in compiler design and more specifically for context-free grammars and parsing.

Left recursion can be problematic for top-down parsing and needs to be eliminated. Left factoring, on the other hand, improves the efficiency of top-down parsing by reducing ambiguity.