Contents

Module 3

1 Context-Free Grammars

Context Free Grammars: Definition and designing CFG's, Derivations Using a Grammar, Parse Trees, Ambiguity and Elimination of Ambiguity, Elimination of Left Recursion, Left factoring

Syntax Analysis Phases of Compilers: **part-1:** Role of Parser, Top-Down Parsing.

3.1 What is Grammar?

In the theory of computation, a grammar is a set of rules that specify the structure of valid strings in a formal language. The formal definition of a grammar involves the use of production rules that describe how symbols (both non-terminal and terminal) can be combined to generate strings in the language.

The formal definition of grammar is that it is defined as four tuples −

G=(V,T,P,S)

G is a grammar, which consists of a set of production rules.

It is used to generate the strings of a language.

T is the final set of terminal symbols. It is denoted by lowercase letters.

V is the final set of non-terminal symbols. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminal symbols (on the left side of production) in a string with other terminals (on the right side of production).

S is the start symbol used to derive the string.

Grammar is composed of two basic elements

Terminal Symbols - Terminal symbols are the components of the sentences that are generated using grammar and are denoted using small case letters like a, b, c etc.

Non-Terminal Symbols - Non-Terminal Symbols take part in the generation of the sentence but are not the component of the sentence. These types of symbols are also called Auxiliary Symbols and Variables. They are represented using a capital letter like A, B, C, etc.

Consider a grammar example1

G = (V , T , P , S)

Where,

V = { S , A , B } ⇒ Non-Terminal symbols

T = { a , b } ⇒ Terminal symbols

Production rules P = { S → ABa , A → BB , B → ab , AA → b }

S = { S } ⇒ Start symbol

Consider a grammar example2

G=(V,T,P,S)

Where,

V= {S, A, B} ⇒ non terminal symbols

T = { 0,1} ⇒ terminal symbols

Production rules P = { S→A1B

A→0A| ε

B→0B| 1B| ε }

S= {S} ⇒ start symbol.

The formal study of grammars in the context of mathematics and theoretical computer science was significantly influenced by Noam Chomsky. Noam Chomsky, an American linguist, cognitive scientist, and political theorist, introduced the concept of grammars in the 1950s. Chomsky proposed a hierarchy of formal languages and grammars, known as the Chomsky hierarchy, which categorizes grammars into different types based on their generative power. This hierarchy is widely used in the theory of computation and formal language theory.

The key types of grammars in the Chomsky hierarchy are:.

Grammar Type	Grammar Accepted	Language Accepted	Automaton	Explanation
Type 3 - Regular Grammars	Regular Grammar	Regular Languages	Finite Automaton	Patterns that can be recognized by simple pattern matching.
Type 2 - CFG	Context-Free Grammar (CFG)	Context-Free Languages	Pushdown Automaton	Describes the syntax of programming languages.
Type 1 - CSG	Context-Sensitive Grammar	Context-Sensitive Languages	Linear Bounded Automaton	More powerful than CFGs, used in natural language processing.

Type 0 - Unrestricted	Unrestricted Grammar	Recursively Enumerable Languages	Turing Machine	The most powerful, can describe any computable language.

3.2 Definition and designing CFG's

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Context-free grammars are studied in fields of theoretical computer science, compiler design, and linguistics. CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars.

Context-free grammars have the following components:

A set of terminal symbols which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.

A set of nonterminal symbols (or variables) which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.

A set of production rules which are the rules for replacing nonterminal symbols. Production rules have the following form:

variable → string of variables and terminals.

A start symbol which is a special nonterminal symbol that appears in the initial string generated by the grammar.

Formal Definition

A context-free grammar can be described by a four-element tuple (V,Σ,R,S), where

V is a finite set of variables (which are non-terminal);

Σ is a finite set (disjoint from V) of terminal symbols;

R is a set of production rules where each production rule maps a variable to a string $s \in (V \cup \Sigma)*$;

S (which is in V) which is a start symbol.

Example: a grammar that will generate the context-free (and also regular) language that contains all strings with matched parentheses.

Many grammars can do this task. This solution is one way to do it, but should give you a good idea of if your (possibly different) solution works too.

S → ( )
S → SS
S → (S)

and translate them into a single line: S → ( ) | SS | (S) | ,ϵ, where ϵ is an empty string.

a context-free grammar that generates arithmetic expressions (subtraction, addition, division, and multiplication)[1].

Start symbol = <expression>

Terminal symbols = {+,−,∗,/,(,),number},{+,−,∗,/,(,),number}, where “number” is any number

Production rules:

<expression> →number

<expression> → (<expression>)

<expression> →<expression> + <expression>

<expression> → <expression> - <expression>

<expression> → <expression> * <expression>

<expression> → <expression> / <expression>

This allows us to construct whatever expressions using multiplication, addition, division, and subtraction we want. What these production rules tell us is that the result of any operation, for example, multiplication, is also an expression (denoted, <expression>).

Why Context-Free Grammar?

Context-free grammars (CFGs) are widely used in formal language theory and the design of programming languages due to several advantages:

- **Simplicity**: CFGs are relatively simple and easy to understand. The production rules define how different components of a language can be combined without being influenced by the context in which they appear. This simplicity makes CFGs a convenient tool for describing the syntax of languages.
- **Parsing**: CFGs are well-suited for parsing, which is the process of analyzing a sequence of symbols to determine its grammatical structure. Many parsing algorithms, such as the CYK (Cocke-Younger-Kasami) algorithm and the Early parser, work efficiently with CFGs. This makes it practical to implement compilers, interpreters, and other language processing tools.

- **Formal Foundation**: CFGs provide a formal foundation for describing the structure of languages. They help in the precise definition of syntactic rules, allowing for unambiguous interpretation and implementation.
- **Expressiveness**: Despite their simplicity, CFGs are expressive enough to describe a wide range of languages, including programming languages. They can capture hierarchical and recursive structures commonly found in natural and programming languages.
- **Language Design**: When designing a new programming language, specifying its syntax using a CFG is a common practice. The grammar serves as a blueprint for the language's structure, guiding compiler or interpreter development.
- **Standard Representation**: CFGs provide a standard and widely accepted way to represent the syntax of a language. This facilitates communication and understanding among developers, language designers, and researchers working on language-related topics.
- **Connection to Automata Theory**: CFGs are closely connected to automata theory, particularly pushdown automata. This connection allows for a deeper understanding of the relationship between syntax and computation.

Despite these advantages, it's essential to note that CFGs have limitations. They cannot capture certain aspects of languages that require more complex mechanisms, such as context-sensitive or context-free grammars with semantic actions. For a complete understanding of a language, both syntax (described by a CFG) and semantics (meaning and behavior) must be considered.

1. Construct the CFG for the language having any number of a's over the set ∑= {a}.

To construct a Context-Free Grammar (CFG) for the language that generates any number of 'a's over the set

Σ={a}, you want to create rules that allow the repetition of the symbol 'a'. The language includes strings with zero or more 'a's.

The CFG can be represented as follows:

G=({S},{a},R,S)

Here:

S is the start symbol.

{a} is the alphabet.

R is the set of production rules.

The production rules can be defined as:

S→aS (This rule allows the addition of 'a' to the string.)

S→ε (This rule allows the generation of an empty string.)

These rules mean that starting with the start symbol

S, you can either add an 'a' (using rule 1) or generate an empty string (using rule 2).

Using this CFG, you can generate strings with any number of 'a's or an empty string. For example:

Applying rule 1 twice:

S→aS→aaS

Applying rule 1 three times:

S→aS→aaS→aaaS

Applying rule 2:

S→ε

Thus, the language generated by this CFG includes strings like "aaa," "aaaaa," and the empty string.

2. Construct a CFG for the regular expression (0+1)*

To construct a Context-Free Grammar (CFG) for the regular expression

(0+1)* , which represents the set of all strings over the alphabet

Σ={0,1}, you need to define rules that generate any combination of '0's and '1's including the empty string.

Let's construct the CFG:

G=({S},{0,1},R,S)

Here:

S is the start symbol.

{0,1} is the alphabet.

R is the set of production rules.

The production rules can be defined as:

S→0S (This rule allows the addition of '0' to the string.)

S→1S (This rule allows the addition of '1' to the string.)

S→ε (This rule allows the generation of an empty string.)

OR

S → 0S | 1S

S → ε

These rules mean that starting with the start symbol

S, you can either add a '0' (using rule 1), add a '1' (using rule 2), or generate an empty string (using rule 3).

Using this CFG, you can generate strings like "0101," "0011," "111," and the empty string. The regular expression

(0+1)* represents any combination (including none) of '0's and '1's.

3. Construct a CFG for a language L = {wcwR | where w € (a, b)*}.

The string that can be generated for a given language is {aacaa, bcb, abcba, bacab, abbcbba, ...}

The CFG can be represented as follows:

G={S,{a,b,c},R,S)

Here:

S is the start symbol.

{[a,b,c} is the alphabet.

R is the set of production rules.

The grammar could be:

S → aSa rule 1

S → bSb rule 2

S → c rule 3

Now if we want to derive a string "abbcbba", we can start with start symbols.

S → aSa

S → abSba from rule 2

S → abbSbba from rule 2

S → abbcbba from rule 3

Construct a CFG for the language L = anb2n where n>=1.

The string that can be generated for a given language is {abb, aabbbb, aaabbbbbb....}.

The grammar could be:

S → aSbb | abb

Now if we want to derive a string "aabbbb", we can start with start symbols.

S → aSbb

S → aabbbb

Write CFG to generate balanced parenthesis where Bal ={w ∈{),(}*; parenthesis balanced }

To generate a Context-Free Grammar (CFG) for the language

Bal ={w ∈{),(}*; parenthesis balanced }, where the language consists of strings with balanced parentheses, you can use the following CFG:

G=({S},{(,)},R,S)

Here:

S is the start symbol.

{(,)} is the alphabet, representing the two types of parentheses.

R is the set of production rules.

The production rules can be defined as follows:

S→ε (This rule allows the generation of an empty string.)

S→(S) (This rule allows the addition of a pair of balanced parentheses.)

S→SS (This rule allows concatenation of two strings with balanced parentheses.)

These rules ensure that any derivation of S will generate a string with balanced parentheses. Rule 1 allows the generation of an empty string, and rules 2 and 3 allow the addition of balanced pairs of parentheses and their concatenation.

Using this CFG, you can generate strings like "()", "(())", "((()))", and so on, where the parentheses are balanced.

4. Write CFG to generate for the following

i)L1 ={W : \|w\| Mod 3 = 0 } over € ={a}

To generate a Context-Free Grammar (CFG) for the language

L1 ={W : |w| Mod 3 = 0 } over € ={a}, where the length of the string is a multiple of 3, we can construct the following CFG:

G=({S},{a},R,S)

Here:

S is the start symbol.

{a} is the alphabet.

R is the set of production rules.

The production rules can be defined as follows:

S→AAA

A→aaA

A→ε

These rules generate strings in the language

L , where the length of the string is a multiple of 3. The non-terminal

A represents a block of two 'a's, and

S ensures that the string has a length divisible by 3 by having three blocks of 'a's.

Using this CFG, you can generate strings like "aaa", "aaaaaa", "aaaaaaaa", and so on, where the length of the string is a multiple of 3.

iI) L2 ={anbmck: m=n+k } over € ={a,b,c}

To generate a Context-Free Grammar (CFG) for the language

L2 ={anbmck: m=n+k } over € ={a,b,c}

Where the number of 'b's is the sum of the number of 'a's and 'c's, we can construct the following CFG:

G=({S,A,B},{a,b,c},R,S)

Here:

S is the start symbol.

{a,b,c} is the alphabet.

R is the set of production rules.

The production rules can be defined as follows:

S→aBc

B→aBb|ε

These rules generate strings in the language

L2 , where the number of 'b's is the sum of the number of 'a's and 'c's. The non-terminal

B represents a balanced number of 'a's and 'c's followed by 'b's.

Using this CFG, you can generate strings like "abbc", "aabbbc", "aaabbbbc", and so on, where the number of 'b's is the sum of the number of 'a's and 'c's.

5. write context free grammars for the following languages

a){aibjck | i=j+k }

To design a context-free grammar (CFG) for the language

{aibjck | i=j+k }, where the number of 'a's is equal to the sum of the number of 'b's and 'c's, you can use the following CFG:

CFG:

S→AB

A→aAc|ε

B→bBc|ε

Non-terminal S is the start symbol, and it generates the entire string.

Non-terminal A generates the 'a's and ensures that the number of 'a's is equal to the sum of the number of 'b's and 'c's.

Non-terminal B generates the 'b's and 'c's in pairs, ensuring the equality condition.

This CFG ensures that the language generated consists of strings where the number of 'a's is equal to the sum of the number of 'b's and 'c's

b) {aibj | i <=2j }

design a context-free grammar (CFG) for the language {aibj | i <=2j }, where the number of 'a's is less than or equal to twice the number of 'b's, you can use the following CFG:

CFG:

S→Ab|ε

A→aAab|ε

Non-terminal S is the start symbol, and it generates the entire string.

Non-terminal A generates the 'a's in pairs, ensuring that the number of 'a's is less than or equal to twice the number of 'b's.

The production S→ε allows the empty string in the language.

This CFG ensures that the language generated consists of strings where the number of 'a's is less than or equal to twice the number of 'b's.

c) The set of all strings of 0's and 1's where the number of O's is equal to the number of 1's

To generate the language consisting of all strings of 0's and 1's where the number of 0's is equal to the number of 1's, you can use the following context-free grammar (CFG):

CFG:

S→0S1S|1S0S|ε

Non-terminal S is the start symbol, and it generates pairs of '0's and '1's.

The production S→0S1S generates a '0' followed by another '0' and '1'.

The production S→1S0S generates a '1' followed by another '1' and '0'.

The production S→ε allows the empty string in the language.

This CFG ensures that the generated strings have an equal number of '0's and '1's

d) The set of all strings of 0's and 1s where the number of 0's is not equal to the number of 1's

To generate the language consisting of all strings of 0's and 1's where the number of 0's is not equal to the number of 1's, you can use the following context-free grammar (CFG):

CFG:

S→0S1S|1S0S|0S|1S|ε

Non-terminal : S is the start symbol, and it generates pairs of '0's and '1's.

The production : S→0S1S generates a '0' followed by another '0' and '1'.

The production : S→1S0S generates a '1' followed by another '1' and '0'.

The productions : S→0S and S→1S allow for additional '0's or '1's without a matching pair.

The production S→ε allows the empty string in the language.

3.3 Derivations Using a Grammar

In the context of formal languages and grammars, a derivation is a sequence of production rule applications that transform a given symbol string according to the rules of a grammar. It's a step-by-step process that starts with the start symbol and applies production rules to generate or recognize a specific string in the language defined by the grammar.

Derivations in formal language theory and grammars are essential for understanding how strings are generated or recognized within a given language. Derivations provide a step-by-step process of applying production rules to transform an initial symbol (usually the start symbol) into a specific string according to the rules defined by the grammar.

These derivations are crucial for several reasons:

- Understanding Language Structure: Derivations provide insight into the hierarchical structure of the generated strings in a language.

- They show how complex strings can be built from simpler components, revealing the composition of the language.
- Grammar Analysis: Derivations help in analyzing the structure and rules of a grammar. They provide a formal way to verify whether a given string is in the language generated by the grammar.
- Language Generation: Derivations serve as a systematic way to generate valid strings in a language.By following the steps of a derivation, you can construct strings that adhere to the grammar's rules.
- Proof of Language Membership: Derivations can be used as a proof that a particular string belongs to the language generated by the grammar. If a derivation exists for a given string, it demonstrates the string's validity in the language.
- Parsing and Compilation: Derivations are foundational in parsing, the process of analyzing the syntactic structure of strings. In compiler construction, derivations help understand the structure of programming languages.

Leftmost and Rightmost Derivations:

Leftmost and rightmost derivations represent different strategies for applying production rules during the derivation process.
These derivations are useful for understanding the order in which non-terminals are expanded, which can be relevant in certain parsing algorithms and in analyzing the structure of generated strings.
In summary, derivations are a fundamental concept in formal language theory, providing a systematic and structured way to generate or recognize strings according to the rules defined by a grammar. They are a key tool for studying the properties of languages and grammars.

Leftmost Derivation: In a leftmost derivation, the leftmost non-terminal in the current sentential form is replaced at each step. The process begins with the start symbol, and in each step, the leftmost non-terminal is selected for replacement.

Example of Leftmost Derivation:

Consider the grammar:

S→AB
A→a
B→b|ε

Starting with the start symbol
S⇒AB (Replace S with AB)
AB⇒aB (Replace leftmost A with a)
aB⇒ab (Replace leftmost B with b)
The leftmost derivation produces the string "ab" by repeatedly replacing the leftmost non-terminal in the current sentential form.

Rightmost Derivation:

In a rightmost derivation, the rightmost non-terminal in the current sentential form is replaced at each step. The process starts with the start symbol, and in each step, the rightmost non-terminal is selected for replacement.

Example of Rightmost Derivation:
Using the same grammar, let's derive the string "ab" using a rightmost derivation:
S⇒AB (Replace S with AB)
AB⇒Ab (Replace rightmost B with b)
Ab⇒ab (Replace rightmost A with a)
The rightmost derivation produces the string "ab" by repeatedly replacing the rightmost non-terminal in the current sentential form.

In summary, both leftmost and rightmost derivations are methods for generating or recognizing strings in a formal language. They specify the order in which non-terminals are replaced in the course of transforming a string according to the rules of a grammar.

1. Derive the string "abb" for leftmost derivation and rightmost derivation using a CFG given by,

S → AB | ε

A → aB

B → Sb

Leftmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AB$:**
   AB
3. **Apply $A \to aB$:**
   aB
4. **Apply $B \to Sb$:**
   asb
5. **Apply $S \to AB$:**
   $absb$
6. **Apply $A \to aB$:**
   $abbsb$
7. **Apply $B \to \varepsilon$:**
   abb

So, the leftmost derivation for "abb" is $S \Rightarrow AB \Rightarrow aB \Rightarrow absb \Rightarrow abbsb \Rightarrow abb$.

Rightmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to \varepsilon$:**
   ε
3. **Apply $B \to Sb$:**
   Sb
4. **Apply $S \to AB$:**
   Ab
5. **Apply $A \to aB$:**
   ab
6. **Apply $B \to \varepsilon$:**
   abb

So, the rightmost derivation for "abb" is $S \Rightarrow \varepsilon \Rightarrow Sb \Rightarrow Ab \Rightarrow aB \Rightarrow abb$.

2. Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,
S → aB | bA
S → a | aS | bAA
S → b | aS | aBB

Leftmost Derivation:

1. **Start with** S**:**
   S
2. **Apply** $S \to aS$**:**
   aS
3. **Apply** $S \to aS$**:**
   aaS
4. **Apply** $S \to bAA$**:**
   $aabAA$
5. **Apply** $S \to aS$**:**
   $aabaBB$
6. **Apply** $S \to b$**:**
   $aababB$
7. **Apply** $S \to b$**:**
   $aababb$
8. **Apply** $S \to aB$**:**
   $aabbB$

9. **Apply** $S \to bA$**:**
   $aabbabA$
10. **Apply** $S \to b$**:**
    $aabbabb$
11. **Apply** $S \to aB$**:**
    $aabbabB$
12. **Apply** $S \to bA$**:**
    $aabbabbA$
13. **Apply** $S \to aB$**:**
    $aabbabBA$
14. **Apply** $S \to bA$**:**
    $aabbabbAA$
15. **Apply** $S \to \varepsilon$**:**
    $aabbabba$

So, the leftmost derivation for "aabbabba" is $S \Rightarrow aS \Rightarrow aaS \Rightarrow aabAA \Rightarrow aabaBB \Rightarrow aababB \Rightarrow aababb \Rightarrow aabbB \Rightarrow aabbabA \Rightarrow aabbabb \Rightarrow aabbabB \Rightarrow aabbabbA \Rightarrow aabbabBA \Rightarrow aabbabbAA \Rightarrow \varepsilon \Rightarrow aabbabba.$

Rightmost Derivation:

1. **Start with** S**:**
   S
2. **Apply** $S \to bAA$**:**
   bAA
3. **Apply** $S \to aBB$**:**
   $baBB$
4. **Apply** $S \to b$**:**
   $babB$
5. **Apply** $S \to b$**:**
   $babb$
6. **Apply** $S \to aB$**:**
   $aabB$
7. **Apply** $S \to aS$**:**
   $aabbB$
8. **Apply** $S \to bA$**:**
   $aabbabA$

9. **Apply** $S \to b$**:**
   $aabbabb$
10. **Apply** $S \to aB$**:**
    $aabbabB$
11. **Apply** $S \to bA$**:**
    $aabbabbA$
12. **Apply** $S \to aB$**:**
    $aabbabBA$
13. **Apply** $S \to bA$**:**
    $aabbabbAA$
14. **Apply** $S \to \varepsilon$**:**
    $aabbabba$

So, the rightmost derivation for "aabbabba" is $S \Rightarrow bAA \Rightarrow baBB \Rightarrow babB \Rightarrow babb \Rightarrow aabB \Rightarrow aabbB \Rightarrow aabbabA \Rightarrow aabbabb \Rightarrow aabbabB \Rightarrow aabbabbA \Rightarrow aabbabBA \Rightarrow aabbabbAA \Rightarrow \varepsilon \Rightarrow aabbabba.$

3. Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

- ***S → A1B***
- ***A → 0A | ε***
- ***B → 0B | 1B | ε***

1. **Start with** S**:**
   S
2. **Apply** $S \to A1B$**:**
   $A1B$
3. **Apply** $A \to 0A$**:**
   $01B$
4. **Apply** $A \to 0A$**:**
   $001B$
5. **Apply** $B \to 0B$**:**
   $0010B$
6. **Apply** $B \to 0B$**:**
   $00100B$
7. **Apply** $B \to 1B$**:**
   $001001B$
8. **Apply** $B \to \varepsilon$**:**
   001001

So, the leftmost derivation for "00101" is $S \Rightarrow A1B \Rightarrow 01B \Rightarrow 001B \Rightarrow 0010B \Rightarrow 00100B \Rightarrow 001001B \Rightarrow 001001.$

Rightmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to A1B$:**
   $A1B$
3. **Apply $B \to \varepsilon$:**
   $A1$
4. **Apply $A \to \varepsilon$:**
   1
5. **Apply $B \to 1B$:**
   $101B$
6. **Apply $B \to \varepsilon$:**
   101
7. **Apply $A \to \varepsilon$:**
   101

So, the rightmost derivation for "00101" is $S \Rightarrow A1B \Rightarrow A1 \Rightarrow 1 \Rightarrow 101$.

4. ***The following grammar generates the grammar of the language consisting of all strings as even length***
   S ->A S | ε
   A ->aa | ab | ba | bb
   Give left most and right most derivations for the following strings

   a)aabbba

Leftmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AS$:**
   AS
3. **Apply $A \to aa$:**
   aaS
4. **Apply $S \to AS$:**
   $aaaS$
5. **Apply $A \to bb$:**
   $aaabbS$
6. **Apply $S \to \varepsilon$:**
   $aaabb$

So, the leftmost derivation for "aabbba" is $S \Rightarrow AS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaabbS \Rightarrow aaabb.$

Rightmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AS$:**
   AS
3. **Apply $A \to bb$:**
   $AbbaS$
4. **Apply $S \to AS$:**
   $AbbAS$
5. **Apply $A \to aa$:**
   $abbaaaS$
6. **Apply $S \to \varepsilon$:**
   $abbaaa$

So, the rightmost derivation for "aabbba" is $S \Rightarrow AS \Rightarrow AbbAS \Rightarrow abbaaaS \Rightarrow abbaaa.$

b) baabab

Leftmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AS$:**
   AS
3. **Apply $A \to ba$:**
   baS
4. **Apply $S \to AS$:**
   $baAS$
5. **Apply $A \to ab$:**
   $baabS$
6. **Apply $S \to AS$:**
   $baabAS$
7. **Apply $A \to bb$:**
   $baabbbS$
8. **Apply $S \to \varepsilon$:**
   $baabbb$

So, the leftmost derivation for "baabab" is $S \Rightarrow AS \Rightarrow baS \Rightarrow baAS \Rightarrow baabS \Rightarrow baabAS \Rightarrow baabbbS \Rightarrow baabbb.$

Rightmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AS$:**
   AS
3. **Apply $A \to bb$:**
   $AbbaS$
4. **Apply $S \to AS$:**
   $AbbAS$
5. **Apply $A \to aa$:**
   $abbaaaS$
6. **Apply $S \to \varepsilon$:**
   $abbaaa$

So, the rightmost derivation for "baabab" is $S \Rightarrow AS \Rightarrow AbbAS \Rightarrow abbaaaS \Rightarrow abbaaa.$

c) aaabbb

Leftmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AS$:**
   AS
3. **Apply $A \to aa$:**
   aaS
4. **Apply $S \to AS$:**
   $aaAS$
5. **Apply $A \to ab$:**
   $aaabS$
6. **Apply $S \to AS$:**
   $aaabAS$
7. **Apply $A \to bb$:**
   $aaabbbS$
8. **Apply $S \to \varepsilon$:**
   $aaabbb$

So, the leftmost derivation for "aaabbb" is $S \Rightarrow AS \Rightarrow aaS \Rightarrow aaAS \Rightarrow aaabS \Rightarrow aaabAS \Rightarrow aaabbbS \Rightarrow aaabbb$.

Rightmost Derivation:

1. **Start with S:**
   S
2. **Apply $S \to AS$:**
   AS
3. **Apply $A \to bb$:**
   $AbbS$
4. **Apply $S \to AS$:**
   $abbaAS$
5. **Apply $A \to aa$:**
   $abbabaaS$
6. **Apply $S \to \varepsilon$:**
   $abbabaa$

So, the rightmost derivation for "aaabbb" is $S \Rightarrow AS \Rightarrow AbbS \Rightarrow abbaAS \Rightarrow abbabaaS \Rightarrow abbabaa$.

5. Consider the CFG G Defined by productions

S->aS | Sb | a | b

a) prove by induction on the string length that no string in L(G) has ba as a substring

b) Describe L(G) informally

a) Prove by Induction:

Base Case (String length = 1):
The productions are
S→a and
S→b.
For both cases, the derived strings have length 1, and "ba" cannot be a substring.
Inductive Step:
Assume that for any string of length n derived from S, "ba" is not a substring.
Now, let's consider the productions:
S→aS
S→Sb
For the first production, adding a to the beginning of a string that does not contain "ba" as a substring will still not introduce "ba" as a substring.
For the second production, adding b to the end of a string that does not contain "ba" as a substring will still not introduce "ba" as a substring.

Therefore, by induction, we can conclude that no string in the language
L(G) has "ba" as a substring.

b) Description of L(G) informally:
The CFG G generates strings in the language
L(G) where "a" and "b" can be arranged in any order, but "ba" is not allowed as a substring. Informally, the language consists of strings that have the same number of "a"s and "b"s, and "ba" is avoided. The grammar allows for arbitrary interleaving of "a" and "b" characters, but it enforces a constraint that prevents the occurrence of "ba" in the generated strings.

6. Define context-free grammar ?Write CFG for the following languages L={0m1m2n ;m>=0,n>=0}.

A context-free grammar (CFG) is a formal grammar that describes the syntactic structure of a language in terms of a set of production rules. These rules define how strings of symbols in the language can be generated through a set of rewriting rules. A CFG consists of a set of terminal symbols, a set of non-terminal symbols, a start symbol, and a set of production rules.

The language L={$0^m1^m2^n$; m≥0, n≥0} consists of strings that start with zero (0) followed by one (1), and then followed by two (2), where the number of 0s and 1s is the same (m) and the number of 2s can be any non-negative integer (n).
A context-free grammar for this language can be defined as follows:
Set of terminal symbols (Σ): {0, 1, 2}
Set of non-terminal symbols (V): {S, A, B}
Start symbol (S): S
Production rules (P):

S -> 0AB | ε (ε represents an empty string)
A -> 1A | ε
B -> 2B | ε

Here's an explanation of the production rules:

The start symbol S generates strings starting with 0, followed by A, and then followed by B. This ensures that the 0s are followed by 1s and then by 2s.
Non-terminal A generates strings with 1s. It can generate any number of 1s (including none, represented by ε).
Non-terminal B generates strings with 2s. It can generate any number of 2s (including none, represented by ε).
The CFG ensures that the language L is generated according to the specified conditions (m≥0, n≥0) where the number of 0s equals the number of 1s, and there can be any number of 2s.

3.4 Parse Tree or Derivation Tree

A parse tree, also known as a derivation tree, is a tree representation that illustrates the syntactic structure of a string according to the rules of formal grammar, particularly in the context of parsing in computer science. It provides a graphical representation of the steps taken during the derivation of a string from the start symbol of a context-free grammar (CFG). The parse tree visually displays how the production rules of the grammar are applied to generate the given string.
In other words, A derivation tree or parse Tree is an ordered rooted tree that graphically represents semantic information of strings derived from context-free grammar.
Let's break down the key concepts:

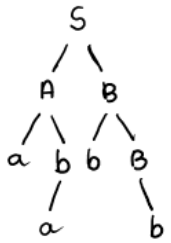
- **Nodes and Edges**: Each node in the parse tree represents a symbol in the grammar (either a terminal or non-terminal).
  Edges between nodes represent the application of a production rule.
- **Root Node**: The topmost node of the tree is the root node, representing the start symbol of the grammar.
- **Leaves**: The leaves of the tree represent the individual symbols in the derived string.
- **Internal Nodes**: Internal nodes represent non-terminals in the grammar.
- **Path from Root to Leaves**: The path from the root to the leaves corresponds to the derivation of the string.

- **Subtrees**: Subtrees represent the application of a specific production rule, and they can be considered as part of the larger parse tree.

For example, consider the CFG:

S -> A | B
A -> "a" A | ε
B -> "b" B | ε

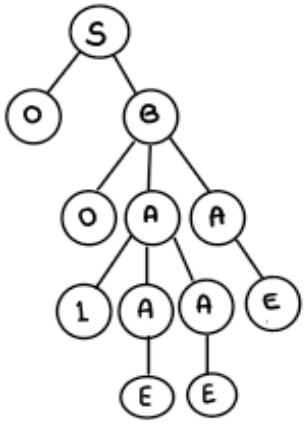
Now, let's say you want to derive the string "aaabbb" using the CFG. The parse tree might look like this:



In this parse tree, each level represents a step in the derivation process. The string "aaabbb" can be read by traversing the leaves from left to right. The internal nodes represent non-terminals, and the edges represent the production rules applied during the derivation.

One more example

Example : For the grammar G= (V,T,P,S) where S-OB, A-IAA|ϵ, B-OAA

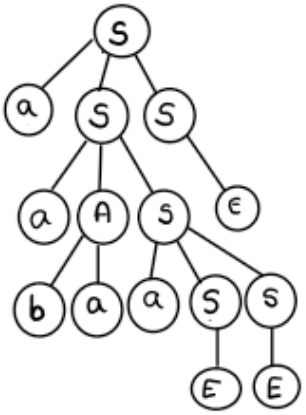


Root vertex : Must labelled by the start symbol
Vertex : labelled by Non- Terminal symbols
Leaves : Labelled by Terminal symbols or ϵ

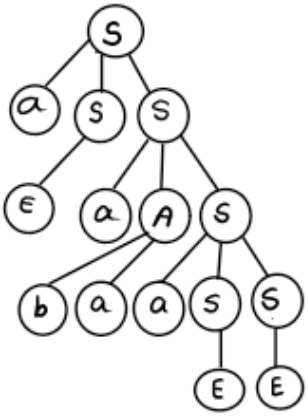
Left Derivation Tree
Left Derivation Tree is obtained by applying production to the leftmost Variable in each step.

Right Derivation Tree
A Right Derivation Tree is obtained by applying production to the rightmost Variable in each step.

Eg. For generating the string aabaa from the grammar S- aAS|aSS|ϵ
A-SbA|ba



aabaa



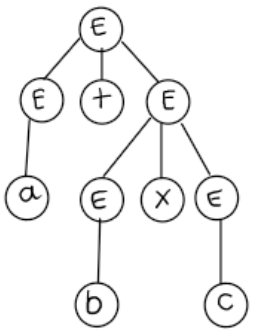
aabaa

Left Derivation Tree
Derivation will be done from left applying production to the rightmost rule

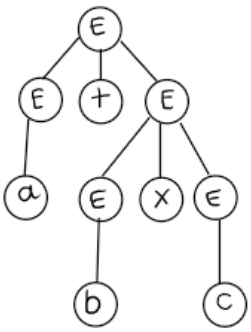
Right Derivation Tree
Derivation Tree will be done from right most non terminal in production rule

String = a+b+c

E
↓
E + E
↓
a + E
↓
a + E × E
↓
a + b × E
↓
a + b × c

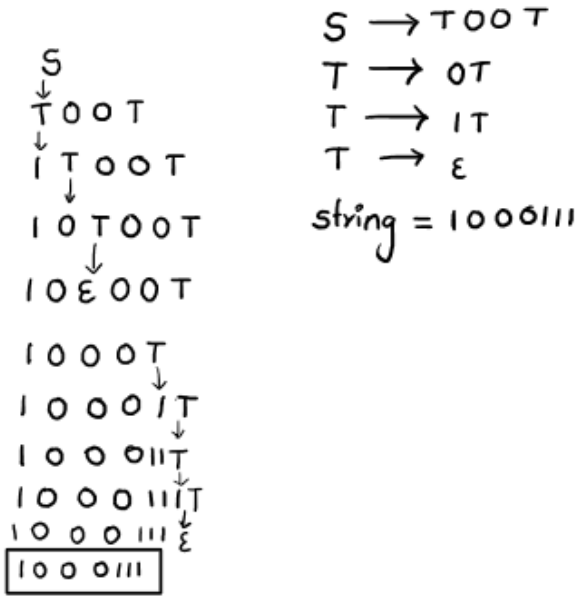


Ex :-
E → E + E
E → E × E
E → a/b/c

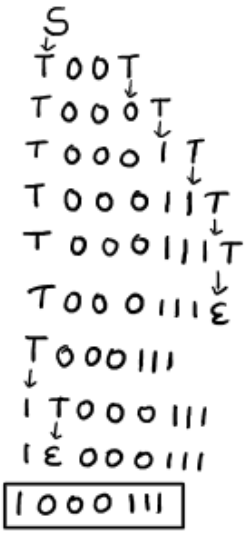


E
↓
E + E
↓
E + E × E
↓
E + E × C
↓
E + b × C
↓
a + b × c

Left most derivation



Right most Derivation



3.5 Ambiguous Grammar

Q: What is Ambiguous Grammar? Illustrate with example.

An ambiguous grammar is a type of context-free grammar (CFG) in which there exists more than one valid parse tree for at least one string in the language generated by the grammar. Ambiguity arises when the production rules of the grammar can lead to multiple interpretations or derivations for the same input string.

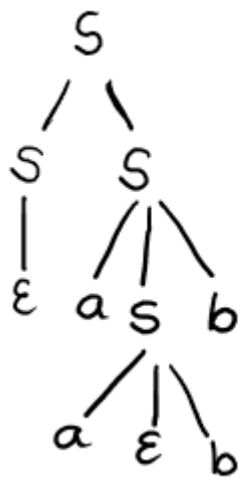
Let's illustrate an ambiguous grammar with an example:

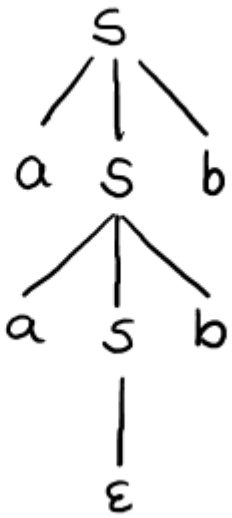
Consider the following grammar:

S = aSb | SS

S = ∈

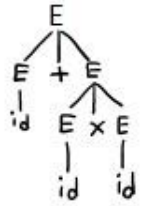
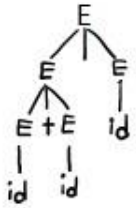
For the string aabb, the above grammar generates two parse trees:





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

Context Free Grammar: E E + E | E × E | id id + id × id

Left Most Derivation	Right Most Derivation	Parse Tree Derivation
E = E + E = id + E = id + E × E = id + id × E = id + id × id	E = E + E = E + E × E = E + E × id = E + E × id = id + id × id	
E = E + E = id + E = id + E × E = id + id × E = id + id × id	E = E + E = E + id = E + E × id = E + id × id = id + id × id	

Q: How to Remove ambiguity from context-free grammar?

Removing ambiguity from context-free grammar involves modifying the grammar to ensure that there is only one valid parse tree for any given input string.
Here are some common techniques to remove ambiguity, along with examples:

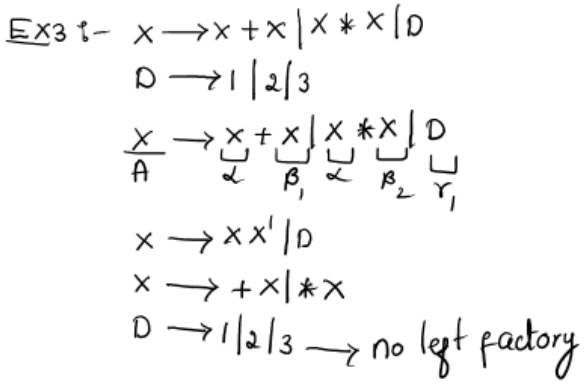
1. Left Factoring:
Combine common prefixes in production rules to make the grammar more deterministic.

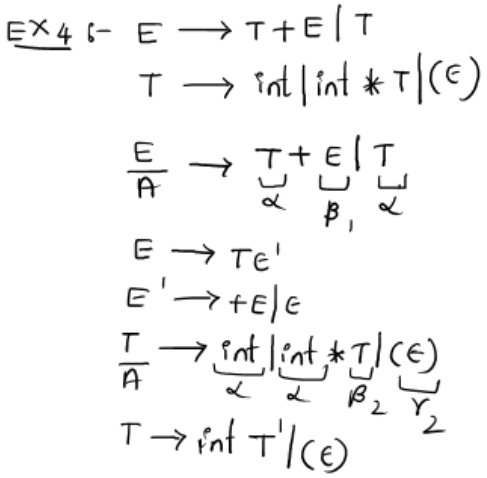
Elimination of left factoring:-

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \gamma_1 \mid \gamma_2 \ldots\ldots$$
$$A \to \alpha A' \mid \gamma_1 \mid \gamma_2$$
$$A \to \beta_1 \mid \beta_2 \mid \ldots\ldots$$

Ex :- $S \to iE+S \mid iEtSeS \mid a$
$E \to b$
$S \to Et\,s \mid \underset{\alpha}{\underline{Et}}\,\underset{\alpha}{\underline{S}}\,\underset{\beta}{\underline{e}}\,\underset{\gamma_1}{\underline{S}} \mid a$
(S: A, Et: α, Et: α, e: β, S: γ_1, a: γ_1)
$S \to iEt\,SS' \mid a$
$S \to \epsilon \mid es$

EX2 :- $A \to aAB \mid aA \mid a$
$B \to bB \mid b$
$A \to \underset{\alpha}{\underline{a}}\,\underset{\beta_1}{\underline{AB}} \mid \underset{\alpha}{\underline{a}}\,\underset{\beta_2}{\underline{A}} \mid \underset{\alpha}{\underline{a}}$
$A \to aA'$
$A' \to AB \mid A \mid \varepsilon$
$\underset{A}{\underline{B}} \to \underset{\alpha}{\underline{b}}\,\underset{\beta_1}{\underline{B}} \mid \underset{\alpha}{\underline{b}}$
$B \to bB'$
$B' \to B \mid \varepsilon$





Imagine you're trying to create a set of instructions for a computer to understand sentences in a made-up language. You've noticed that many sentences start with the same words or symbols. Left factoring is like streamlining your instructions to avoid repeating the same beginning over and over.
Example in Layman's Terms:
Suppose you're describing actions for a robot:
Original Instructions:
"If the robot sees a red ball, it should move forward."
"If the robot sees a blue ball, it should move forward."
Left Factored Instructions:
"If the robot sees a"
"red ball, it should move forward."
"blue ball, it should move forward."
In the original instructions, you repeated the starting phrase ("If the robot sees a") for both red and blue balls. Left factoring is like recognizing the common partsx` and combining them to make the instructions clearer and more concise.

Example 1:
Original Grammar:
Expr -> Expr + Term | Expr * Term | Term
Term -> id

Left Factored Grammar:
Expr -> Term Expr'
Expr' -> + Term Expr' | * Term Expr' | ε
Term -> id

Original Grammar:
Stmt -> if Expr then Stmt else Stmt | while Expr do Stmt | id = Expr ;
Expr -> id + id | id

Left Factored Grammar:
Stmt -> if Expr then Stmt else Stmt | while Expr do Stmt | id Stmt'
Stmt' -> = Expr ; | ε
Expr -> id + id | id

In this example, left factoring is applied to the alternatives of Stmt by recognizing the common prefix (id). Left factoring simplifies the grammar by making it more concise and helps parsers during the parsing process. The key is to identify common prefixes in alternative production rules and factor them out to enhance readability and reduce redundancy

2. Left Recursion Elimination:
Remove left recursion to make the grammar non-left-recursive and, therefore, unambiguous. In the context of context-free grammars (CFGs), left recursion elimination is a technique used to modify a grammar to avoid a situation where a non-terminal refers to itself in a leftmost position in its production rules. This left recursion can lead to ambiguity and difficulties in parsing. The goal of left recursion elimination is to rewrite the grammar in a way that avoids this left recursion.
In the context of context-free grammars (CFGs), left recursion elimination is a technique used to modify a grammar to avoid a situation where a non-terminal refers to itself in a leftmost position in its production rules. This left recursion can lead to ambiguity and difficulties in parsing. The goal of left recursion elimination is to rewrite the grammar in a way that avoids this left recursion.

Layman's Explanation:
Imagine you're giving a set of instructions to someone who is following a sequence of steps. Left recursion would be like giving an instruction that says, "Do this, and then do this again." It's like a loop that keeps going without a clear endpoint. Left recursion elimination is about making the instructions clearer by breaking the loop and providing a more structured set of steps.

Example in Layman's Terms:
Suppose you are describing a process for making a sandwich, and you want to avoid an instruction like, "Make a sandwich, and then make a sandwich again." This could go on forever, and it's not clear when to stop.

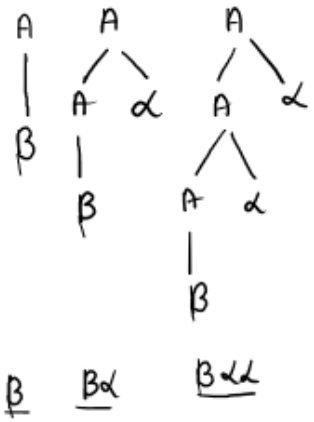
Original Instructions:
Make a sandwich.
Make a sandwich again.
After Left Recursion Elimination:
Make a sandwich.
Finish making sandwiches.
In this example, "Finish making sandwiches" replaces the unclear instruction to make a sandwich again. This is similar to how left recursion elimination works in a CFG.

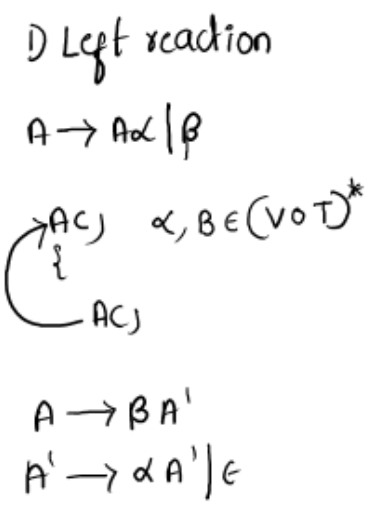
In Grammar Terms:
Consider a CFG with left recursion:
Expr -> Expr + Term | Term
Term -> Term * Factor | Factor
Factor -> id

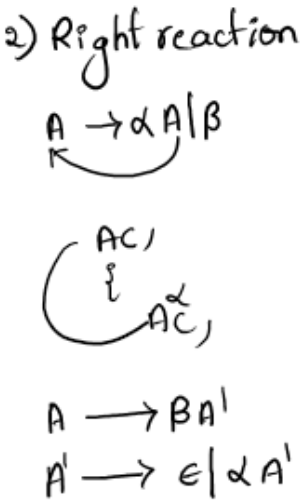
Here, Expr is left-recursive because it refers to itself in a leftmost position. Left recursion elimination transforms this into a non-left-recursive form:
Expr -> Term Expr'
Expr' -> + Term Expr' | ε
Term -> Factor Term'
Term' -> * Factor Term' | ε
Factor -> id
In this modified grammar, Expr' is introduced to handle the continuation of Expr without left recursion. It breaks the loop and provides a clearer structure for parsing.

Left recursion elimination is like improving a set of instructions to avoid an endless loop. It clarifies the steps and helps parsers understand the language better by providing a more structured and non-ambiguous set of rules

Elimination of left reaction







Elimination of left recursion:-

Ex :- Eliminate left recursion in the following groups

Ex 1

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

Ex 2

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

$\frac{E}{A} \rightarrow \frac{E}{A}+\frac{T}{\alpha} \mid \frac{T}{\beta}$ → Left reaction

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$

$\frac{T}{A} \rightarrow \frac{T}{A}*\frac{F}{\alpha} \mid \frac{F}{\beta}$ → left reaction

$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$

Ex 3

$A \rightarrow A\alpha \mid \beta$
↓
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \epsilon$

$F \rightarrow (E) \mid id$ → No left reaction

Elimination of Left recursion

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \beta_1 \mid \beta_2 \mid \ldots\ldots$
$A \rightarrow \beta_1 A' \mid \beta_2 A'$
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \epsilon$

$A \rightarrow A\alpha \mid \beta$
↓
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \epsilon$

Ex 4

$\frac{expr}{A} \rightarrow \frac{expr}{A}+\frac{expr}{\alpha} \mid \frac{expr}{A}*\frac{expr}{\alpha_2} \mid \frac{id}{\beta_1}$

$expr \rightarrow id\ expr'$
$expr \rightarrow +expr\ expr' \mid *expr\ expr' \mid \epsilon$

Ex 5 :- $\frac{S}{A} \rightarrow \frac{SX}{A\alpha_1} \mid \frac{Ssb}{A\alpha_2} \mid \frac{XS}{\beta_1} \mid \frac{w}{\beta_2}$

$S \rightarrow xSS' \mid aS'$
$S' \rightarrow xS' \mid sbA' \mid \epsilon$

Elimination of Left recursion :-

Ex 6 :- $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sa \mid f$

$S \rightarrow Aa \mid b$ → No left recursion
$A \rightarrow Ac \mid Sd \mid f$

So substitute S products in A

$A \rightarrow Ac \mid Sd \mid f$

$A \rightarrow \frac{Ac}{A\alpha_1} \mid \frac{Aad}{A\alpha_2} \mid \frac{bd}{\beta_1} \mid \frac{f}{\beta_2}$

$A \rightarrow bdA' \mid fA'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$
$A \rightarrow \beta_1 A' \mid \beta_2 A'$
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \epsilon$

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid fA'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

Example:
Original Grammar:
Expr -> Expr + Term | Term
Term -> Term * Factor | Factor
Factor -> id

Left-Recursion Eliminated Grammar:
Expr -> Term Expr'
Expr' -> + Term Expr' | ε
Term -> Factor Term'
Term' -> * Factor Term' | ε
Factor -> id

3. Operator Precedence and Associativity:
Define clear rules for operator precedence and associativity in the 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 :

- The level at which the production is present denotes the priority of the operator used.

- The production at higher levels will have operators with less priority. In the parse tree, the nodes that are at top levels or close to the root node will contain the lower priority operators.
- The production at lower levels will have operators with higher priority. In the parse tree, the nodes that 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 productions. The parse tree will also be right recursive and grow on the right side.
  ^ is a right associative operator

Example:
Original Grammar:
Expr -> Expr + Expr | Expr * Expr | id

Modified Grammar:
Expr -> Expr + Term | Term
Term -> Term * Factor | Factor
Factor -> id

Parentheses or Brackets for Grouping:
Use explicit parentheses or brackets to clarify the grouping of expressions.
Example:
Original Grammar:
Expr -> Expr + Term | Term * Expr | id

Modified Grammar:
Expr -> Expr + Term | (Expr * Term) | id
Term -> id

Associativity Rules:
Explicitly specify associativity to resolve conflicts.
Example:
Original Grammar:
Expr -> Expr + Expr | id
Modified Grammar:
Expr -> Expr + Expr | Expr - Expr | id
Introduce New Non-terminals:
Introduce new non-terminals to disambiguate certain productions.

Example:

Original Grammar:
Expr -> Expr + Term | Term
Term -> id

Modified Grammar:
Expr -> Term Expr'
Expr' -> + Term Expr' | ε
Term -> id
Each of these techniques is applicable depending on the specific structure and requirements of the grammar. The choice of technique depends on the nature of the ambiguity present in the grammar and the desired parsing behavior.

Q: what is ambiguity in grammar? Eliminate ambiguity from balanced parenthesis grammar.
Ambiguity in grammar occurs when a particular string in the language can be derived by the grammar in more than one way. In other words, there are multiple parse trees for the same input string. Ambiguity can lead to confusion in the parsing and interpretation of the language.

Let's consider a simple context-free grammar for generating balanced parentheses:

Set of terminal symbols (Σ): {'(', ')'}
Set of non-terminal symbols (V): {S}
Start symbol (S): S
Production rules (P):
S -> (S) | ε
This grammar generates strings of balanced parentheses, where each opening parenthesis '(' is matched with a closing parenthesis ')'.

Now, let's see an example of an ambiguous string and how to eliminate ambiguity:

Ambiguous string: "()()"

This string can be derived by the grammar in two different ways:

S -> (S) -> ()(S) -> ()()
S -> (S) -> (S) -> ()()
To eliminate ambiguity, we can modify the grammar by introducing a distinction between the left and right parentheses. Here's an unambiguous grammar for balanced parentheses:

Set of terminal symbols (Σ): {'(', ')'}
Set of non-terminal symbols (V): {S, L, R}
Start symbol (S): S
Production rules (P):

S -> LR | ε
L -> (S
R ->)S
This grammar ensures that a corresponding right parenthesis always follows the left parenthesis. Now, the derivation for the string "()()" is unique:

S -> LR -> (S)R -> ()S -> ()LR -> ()()

Q: Simplify the grammar by removing productive and unreachable symbols.

S->AB | AC

A->aA b| ε

B->bA

C->bCa

D->AB

Identify Unproductive Symbols:

A non-terminal is unproductive if it cannot generate any terminal string.
Start by marking all non-terminals as unproductive.
Identify Unreachable Symbols:

A non-terminal is unreachable if it cannot be reached from the start symbol.
Start by marking all non-terminals as unreachable.
Remove Unproductive and Unreachable Symbols:

Remove all productions involving unproductive or unreachable symbols.
Let's apply these steps to the given grammar:

Mark all non-terminals as unproductive and unreachable initially:

Unproductive: {S, A, B, C, D}
Unreachable: {S, A, B, C, D}
Now, let's go through each production rule and update our sets:

S -> AB | AC: Both A and B are unproductive, so S is unproductive.
A -> aA b | ε: A can generate terminal strings, so it is productive.
B -> bA: B can generate terminal strings, so it is productive.
C -> bCa: C can generate terminal strings, so it is productive.
D -> AB: Both A and B are unproductive, so D is unproductive.

Unproductive: {S, D}
Unreachable: {S, A, B, C, D}
Since we've identified unproductive symbols (S and D), and we've marked all symbols as unreachable initially, we can remove the productions involving unproductive or unreachable symbols. After removing those productions, the simplified grammar becomes:

A -> aA b | ε
B -> bA
C -> bCa
This simplified grammar only includes the productive and reachable symbols and their corresponding production rules.

2 Syntax Analysis Phase of Compiler

The syntax analysis phase, also known as parsing, is a crucial stage in the compilation process of a programming language. Its primary goal is to analyze the source code and determine its syntactic structure according to the rules of the language's grammar. This phase ensures that the source code adheres to the specified syntax of the programming language. The output of this phase is typically a hierarchical structure called **a parse tree or abstract syntax tree (AST),** which represents the syntactic structure of the program.

Here are the key steps and concepts involved in the syntax analysis phase of a compiler:

Lexer (Lexical Analysis): The process begins with lexical analysis, performed by a lexer or lexical analyzer.The lexer breaks down the source code into tokens, which are the smallest units of meaning in a programming language (e.g., keywords, identifiers, operators, and literals).Each token is associated with a token type that represents its category.

Grammar: The compiler relies on a formal grammar to define the syntactic rules of the programming language. The grammar is typically specified using Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).

It defines the structure of valid programs in the language.

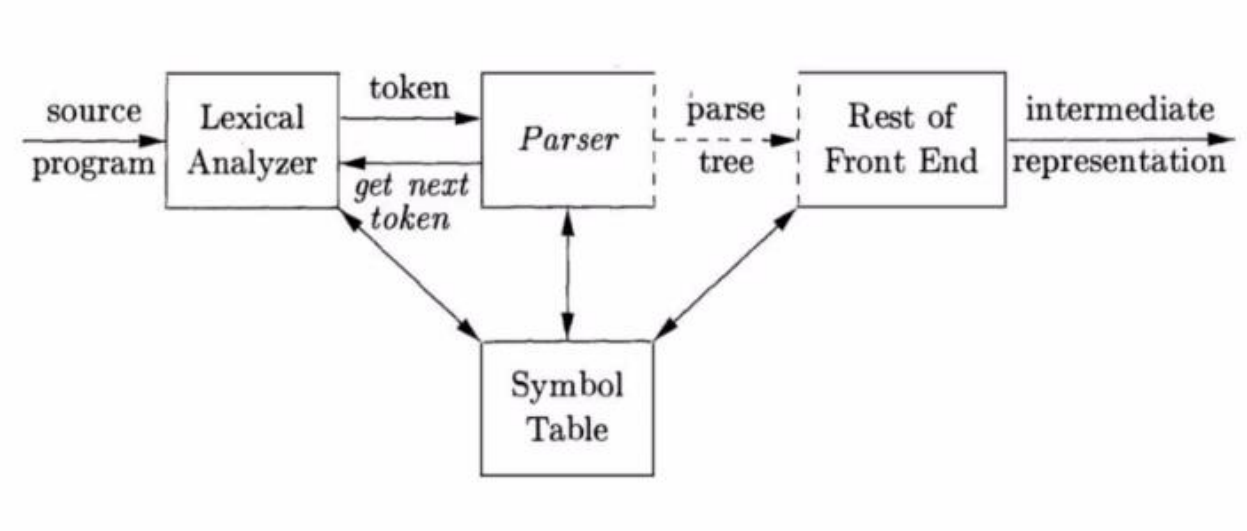
Parser (Syntax Analyzer): The parser takes the stream of tokens generated by the laxer and constructs a parse tree or AST based on the grammar rules. It checks whether the input adheres to the syntactic rules specified by the grammar. If the input is syntactically correct, the parser produces a parse tree or AST; otherwise, it generates syntax error messages.

Parse Tree or Abstract Syntax Tree (AST): The parse tree is a hierarchical tree structure that represents the syntactic structure of the program. The nodes of the tree correspond to grammar rules, and the leaves correspond to the actual tokens from the source code. AST is a more refined version of the parse tree that abstracts away some of the details, focusing on the essential structure of the program.

Semantic Analysis (optional): While syntax analysis focuses on the program's structure, semantic analysis checks for meaningful relationships and ensures that the program's behavior adheres to the language's semantics. This phase may involve type checking, scope resolution, and other checks that go beyond syntax. The output of the syntax analysis phase, typically the AST, serves as input for subsequent compilation phases, such as semantic analysis, optimization, and code generation. The successful completion of the syntax analysis phase indicates that the source code is grammatically correct, allowing the compiler to proceed with further processing.

2.1 Role of Parser

The parser, also known as the syntax analyzer, plays a crucial role in the compilation process of a programming language. Its primary responsibility is to analyze the syntactic structure of the source code based on the rules defined by the language's grammar.



Here are the key roles and functions of the parser in the compiler:

Tokenization: The parser receives a stream of tokens from the lexer (lexical analyzer), which breaks down the source code into meaningful units (tokens). Tokens represent fundamental language elements such as keywords, identifiers, operators, and literals.

Syntactic Analysis: The parser verifies whether the sequence of tokens adheres to the syntactic rules defined by the formal grammar of the programming language. It checks the correctness of the program's structure and identifies the relationships between different language constructs.

Parse Tree or Abstract Syntax Tree (AST) Generation: The parser constructs a hierarchical representation of the syntactic structure, known as a parse tree or abstract syntax tree (AST).The parse tree/AST captures the hierarchical relationships between different language constructs and provides a structured representation of the program.

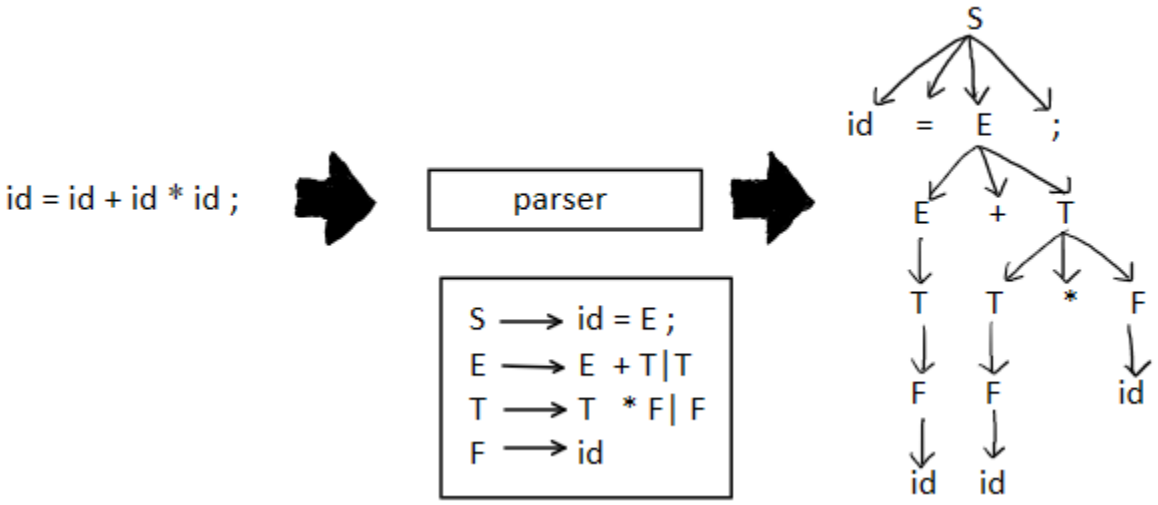
Error Handling: If the parser encounters syntax errors in the source code, it generates appropriate error messages. Syntax errors may include misplaced tokens, missing or extra symbols, or violations of the grammar rules.

Ambiguity Resolution: In cases where the grammar is ambiguous (allowing multiple valid interpretations), the parser may implement specific rules or heuristics to resolve ambiguities and choose a unique interpretation.

Intermediate Representation: The parse tree or AST serves as an intermediate representation of the program's structure.This intermediate representation is used in subsequent phases of the compiler, such as semantic analysis, optimization, and code generation.

Integration with Semantic Analysis: The parser may be closely integrated with the semantic analysis phase, where it helps identify and analyze semantic constructs in addition to the syntactic structure. Semantic analysis involves checking for meaning and ensuring that the program adheres to the language's semantics.

Communication with Code Generator: The output of the parser, typically the parse tree or AST, is passed on to the subsequent phases of the compiler, especially the code generation phase. The information provided by the parse tree or AST guides the code generator in producing an executable representation of the program.



In summary, the parser is a fundamental component of the compiler that ensures the syntactic correctness of the source code, facilitates the generation of intermediate representations, and plays a crucial role in the overall compilation process. It acts as a bridge between the lexical analysis and subsequent phases, providing a structured representation of the program for further processing.

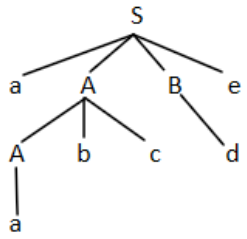
2.2 Types of Parsers

The process of deriving the string from the given grammar is known as derivation(parsing). Based on the derivation process and it is classified in two ways

a. Top-down parser
b. Bottom-up Parser

Gereration of parse Tree: s⟶ aABe, A⟶ Abc | a, B⟶ d aabcde

Top down approach:

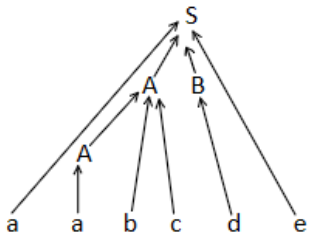


S ⇒ aABe
⇒ aAbcBe
⇒ aabcBe
⇒ aabcde

Decision: which production to use.

(left most derivation)

Bottom up approach:

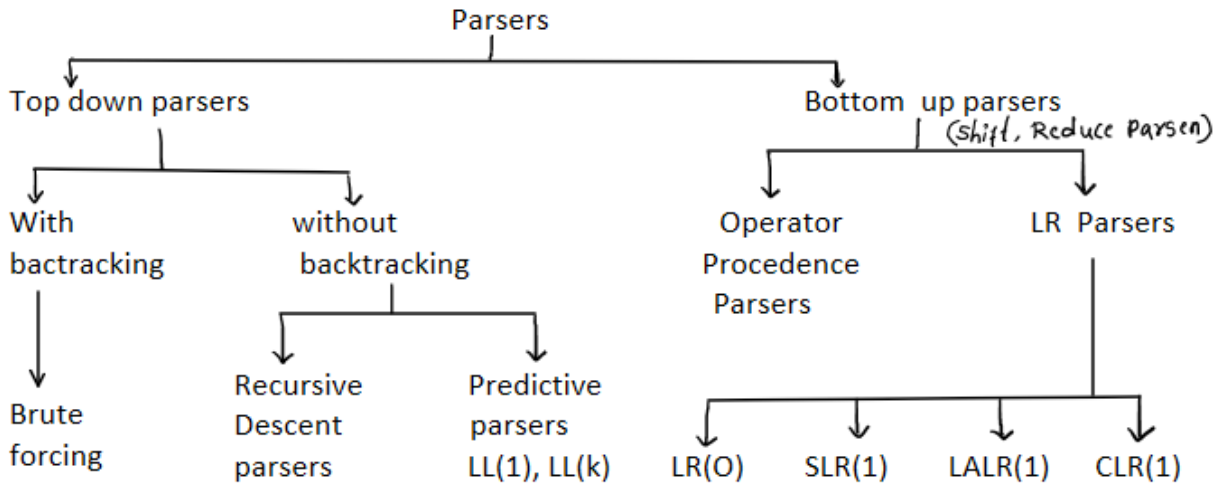


S ⇒ aABe
⇒ aAde
⇒ aAbcde
⇒ aabcde

Decision: when to reduce

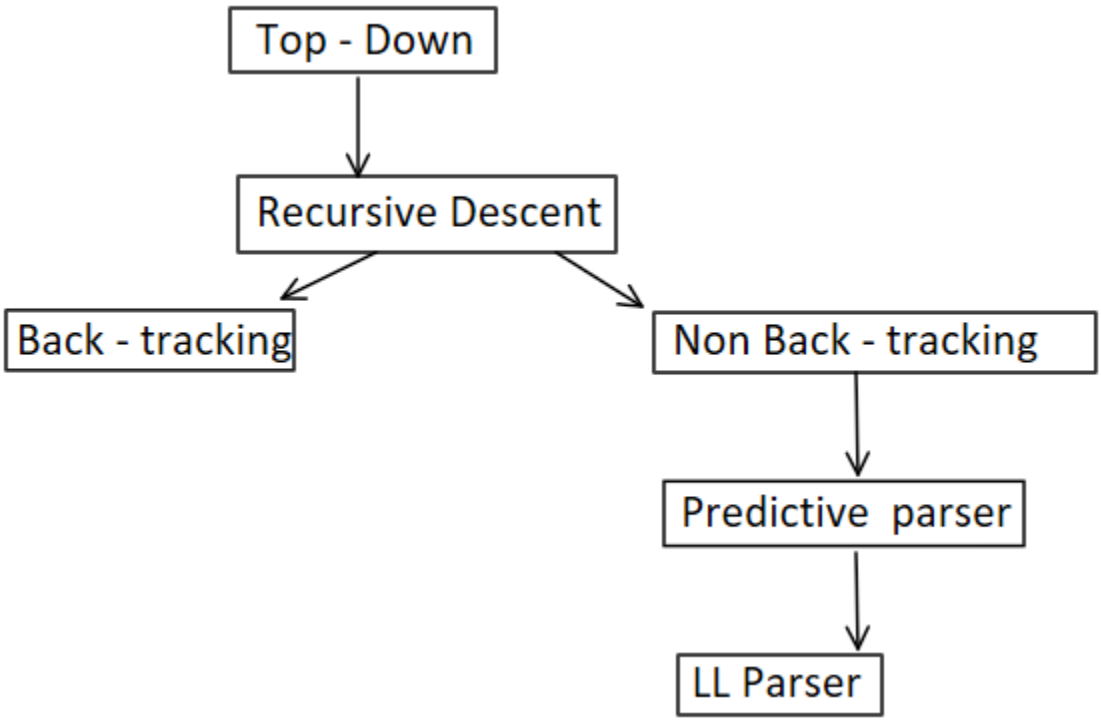
(Right most derivation) . In Reverse

Classification of parsers:



2.2.1 Top-down Parsing

Top-down parsing is a parsing technique used in the syntax analysis phase of a compiler. In top-down parsing, the process starts from the top of the parse tree (root) and moves down towards the leaves. The goal is to recognize the input source code by applying production rules of the formal grammar specified for the programming language. The primary approach in top-down parsing is to start with the highest-level construct (the start symbol of the grammar) and recursively expand it until the entire parse tree is built.



Here are the key steps and characteristics of top-down parsing:

Start Symbol: The parsing process begins with the start symbol of the formal grammar. The start symbol represents the highest-level construct in the language.

Grammar Rules: The parser applies production rules from the grammar to expand the start symbol into a sequence of symbols. These production rules define how different language constructs can be derived.

Predictive Parsing: Top-down parsing is often associated with predictive parsing, where the parser uses a look-ahead symbol to predict which production rule to apply without backtracking. Predictive parsing can be implemented using a table-driven approach, such as a predictive parsing table or a set of recursive descent procedures.

Recursive Descent Parsing: Recursive descent parsing is a common technique for implementing top-down parsers. In this approach, each non-terminal in the grammar is associated with a recursive procedure or function that corresponds to the production rules for that non-terminal.

The parser calls these procedures recursively to expand non-terminals into terminals and other non-terminals until a complete parse tree is constructed.

Backtracking: In case of ambiguity or if a chosen production rule does not match the input, top-down parsers may use backtracking to explore alternative choices. Backtracking involves undoing a decision and trying a different path in the parse tree.

Leftmost Derivation: Top-down parsing typically aims for leftmost derivations, meaning it selects the leftmost non-terminal to expand at each step. Leftmost derivations are useful for constructing parse trees that resemble the structure of the source code.

LL Parsing: Top-down parsing is often referred to as LL parsing, where "L" stands for left-to-right scan of the input and "L" stands for constructing a leftmost derivation.The number following "LL" represents the number of look-ahead symbols used for prediction (e.g., LL(1) uses one look-ahead symbol).

Top-down parsing has some advantages, such as simplicity and ease of implementation, especially when the grammar is LL(1) (no left recursion and no common prefixes in the production rules). However, it may encounter challenges with certain types of grammars, such as left-recursive or ambiguous grammars, which may require additional techniques or modifications.

2.2.1.1 Recursive-Descent Parsing

A recursive descent parser is a top-down parsing technique where the parsing process starts from the top of the grammar tree and moves down to the leaves. It uses a set of recursive procedures or functions to recognize the different non-terminals in the grammar. Each non-terminal in the grammar is associated with a parsing function, and the parser recursively calls these functions to recognize the structure of the input according to the production rules of the grammar.

The typical procedure for handling a non-terminal in a top-down parser involves creating a parsing function for that non-terminal. The parsing function is responsible for recognizing the syntax defined by the production rules associated with the non-terminal.

- **Algorithm**

```
void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
}
```

•A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.
•Execution begins with the procedure for the start symbol,
Recursive-descent may require backtracking; that is, it may require repeated scans over the input (backtracking is not very efficient)

Recursive decent Parser:-

1) E → i E

   E → i E | E i + i

Solution:-

```
E( )
{
      if ( input = = 'i' )
            input ++,

       EPRIME( ),
}
```

```
EPRIME( )
{
     if ( input = = '+')
       {
            input ++;
            if (input == 'i')

            input ++,
            EPRIME( ),
       }
     else
       return;
}
```

1. $E \rightarrow iE'$

   $E' \rightarrow iE' \mid \epsilon$

2. $E \rightarrow TE'$

   $E' \rightarrow +TE' \mid \epsilon$

   $T \rightarrow FT'$

   $T \rightarrow *FT' \mid \epsilon$

   $F \rightarrow (\epsilon) \mid id$

2)

$E \longrightarrow T\ E'$

$E \longrightarrow +\ T\ E' \mid E$

$T \longrightarrow F\ T'$

$T' \longrightarrow *\ F\ T' \mid E$

$F \longrightarrow (\ E\ ) \mid id$

id + id $

```
E( )
  {
      TC
      EPRIME( ),
  }
```

```
   EPRIME( )
{
      if ( input = = '+')
    {
         input ++,
          T( ),
         EPRIME( ),
     }
    else
      return;
}
  T( )

 {
    F( );
    TPRIME( ),
 }
```

```
  EPRIME( )
{
     if ( input = = '*')
    {
        input ++,
          T( ),
        EPRIME( ),
     }
     else
     return
}
    if ( input = = '( ')
   {
     input ++,
     E ( )
     if ( input = = ' )')
         input ++,
    }
     else if ( input = = ' id')
          input ++,
```

Example :

Expr -> Term + Expr | Term

Term -> num

Typical Procedure :

```
def parse_Expr():
    parse_Term()
    if current_token == '+':
        consume('+')
        parse_Expr()
```

```
def parse_Term():
    if current_token.isdigit():
        consume(current_token)
    else:
        raise Exception("Expected a number.")
```

2.2.1.2 FIRST AND FOLLOW

Rules for Computing First Sets:

step 1: if A-> aα , α $\in (V \cup T)^*$

FIRST(A) = { a }

Step 2 : if A -> ε Then FIRST(A) = { ε}

Step 3: if A-> BC then

FIRST(A) = FIRST(B) if FIRST(B) does not contain ε

If FIRST(B) contains FIRST(A) = FIRST(B) U FIRST(C)

Rules for Computing Follow Sets:

Step 1: 'S' then FOLLOW(S) = { $}

Step 2: if A -> α B β then FOLLOW(B) = FIRST(β) if FIRST(B) doesnot contain ε

Step 3: if A -> α B then FOLLOW(B)=FOLLOW(A)

Step 4: if A -> α B β where β-> ε then FOLLOW(B) = FOLLOW(A)

Example:

Consider the grammar:

1. S → Aa | b

2. A → Ac | ε

First Sets:

First(S) = {b, a}

First(A) = {b, a, ε}

Follow Sets:

Follow(S) = {$}

Follow(A) = {a, $}

These rules are applied iteratively until the sets no longer change. The computed First and Follow sets are essential for constructing LL(1) parsing tables and are used in top-down parsing techniques like predictive parsing.

EX 1 :-
$E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \epsilon$
$F \rightarrow (E) | id$

Solution :-

First (E) = {(, id}
First (E') = {+, ε}
First (T) = {(, id}
First (T') = {*, ε}
First (F) = {(, id}
Fallow (E) = {$,)}
Fallow (E') = {$,)}
Fallow (T) = {+, $,)}
Fallow (T') = {+, $,)}
(F) = {*, +, $,)}

EX 2 :-
$S \rightarrow ABCDE$
$A \rightarrow a | \epsilon$
$B \rightarrow b | \epsilon$
$C \rightarrow c$
$D \rightarrow d | \epsilon$
$E \rightarrow e | \epsilon$

$S \rightarrow ABCDE$
$S \rightarrow ABCD$

First(S) = {a, b, c}
First (A) = {a, ε}
First (B) = {b, ε}
First (C) = {c}
First (D) = {d, ε}
First (E) = {e, t}
Fallow (S) = {$}
Fallow (A) = {b, c}
Fallow (B) = {c}
Fallow (C) = {d, e, $}
Fallow (D) = {e, $}
Fallow (E) = {$}

Ex 3 :- $S \rightarrow Bb \mid cd$
$B \rightarrow aB \mid \epsilon$
$C \rightarrow cC \mid \epsilon$

First (S) = {a, b, c, d}
First (B) = {a, ε}
First (C) = {c, ε}
Follow (S) = {$}
Follow (B) = {b}
Follow (C) = {d}

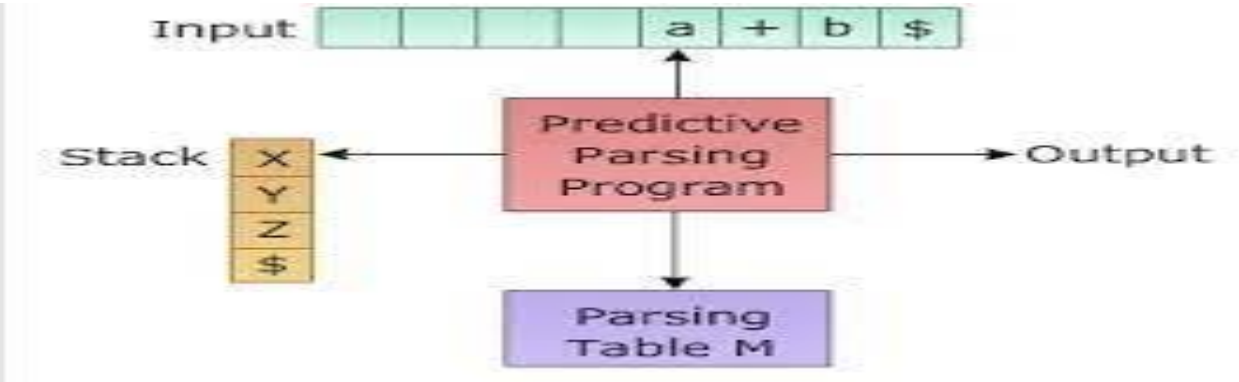
Ex 4 :- $S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid Bc$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

First (S) = {d, g, h, b, a}
First (A) = {d, g, h, ε}
First (B) = {g, ε}
First (C) = {h, ε}
Follow (S) = {$}
Follow (A) = {h, g, $}
Follow (B) = {$, a, h, g}
Follow (C) = {g, $, b, h}

Ex 5 :- $S \rightarrow aABb$
$A \rightarrow c \mid \epsilon$
$B \rightarrow d \mid \epsilon$

First (S) = {a}
First (A) = {c, ε}
First (B) = {d, ε}
Follow (S) = {$}
Follow (A) = {d, b}
Follow (B) = {b}

2.2.1.3 LL(1) PARSER / PREDICTIVE PARSER /NON-RECURSIVE PREDICTIVE PARSING



Construction of LL1(Parser) :

1. Elimination of Left recursion
2. Elimination of Left Factoring
3. Calculation of FIRST and FOLLOW
4. Construction of parsing table
5. Check whether string is accepted by parser or not

Construction of LL1(Parser) :

1. Elimination of Left recursion
2. Elimination of Left Factoring
3. Calculation of FIRST and FOLLOW
4. Construction of parsing table
5. Check whether string is accepted by parser or not

Construction of Parse Table :

A -> α

Step 1. A -> α

FIRST(α) -> a

Add A -> α to M[A,a]

Step 2. FIRST(α) contains ε or A-> ε

FOLLOW(A) ->b

Then ADD A -> α to M[A,b]

Example :- $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Solution:-

Step 1 :- Elimination of left recursion

$A \rightarrow A\alpha \mid \beta$

Replace with

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Similarly

(a) $E \rightarrow E + T \mid T$

$A \quad A \quad \alpha \quad \beta$

Replace with

$E \rightarrow TE'$

$E' \rightarrow +T \mid \epsilon$

(b) $T \rightarrow T * F \mid F$

$A \quad A \quad \alpha \quad \beta$

Replace with

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \epsilon$

(c) $F \rightarrow (E) | id$ → No left recursion

After Eliminating left recursion is

$E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \epsilon$
$F \rightarrow (E) | id$

Step 2 :- Eliminated left Factors

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \gamma_1 | \gamma_2$

Replace with

$A \rightarrow \alpha A' | \gamma_1 | \gamma_2$
$A' \rightarrow \beta_1 | \beta_2$

No left factor (No common left factor)

Step 3 :- calculate First and Fallow

First(E) → {c, id}
First(E') → {+, ε}
First(T) → {c, id}
First(T') → {*, ε}
First(F) → {c, id}
Fallow(E) → {$,)}
Fallow(E') → {$,)}
Fallow(T) → {+, $,)}
Fallow(T') → {+, $,)}
Fallow(F) → {*, +, $,)}

Step 4 :- Construct the Parse table.

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

Production (1)

$E \to TE'$

$First(TE') = \{(, id\}$

Add $E \to TE'$ to $M[E, (]$

$M[E, id]$

Production (2)

$E \to TE'$

$E' \to FTE' | \epsilon$

$T \to FT'$

$T' \to *FT' | \epsilon$

$F \to (E) | id$

Production (3)

$T \to FT'$

$First(FT') = \{(, id\}$

Add $T \to FT'$ to $M(T, ()$

$M(T, id)$

Production ④

a) $\frac{T'}{A} \rightarrow \frac{*FT'}{\alpha}$

Follow $(*FT') = \{*\}$

Add $T' \rightarrow *FT'$ to $M[T', *]$

b) $\frac{T'}{A} \rightarrow \frac{E}{\alpha}$

Follow $(\epsilon) = \{+, \$,)\}$

Production ⑤

a) $\frac{F}{A} \rightarrow \frac{(E)}{\alpha}$

First $(CE)) = \{c\}$

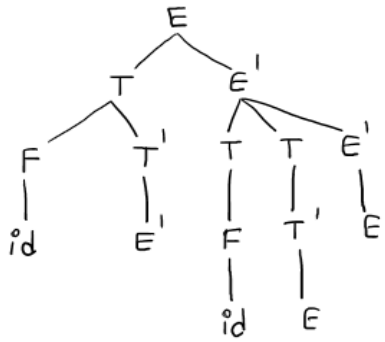
b) $F \rightarrow id$

Evaluation :- id + id $

Stack	Input string	Action
$E	id + id $	$T \rightarrow TE'$
$E'T	id + id $	$T \rightarrow FT'$
$E'T'F	id + id $	$F \rightarrow id$
$E'T'id	id + id $	pop
$E'T'	+ id $	$T' \rightarrow \epsilon$
$E'	+ id $	$E' \rightarrow +TE'$
$E'T+	+ id $	pop
$E'T	id $	$T \rightarrow FT'$
$E'T'F	id $	$F \rightarrow id$
$E'T'id	id $	pop
$E'T'	$	$T' \rightarrow \epsilon$
$E'T'	$	$E' \rightarrow \epsilon$
$	$	

It can recognize the input string

So



id E + id E E
→ id + id

Ex 1 :- construct predictive parser

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Step 1 :- Elimination of left recursion

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$S \rightarrow (L) \mid a \rightarrow$ No left recursion

$\underset{A}{L} \rightarrow \underset{A}{L}, \underset{\alpha}{S} \mid \underset{\beta}{S}$

Replace with

$L \rightarrow SL'$

$L' \rightarrow SL' \mid \varepsilon$

After Eliminating Left recursion is

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L \rightarrow , SL' \mid \varepsilon$

Step 2 :- Elimination of Left factors
No left factor

Step 3 :- calculate First and Fallow

First(S) = {(, a}

First(L) = {(, a}

First(L') = {, , ε}

Fallow(S) = {, ,), $}

Fallow(L) = {) }

Fallow(L') = {) }

Step 4 :- construct of parser table

	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→E		L'→,SL'	

1. (a) $\frac{S}{A} \rightarrow \frac{(L)}{\alpha}$

First ((L,) = {(}

Add S→(L) to M[S,(]

$\frac{S}{A} \rightarrow \frac{a}{\alpha}$

Fallow (a) = {a}

Add S→a to m[S,a]

2. L→SL'

First(SL') = {(,a}

Add L→SL' to m[L,(]

m[L,a]

3. a) $\frac{L'}{A} \rightarrow \frac{SL'}{\alpha}$

First (,SL') = {,}

b) L'→E

Fallow(L') = {)}

Stock	input string	Action
$S	(a)$	S → (L)
$)L(	(a)$	pop
$)L	a)$	L → SL'
$)L'S	a)$	S → a
$)L'a	a)$	pop
$)L'	)$	L' → ε
$)	)$	pop
$	$	Accepted

Another

Example:- construct predictable parser for one following check wheather the string uaabb is Accepted or not

S → aAB | bA | ε

A → aAb | ε

B → bB | ε

First (S) = {a, b, ε}
First (A) = {a, ε}
First (B) = {b, ε}
Fallow(S) = {$}
Fallow (A) = {$, b}
Fallow(B) = {$}

Parsing table:-

	a	b	$
S	S→aAB	S→bA	S→
A	A→aAb	A→ε	A→
B		B→bB	B→

1. a) S→aAB
   First(aAB) = {a}
   S→aAB to M[S→a]
   b) S→bA
   First(bA) = {b}
   S→bA to m[S,b]
   c) S→ε
   Fallow(S) = {$}
   Add S→ε to m[S,$]

2. a) A→aAb
   First(aAb) = {a}
   b) A→ε
   Fallow(A) = {$,b}

3. a) B→bB
   First(bB) = {b}
   b) B→ε
   Fallow(B) = {$}

Another example

Stack	Input string	Action
$S	aabb$	S→aAB
$BAa	aabb$	pop
$BA	abb$	A→aAb
$BbAa	abb$	pop
$BbA	bb$	A→ε
$Bb	bb$	pop
$B	b$	B→bB
$Bb	b$	pop
$B	$	B→ε
$ε	$	
$	$	

Another example :-

$S \rightarrow iEES \mid iEESes \mid a$

$E \rightarrow b$

step 1 :- Elimination of left recursion

No left recursion

Step 2 :- Elimination of Left factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma_1 \mid \gamma_2$

Replace with

$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2$

$A' \rightarrow \beta_1 \mid \beta_2$

$S \rightarrow \underbrace{iEES}_{\alpha}\underbrace{e}_{\beta_1} \mid \underbrace{iEtS}_{\alpha}\underbrace{eS}_{\beta_2} \mid \underbrace{a}_{\gamma}$

Replace with

$S \rightarrow iEESS' \mid a$

$S' \rightarrow E \mid es$

$E \rightarrow b$

Step 3 :- calculate First and Fallow

First (S) = {i, a}

First (S') = {ε, e}

First (E) = {b}

Fallow (S) = {$, e}

Fallow (S') = {$, e}

Fallow (E) = {t}

$S \rightarrow iEESS' \mid a$

$S' \rightarrow E \mid es$

$E \rightarrow b$

Step 4 :- Predictive parse table

	i	t	a	e	b	$
S	S→iEtSS'		S→a			
S'				S→eS S→ε		S'→ε
E					E→b	

1. a) $\frac{S}{A} \; \frac{iEtSS'}{\alpha}$ to m[S, i]

First (iEtSS') = {i}

b) $\frac{S}{A} \rightarrow \frac{a}{\alpha}$

First (a) = {a}

2. a) $\frac{S'}{A} \longrightarrow \frac{\epsilon}{\alpha}$

Fallow (S') = {$, e}

b) S' $\longrightarrow$ eS

First (eS) = {e}

3. $\epsilon \longrightarrow b$

First (b) = {b}