Introduction to CFG (Context Free Grammar)

❖ Table of Contents:

1. Introduction to CFG

A context Free Grammar (CFG) is a 4-tuple such that-

G = (V, T, P, S)

where-

V = Finite non-empty set of variables / non-terminal symbols

T = Finite set of terminal symbols

P = Finite non-empty set of production rules

S = Start symbol

Example-01: Find out the characteristics of CFG from the following expression: -

Expression → Expression + Term | Term Term → Term * Factor | Factor Factor → (Expression) | id

Variables V = {Expression, Term, Factor}

Terminals T = {+, *, (,), id}

Production rules P = {

Expression → Expression + Term | Term

Term → Term * Factor | Factor

Factor → (Expression) | id

}

Start Symbol S = {Expression}

Example-02: Write down the formal definition of CFG for the following expression: -

S → aSb / ∈

Variables V = {S}

Terminals T = {a, b, ∈}

Production rules P = S → aSb / ∈

Start Symbol S = {S}

Introduction to CFG (Context Free Grammar)

Example-03: Write down the formal definition of CFG for the following expression: -

S → aAb / ∈
A → aAb / ∈

Consider a grammar G = (V, T, P, S) where-
Variables V = {S}
Terminals T = { a , b }
Production rules P = { S → aSbS , S → bSaS , S → ∈ }
Start Symbol S = { S }

Example-04: Find out the characteristics of CFG from the following expression: -

S → SS
S → (S)
S → ∈

Consider a grammar G = (V , T , P , S) where-
V = { S }
T = { (,) }
P = { S → SS , S → (S) , S → ∈ }
S = { S }

Example 5 : Find out the characteristics of CFG from the following expression: -

S → 0S | 1S
S → ε

Consider a grammar G = (V , T , P , S) where-
V = { S }
T = { 0, 1}
P = { S → 0S, S → 1S , S → ε }
S = { S }

Example 6 : Find out the characteristics of CFG from the following expression: -

S → ABa ,
A → BB ,
B → ab ,
AA → b

Consider a grammar G = (V , T , P , S) where-
V = { S , A , B } // Set of Non-Terminal symbols
T = { a , b } // Set of Terminal symbols
P = { S → ABa , A → BB , B → ab , AA → b } // Set of production rules
S = { S } // Start symbol

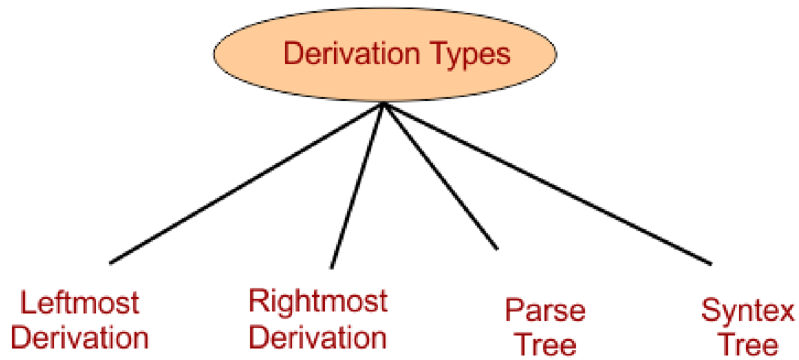
Introduction to CFG (Context Free Grammar)

2. Derivations of CFG

The process of deriving a string from a given grammar is called derivation. The Representation of derivation in the form of a tree is called a derivation tree.

Types of Derivation

There are four basic types of derivations in the theory of Automata:



1. Leftmost Derivation in Automata

The process of deriving a string by expanding the leftmost non-terminal at each step is called as leftmost derivation.

- ✓ We read the given input string (W) from left to right in the leftmost derivation tree.
- ✓ The representation of leftmost derivation in a tree is called a leftmost derivation tree.

2. Rightmost Derivation-

The process of deriving a string by expanding the rightmost non-terminal at each step is called as rightmost derivation.

- ✓ We read the given input string (W) from right to left in the rightmost derivation tree.
- ✓ The representation of rightmost derivation in a tree is called a rightmost derivation tree.

3. Parse Tree in Automata

The process of deriving a string from a given grammar is called derivation. The geometrical representation of a derivation is known as a derivation tree or parse tree.

Properties Of Parse Tree-

- ✓ Root node of a parse tree is the start symbol of the grammar.
- ✓ Each leaf node of a parse tree represents a terminal symbol.
- ✓ Each interior node of a parse tree represents a non-terminal symbol.
- ✓ Parse tree is independent of the order in which the productions are used during derivations.

4. Syntax Tree in Automata

When constructing a parse tree in Automata, it may contain unnecessary details. So, it is very difficult for a compiler to execute unnecessary information. That's why a syntax tree, which holds just the necessary information, is used. Syntax trees are abstract or compact representation of parse trees. They are also called as **Abstract Syntax Trees**.

Introduction to CFG (Context Free Grammar)

❑ Practice Problems Based on Derivations and Parse Tree-

Problem-01:

Consider the grammar-

S → bB / aA

A → b / bS / aAA

B → a / aS / bBB

For the string w = bbaababa, find-

- Leftmost derivation
- Rightmost derivation
- Parse Tree
- Syntax tree

Solution-

1. Leftmost Derivation-

S → bB

→ bbBB (Using B → bBB)

→ bbaB (Using B → a)

→ bbaaS (Using B → aS)

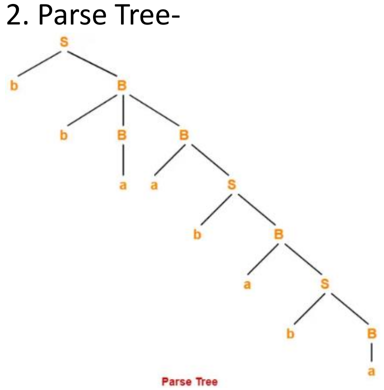
→ bbaabB (Using S → bB)

→ bbaabaS (Using B → aS)

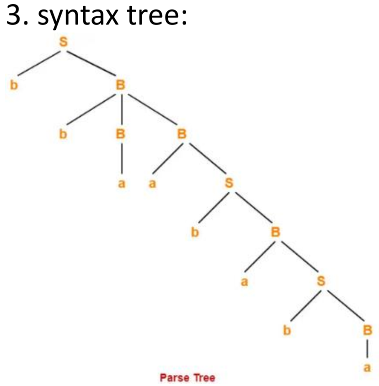
→ bbaababB (Using S → bB)

→ bbaababa (Using B → a)

2. Parse Tree-



3. syntax tree:



1. Rightmost Derivation-

S → bB

→ bbBB (Using B → bBB)

→ bbBaS (Using B → aS)

→ bbBabB (Using S → bB)

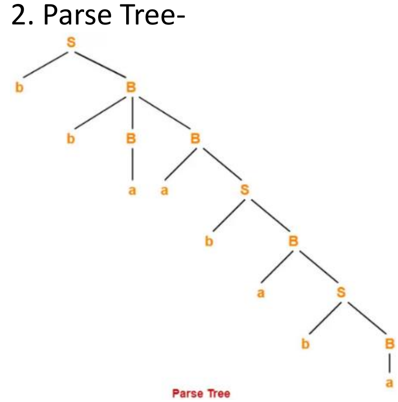
→ bbBabaS (Using B → aS)

→ bbBababB (Using S → bB)

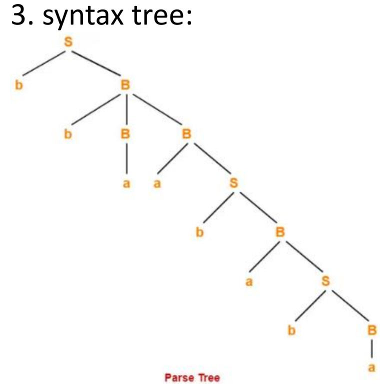
→ bbBababa (Using B → a)

→ bbaababa (Using B → a)

2. Parse Tree-



3. syntax tree:



Introduction to CFG (Context Free Grammar)

Problem-02:

Consider the grammar-

S → A1B

A → 0A / ∈

B → 0B / 1B / ∈

For the string w = 00101, find-

- Leftmost derivation
- Rightmost derivation
- Parse Tree
- Syntax Tree

Solution-

1. Leftmost Derivation-

S → A1B

→ 0A1B (Using A → 0A)

→ 00A1B (Using A → 0A)

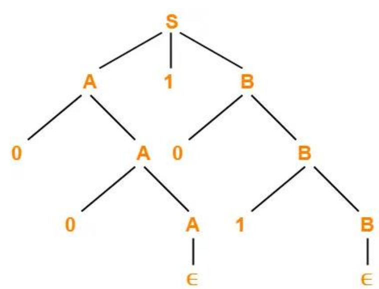
→ 001B (Using A → ∈)

→ 0010B (Using B → 0B)

→ 00101B (Using B → 1B)

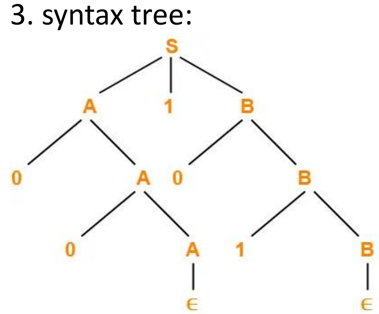
→ 00101 (Using B → ∈)

2. Parse Tree-



Parse Tree

3. syntax tree:



Parse Tree

1. Rightmost Derivation-

S → A1B

→ A10B (Using B → 0B)

→ A101B (Using B → 1B)

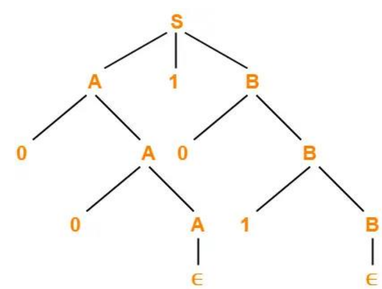
→ A101 (Using B → ∈)

→ 0A101 (Using A → 0A)

→ 00A101 (Using A → 0A)

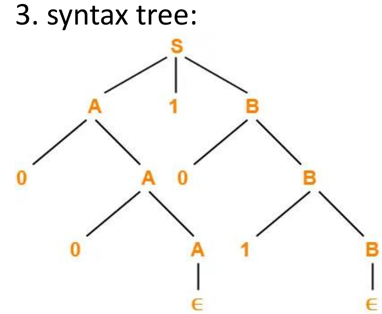
→ 00101 (Using A → ∈)

2. Parse Tree-



Parse Tree

3. syntax tree:



Parse Tree

Introduction to CFG (Context Free Grammar)

Problem-03:
S -> aAS | aSS | ∈
A -> SbA | ba
string to derive "aabbaa". find-

- Leftmost derivation
- Rightmost derivation
- Parse Tree
- Syntax tree

1. Left Derivation Tree
S
→ aAS
→ aSbA S
→ a aSS bA S (S → aSS for the S in A)
→ a a ε S bA S (S → ε)
→ a a S bA S
→ a a ε bA S (first S → ε)
→ a a bA S
→ a a b ba S (A → ba)
→ a a b b a S
→ a a b b a ε (S → ε)
→ a a b b a a

2. Parse tree:

3. Syntax tree:

1. Right Derivation Tree
S
→ aAS
→ aA aAS (right S → aAS)
→ aA aA ε (last S → ε)
→ aSbA aA ε (A → SbA)
→ aεbA aA (S → ε)
→ abA aA
→ abba aA (A → ba)
→ abba a ba
→ aabbaa

2. Parse tree:

3. Syntax tree:

Introduction to CFG (Context Free Grammar)

Problem-04:

Suppose the following Production rules of a Grammar (G)

S → S + S | S * S

S → x|y|z

and Input is (x * y + Z). find-

- Leftmost derivation
- Rightmost derivation
- Parse Tree
- Syntax tree

Solution:

1. Left derivation:

S

→ S + S (using S → S + S)

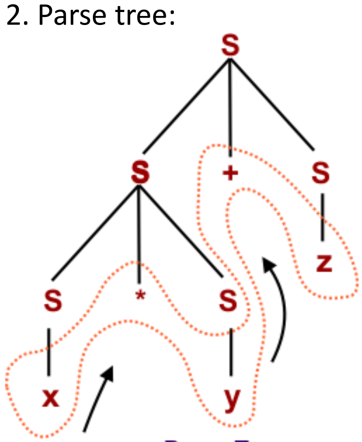
→ S * S + S (leftmost S → S * S)

→ x * S + S (leftmost S → x)

→ x * y + S (next S → y)

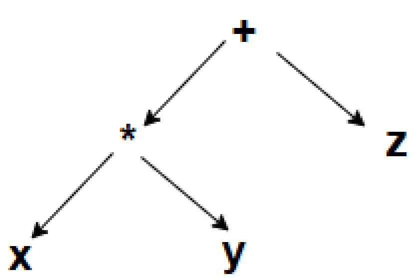
→ x * y + z (final S → z)

2. Parse tree:



Parse Tree

3. Syntax tree:



Right derivation:

S

→ S + S (S → S + S)

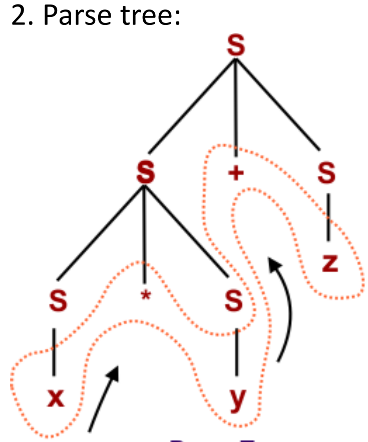
→ S * S + S (rightmost S → S * S)

→ x * S + S (rightmost S → x)

→ x * y + S (next S → y)

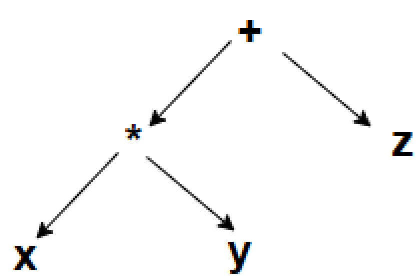
→ x * y + z (final S → z)

2. Parse tree:



Parse Tree

3. Syntax tree:



Introduction to CFG (Context Free Grammar)

Problem-05:

Suppose the following Production rules of a Grammar (G)

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

and Input is aaabbabbba. find-

- Leftmost derivation
- Rightmost derivation
- Parse Tree
- Syntax tree

Solution:

Leftmost Derivation-

S → a**B**

→ aa**B**B (Using B → aBB)

→ aaa**B**BB (Using B → aBB)

→ aaab**B**B (Using B → b)

→ aaabb**B** (Using B → b)

→ aaabba**B**B (Using B → aBB)

→ aaabbab**B** (Using B → b)

→ aaabbabb**S** (Using B → bS)

→ aaabbabbb**A** (Using S → bA)

→ aaabbabbba (Using A → a)

Rightmost Derivation-

S → aB

→ aaBB (Using B → aBB)

→ aaBaBB (Using B → aBB)

→ aaBaBbS (Using B → bS)

→ aaBaBbbA (Using S → bA)

→ aaBaBbba (Using A → a)

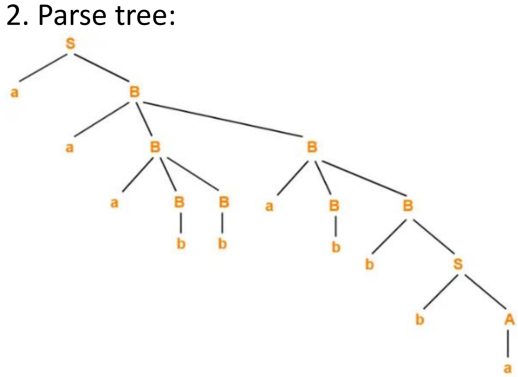
→ aaBabbba (Using B → b)

→ aaaBBabbba (Using B → aBB)

→ aaaBbabbba (Using B → b)

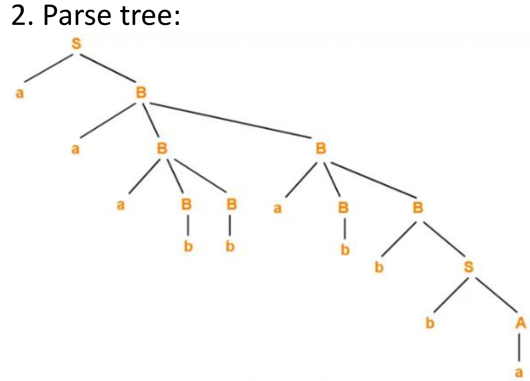
→ aaabbabbba (Using B → b)

2. Parse tree:



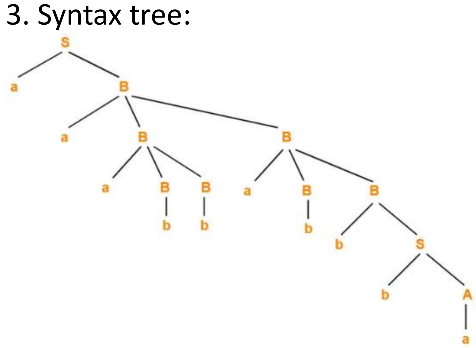
Leftmost Derivation Tree

2. Parse tree:



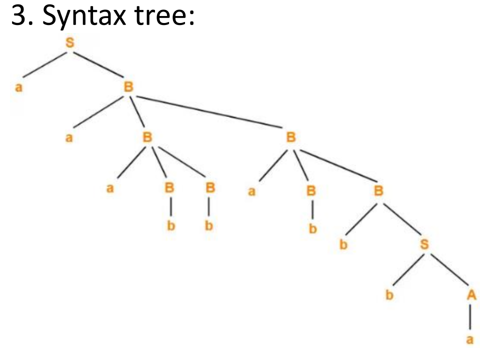
Rightmost Derivation Tree

3. Syntax tree:



Leftmost Derivation Tree

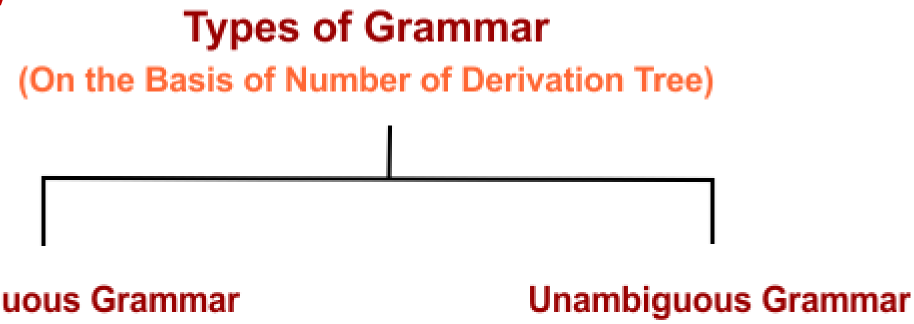
3. Syntax tree:



Leftmost Derivation Tree

Introduction to CFG (Context Free Grammar)

3. Ambiguity in CFG



1. Ambiguous Grammar

A grammar is said to be ambiguous grammar if any string generated by it produces more than one Parse tree Or syntax tree Or leftmost derivation Or rightmost derivation.

Examples of Ambiguous Grammar

<u>**Example 01**</u>

Check whether the following grammar is ambiguous or not for string w = ab

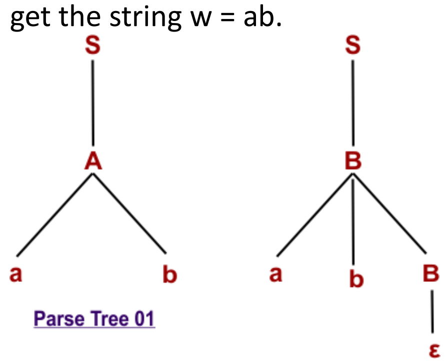
S → A / B

A → aAb / ab

B → abB / ∈

<u>Solution</u>

Now, we draw more than one parse tree to get the string w = ab.



Parse Tree 01

Parse Tree 02

The original string (w =ab) can derived through two different parse trees. So, the given grammar is ambiguous.

<u>**Example-02**</u>

Check whether the following grammar is ambiguous or not for string w = aabbccdd

S → AB / C

A → aAb / ab

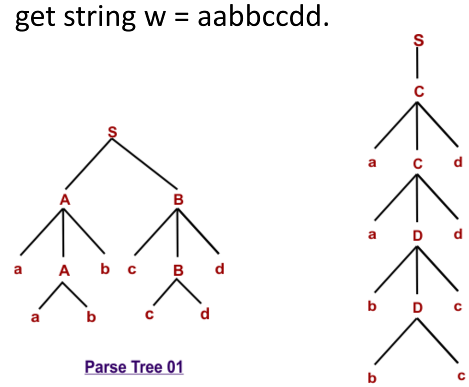
B → cBd / cd

C → aCd / aDd

D → bDc / bc

<u>Solution</u>

Now we draw more than one parse tree to get string w = aabbccdd.



Parse Tree 01

Parse Tree 02

The original string (w =aabbccdd) can derived through two different parse trees. So, the given grammar is ambiguous.

Introduction to CFG (Context Free Grammar)

2. Unambiguous Grammar

A grammar is said to be unambiguous grammar if every string generated by it produces exactly one Parse tree Or syntax tree Or leftmost derivation Or rightmost derivation.

Note: So, If we try to derive more than one tree of unambiguous grammar, then all trees will be similar.

Examples of Unambiguous Grammar

Example 01

For string “aab” the following grammar is unambiguous

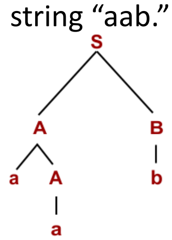
S → AB

A → Aa | a

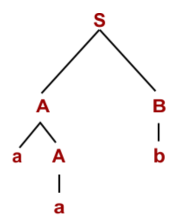
B → b

Solution

Let’s draw the leftmost and rightmost derivations of the above grammar to get the string “aab.”



Left Most Derivation



Rightmost Derivation

Because all parse trees, syntax trees, and left or right derivations will be similar for the above grammar of string “aab.” So, the above grammar is unambiguous.

Example 02

For string “id+id*id,” the following grammar is unambiguous

E → E + T

E → T

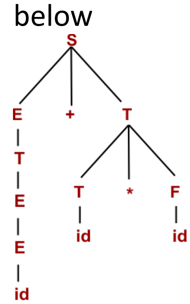
T → T * F

T → F

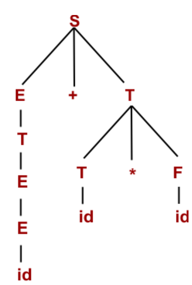
F → id

Solution

Because all parse trees, syntax trees, and left or right derivations will be similar for the above string grammar “id+id*id.” As given below



Parse Tree 1



Parse Tree 2

Introduction to CFG (Context Free Grammar)

❑ General Approach To Check Grammar Ambiguity-

To check whether a given grammar is ambiguous or not, we follow the following steps-

Step-01:

We try finding a string from the Language of Grammar such that for the string there exists more than one-

- parse tree
- or derivation tree
- or syntax tree
- or leftmost derivation
- or rightmost derivation

Step-02:

If there exists at least one such string, then the grammar is ambiguous otherwise unambiguous.

❖ **PROBLEMS BASED ON CHECKING WHETHER GRAMMAR IS AMBIGUOUS-**

Problem-01: Check whether the given grammar is ambiguous or not-

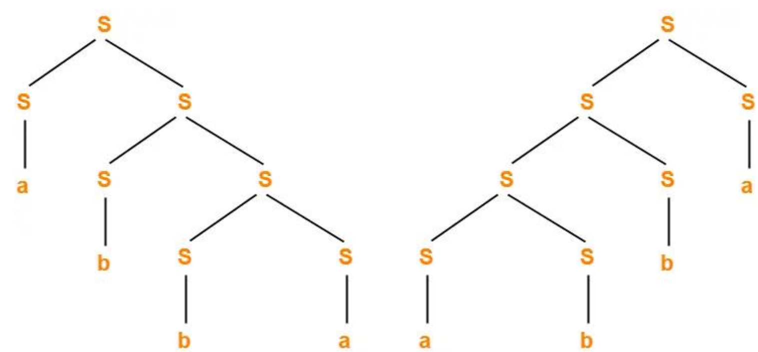
S → SS
S → a
S → b

Solution-

Let us consider a string w generated by the given grammar-

w = abba

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem-02: Check whether the given grammar is ambiguous or not-

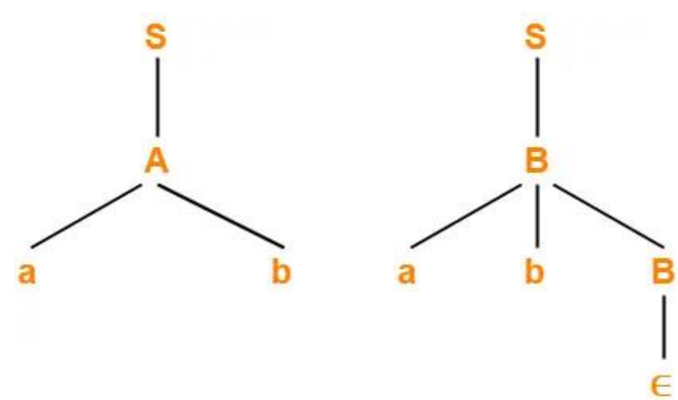
S → A / B
A → aAb / ab
B → abB / ∈

Solution-

Let us consider a string w generated by the given grammar-

w = ab

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Introduction to CFG (Context Free Grammar)

Problem-03: Check whether the given grammar is ambiguous or not-

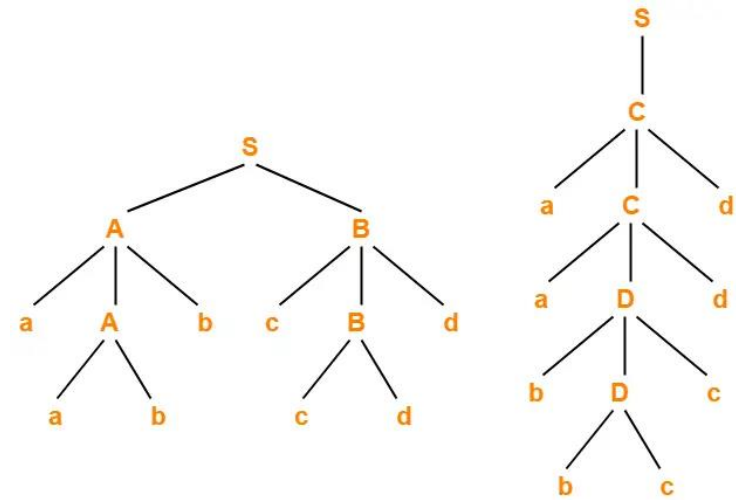
S → AB / C
A → aAb / ab
B → cBd / cd
C → aCd / aDd
D → bDc / bc

Solution-

Let us consider a string w generated by the given grammar-

w = aabbccdd

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem-04:

Check whether the given grammar is ambiguous or not-

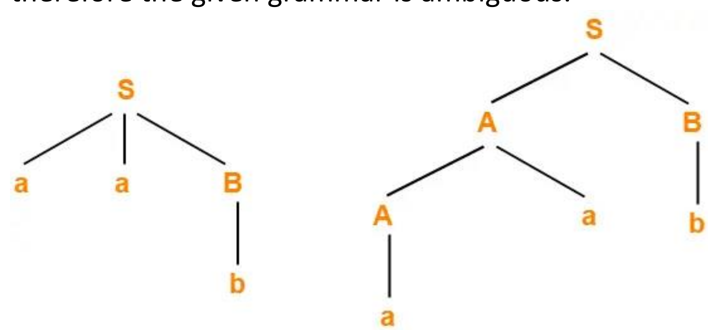
S → AB / aaB
A → a / Aa
B → b

Solution-

Let us consider a string w generated by the given grammar-

w = aab

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem-05:

Check whether the given grammar is ambiguous or not-

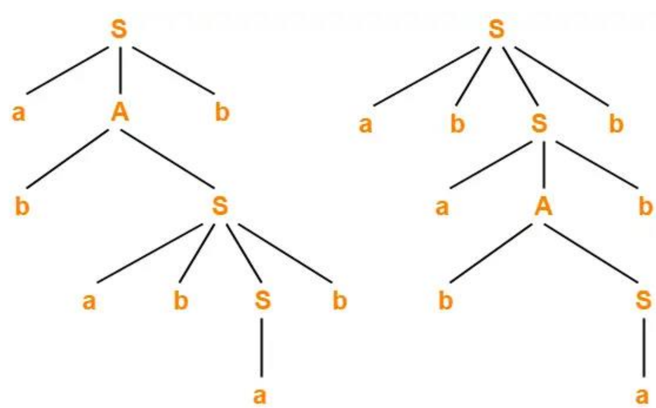
S → a / abSb / aAb
A → bS / aAAb

Solution-

Let us consider a string w generated by the given grammar-

w = abababb

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Introduction to CFG (Context Free Grammar)

Problem-06:
Check whether the given grammar is ambiguous or not-

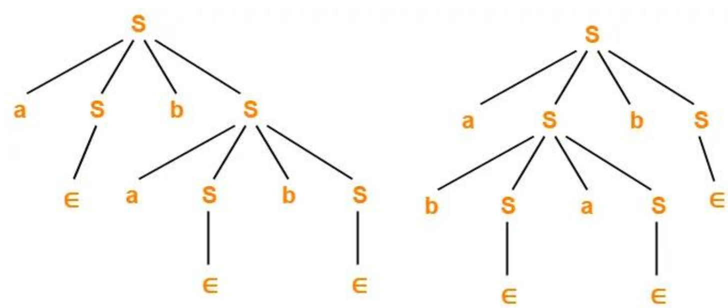
E → E + T / T
T → T x F / F
F → id

Solution-
There exists no string belonging to the language of grammar which has more than one parse tree. Since a unique parse tree exists for all the strings, therefore the given grammar is unambiguous.

Problem-07:
Check whether the given grammar is ambiguous or not-

S → aSbS / bSaS / ∈

Solution-
Let us consider a string w generated by the given grammar-
w = abab
Now, let us draw parse trees for this string w.



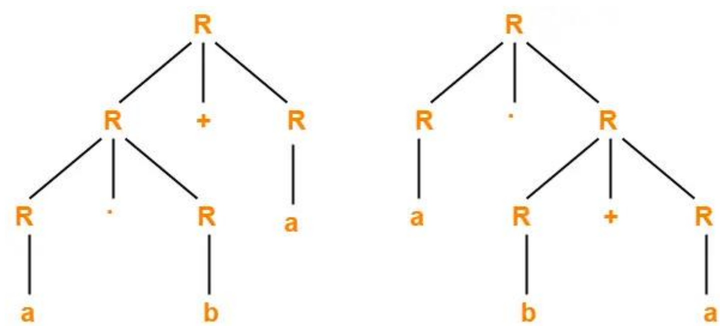
Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem-08:
Check whether the given grammar is ambiguous or not-

R → R + R / R . R / R* / a / b

Solution-
Let us consider a string w generated by the given grammar-
w = ab + a
Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Introduction to CFG (Context Free Grammar)

❑ Ambiguity and Limitations of CFG

One big challenge with using CFGs in compilers is ambiguity. Ambiguity happens when the same piece of code can be interpreted in more than one way, which makes it unclear what the actual structure should be.

$$\text{Statement} \rightarrow \text{if Expression then Statement} \mid \text{if Expression then Statement else Statement}$$

Figure 2: A common example is the "if-then-else" problem.

If you have two if statements followed by an else, the compiler might get confused about which if the else is supposed to match with. This makes it hard to figure out the intended structure of the code.

To deal with this, compiler designers use **disambiguation rules**. A typical rule is that an else clause always pairs with the closest if statement. Another way to resolve ambiguity is using advanced parsing techniques, like operator precedence, to handle tricky cases and make sure things are interpreted correctly.

Another limitation of CFGs is that they can't handle everything in modern programming languages. Some languages have context-sensitive rules, meaning that how certain parts of the code should be understood depends on the surrounding context. Since CFGs are context-*free*, they can't directly handle those cases. To get around this, compilers use additional methods like attribute grammars or extra checks during the semantic analysis phase.

Another limitation of CFG is that it cannot handle all the features of modern programming languages. Some languages have **context-sensitive** rules, which require the meaning of a part of the code to depend on the context in which it appears. Since CFG is **context-free**, it cannot handle such cases directly. To address this, compilers often use **attribute grammars** or perform additional checks during semantic analysis (*6.0 Semantic Analysis Translation and Attribute Grammars*, 2024).

Introduction to CFG (Context Free Grammar)

4. Removal of Ambiguity from CFG

Removing Ambiguity By Precedence & Associativity Rules-

An ambiguous grammar may be converted into an unambiguous grammar by implementing-

- Precedence Constraints
- Associativity Constraints

Associative:

Left associative: *,/ > . > +,-

Right associative: ↑

Precedence:

@ > # > $

PROBLEMS BASED ON CONVERSION INTO UNAMBIGUOUS GRAMMAR-

Problem-01:

Convert the following ambiguous grammar into unambiguous grammar-

R → R + R
R → R . R
R → R*
R → a | b

Solution-

Using the precedence and associativity rules, we write the corresponding unambiguous grammar as-

R → R + T | T
T → T . F | F
F → F* | G
G → a | b

Problem-02:

Convert the following ambiguous grammar into unambiguous grammar-

bexp → bexp or bexp
bexp → bexp and bexp
bexp → not bexp
bexp → T | F
where bexp represents Boolean expression, T represents True and F represents False.

Solution-

bexp → bexp or M / M
M → M and N / N
N → not N / G
G → T / F

Introduction to CFG (Context Free Grammar)

Problem-03:

Convert the following ambiguous grammar into unambiguous grammar-

E → E+E

E → E *E

E → id

Solution-

Equivalent Unambiguous Grammar

E → E+T | T

T → T*F | F

F → id

Problem-04:

Convert the following ambiguous grammar into unambiguous grammar-

E→E+E

E→E *E

E→E ^ E

E→ id

Solution-

Equivalent Unambiguous Grammar

E → E + T | T

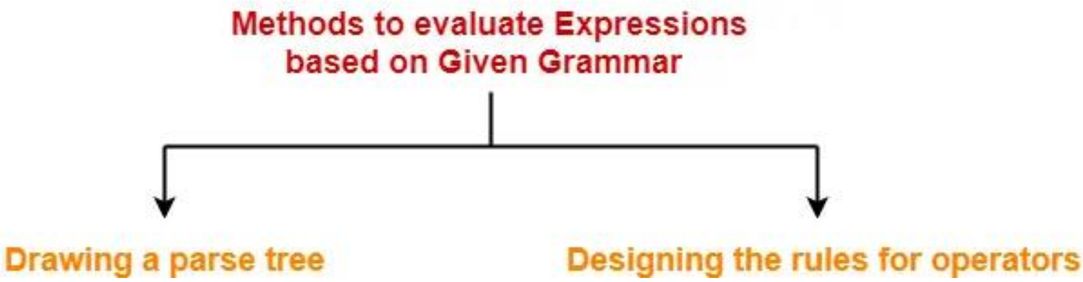
F → G ^ F |G

G→ id

Introduction to CFG (Context Free Grammar)

5. Evaluating Expressions Based on Given Grammar-

There are two methods for evaluating the expressions based on given grammar-



<u>Rules For Deciding Priority Of Operators-</u>

For the given unambiguous grammar,
The priority of operators is decided by checking the level at which the production is present.
The higher the level of production, the lower the priority of operator contained in it.
The lower the level of production, the higher the priority of operator contained in it.

<u>Example-</u>

Consider the grammar-
E → E x F / F + E / F
F → F – T / T
T → id
Here,
id has the highest priority.
(since T → id is present at the bottom most level)
x and + operators have the least priority.
(since E → E x F / F + E are present at the top most level)
x and + operators have the same priority.
(since E → E x F / F + E are present at the same level)
– operator has higher priority than x and + but lesser priority than id.
(since F → F – T is present at the middle level)

So, the priority order is-
id > **–** > (**x** , **+**)

<u>Rules For Deciding Associativity Of Operators-</u>

For the given unambiguous grammar,
The associativity of operators is decided by checking the Type of Recursion in the production.
If the production has left recursion, then the operator is left associative.
If the production has right recursion, then the operator is right associative.
If the production has both left and right recursion, then the operator is neither left associative nor right associative.

<u>Example-</u>

Consider the grammar-
E → E x F / F + E / F
F → F – F / T
T → id

Here,
x operator is left associative.
(since E → E x F has left recursion in it)
+ operator is right associative.
(since E → F + E has right recursion in it)
F – F is neither left associative nor right associative.
(since F → F – F has both left and right recursion in it)

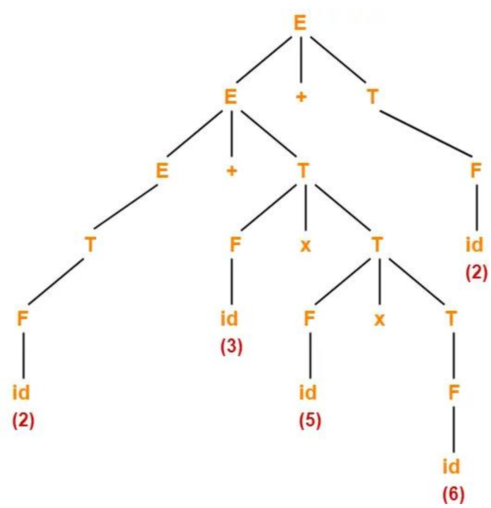
Introduction to CFG (Context Free Grammar)

PRACTICE PROBLEMS BASED ON EVALUATING EXPRESSIONS FOR GRAMMAR-

Problem-01: Consider the given grammar-
E → E + T / T
T → F x T / F
F → id
Evaluate the following expression in accordance with the given grammar-
2 + 3 x 5 x 6 + 2
Solution-

Method-01:
Let us draw a parse tree for the given expression.
Evaluating the parse tree will return the required value of the expression.

The parse tree for the given expression is-



On evaluating this parse tree, we get the value = 94.

Method-02:

The priority order and associativity of operators on the basis of given grammar is-
id > x > +
where-
x operator is right associative.
+ operator is left associative.

Now, we parenthesize the given expression based on the precedence and associativity of operators as-
(2 + (3 x (5 x 6))) + 2

Now, we evaluate the parenthesized expression as-
= (2 + (3 x (5 x 6))) + 2
= (2 + (3 x 30)) + 2
= (2 + 90) + 2
= 92 + 2
= 94

Introduction to CFG (Context Free Grammar)

Problem-02: Consider the given grammar-

$E \rightarrow E + T / E - T / T$

$T \rightarrow T \times F / T \div F / F$

$F \rightarrow G \uparrow F / G$

$G \rightarrow id$

Evaluate the following expression in accordance with the given grammar-

$2 \times 1 + 4 \uparrow 2 \uparrow 1 \times 1 + 3$

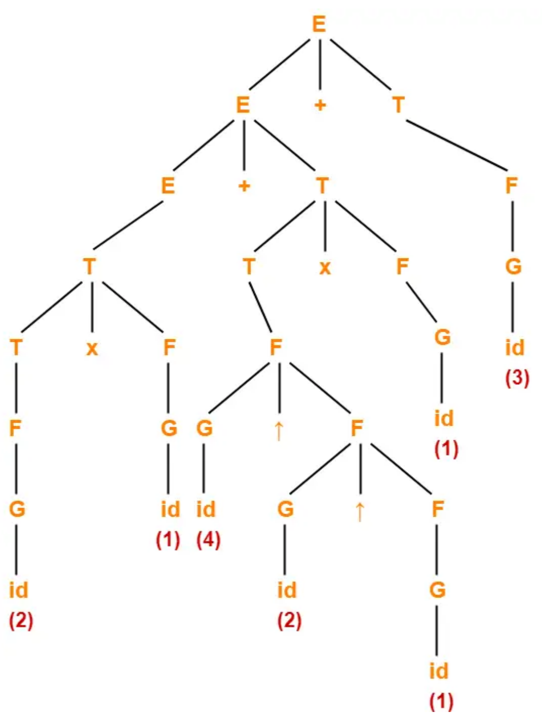
Solution-

Method-01:

Let us draw a parse tree for the given expression.

Evaluating the parse tree will return the required value of the expression.

The parse tree for the given expression is-



On evaluating this parse tree, we get the value = 21.

Method-02:

The priority order and associativity of operators on the basis of given grammar is-

id > **↑** > (**x** , **÷**) > (**+** , **–**)

where-

+ , – , x , ÷ operators are left associative.

↑ operator is right associative

Now, we parenthesize the given expression based on the precedence and associativity of operators as-

$((2 \times 1) + ((4 \uparrow (2 \uparrow 1)) \times 1)) + 3$

Now, we evaluate the parenthesized expression as-

$= ((2 \times 1) + ((4 \uparrow \mathbf{(2 \uparrow 1)}) \times 1)) + 3$

$= ((2 \times 1) + (\mathbf{(4 \uparrow 2)} \times 1)) + 3$

$= (\mathbf{(2 \times 1)} + (16 \times 1)) + 3$

$= (2 + \mathbf{(16 \times 1)}) + 3$

$= \mathbf{(2 + 16)} + 3$

$= \mathbf{18 + 3}$

$= 21$

Introduction to CFG (Context Free Grammar)

Problem-03:

Consider the given grammar-

E → E + T / E – T / T

T → T x F / T ÷ F / F

F → F ↑ G / G

G → id

Evaluate the following expression in accordance with the given grammar-

2 ↑ 1 ↑ 4 + 3 x 5 x 6 ↑ 1 + 2 ↑ 3

Solution-

The priority order and associativity of operators on the basis of given grammar is-

id > **↑** > (**x** , **÷**) > (**+** , **–**)

where + , – , x , ÷, ↑ operators are left associative.

Now, we parenthesize the given expression based on the precedence and associativity of operators as-

(((2 ↑ 1) ↑ 4) + ((3 x 5) x (6 ↑ 1))) + (2 ↑ 3)

Now, we evaluate the parenthesized expression as-

= (((**2 ↑ 1**) ↑ 4) + ((3 x 5) x (6 ↑ 1))) + (2 ↑ 3)

= (**(2 ↑ 4)** + ((3 x 5) x (6 ↑ 1))) + (2 ↑ 3)

= (16 + ((3 x 5) x (**6 ↑ 1**))) + (2 ↑ 3)

= (16 + ((3 x 5) x 6)) + **(2 ↑ 3)**

= (16 + (**(3 x 5)** x 6)) + 8

= (16 + **(15 x 6)**) + 8

= **(16 + 90)** + 8

= **106 + 8**

= 114

Problem-04:

Consider the given grammar-

E → E ↑ T / T

T → T + F / F

F → G – F / G

G → id

Evaluate the following expression in accordance with the given grammar-

2 ↑ 1 ↑ 3 + 5 – 6 – 8 – 5 + 10 + 11 ↑ 2

Solution-

The priority order and associativity of operators on the basis of given grammar is-

id > **–** > **+** > **↑**

where-

+ , ↑ operators are left associative.

– is right associative.

Now, we parenthesize the given expression based on the precedence and associativity of operators as-

((2 ↑ 1) ↑ (((3 + (5 – (6 – (8 – 5)))) + 10) + 11)) ↑ 2

Now, we evaluate the parenthesized expression as-

= ((2 ↑ 1) ↑ (((3 + (5 – (6 – **(8 – 5)**))) + 10) + 11)) ↑ 2

= ((2 ↑ 1) ↑ (((3 + (5 – **(6 – 3)**)) + 10) + 11)) ↑ 2

= ((2 ↑ 1) ↑ (((3 + **(5 – 3)**) + 10) + 11)) ↑ 2

= ((2 ↑ 1) ↑ ((**(3 + 2)** + 10) + 11)) ↑ 2

= ((2 ↑ 1) ↑ (**(5 + 10)** + 11)) ↑ 2

= ((2 ↑ 1) ↑ **(15 + 11)**) ↑ 2

= (**(2 ↑ 1)** ↑ 26) ↑ 2

= **(2 ↑ 26)** ↑ 2

= **(2^{26}) ↑ 2**

= $(2^{26})^2$