Pushdown Automata Introduction

Basic Structure of PDA

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is −

"Finite state machine" + "a stack"

A pushdown automaton has three components −

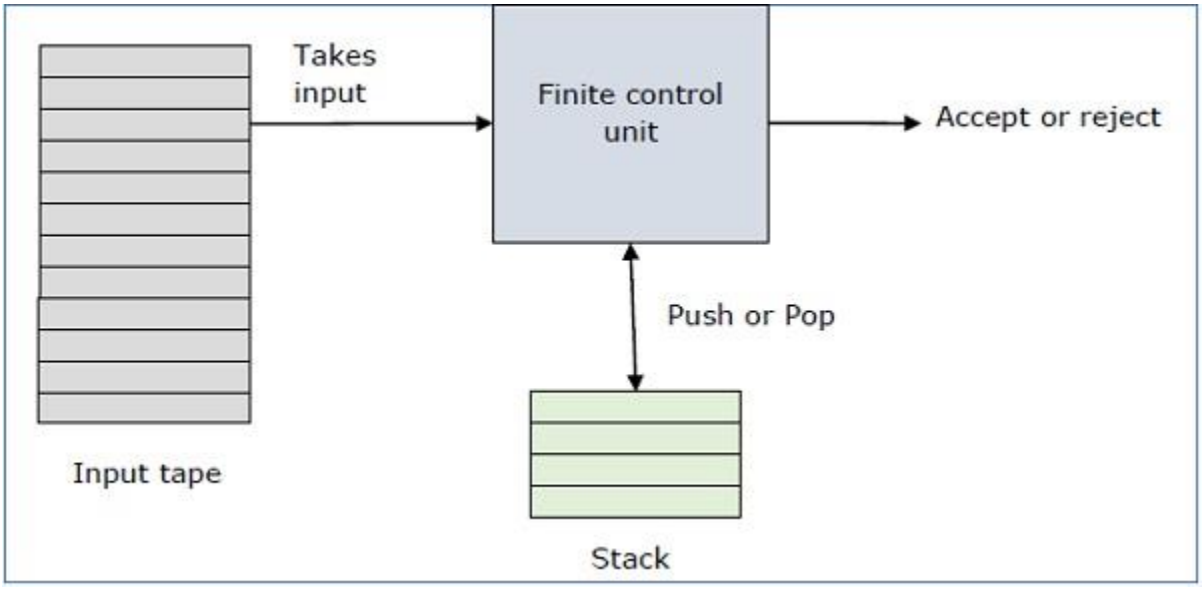
- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations −

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

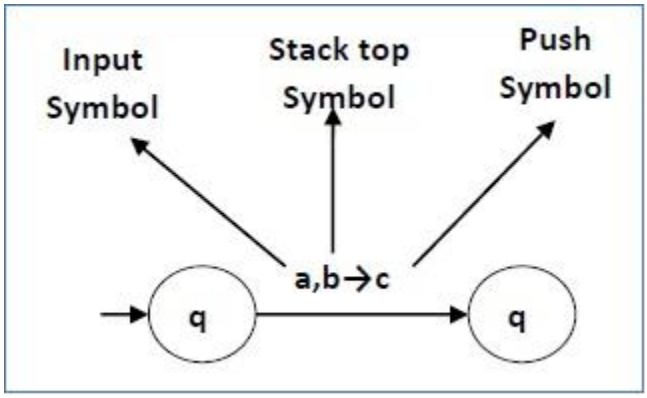


A PDA can be formally described as a 7-tuple (Q, ∑, S, δ, q_0, I, F) −

- **Q** is the finite number of states
- ∑ is input alphabet

- **S** is stack symbols
- **δ** is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*
- **q_0** is the initial state (q_0 ∈ Q)
- **I** is the initial stack top symbol (I ∈ S)
- **F** is a set of accepting states (F ∈ Q)

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as a,b → c −



Terminologies Related to PDA

Instantaneous Description

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- **q** is the state
- **w** is unconsumed input
- **s** is the stack contents

Turnstile Notation

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol "⊢".

Consider a PDA (Q, ∑, S, δ, q_0, I, F). A transition can be mathematically represented by the following turnstile notation −

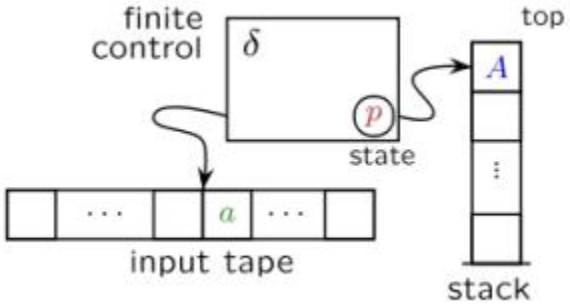
$(p, aw, T\beta) \vdash (q, w, \alpha b)$

This implies that while taking a transition from state **p** to state **q**, the input symbol **'a'** is consumed, and the top of the stack **'T'** is replaced by a new string **'α'**.

Note − If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

Non-deterministic push down automata

The Non-deterministic Push down Automata (NPDAs) are like finite automata (FA), except they also have a stack memory where they can store an arbitrary amount of information.



Read/write stack memory works as LIFO: Last In, First Out

What can we do with a stack?

The pop operation reads the top symbol and removes it from the stack, the push operation writes a designated symbol onto the top of the stack, e.g. push(X) means put X on top of the stack, the nop operation does nothing to the stack.

The stack symbols are different from the “language” alphabet used on the input tape.

We start with the following −

- With only the initial stack symbol on the stack, (nothing else on the stack!) in the start state of the control automaton.
- At each step, the state, input element and top symbol in the stack determine the next step (transition).

One transition step includes the following −

- Change the state, (as FA).
- Reading of a symbol from the input tape and moving to the next right symbol,(as FA).
- Change the stack (pushing a symbol onto the stack/popping a symbol of the stack/no changes to the stack).

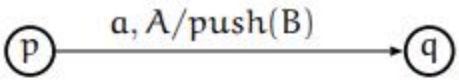
Transition steps are formally defined by transition functions (often in the form of the transition instructions).

NPDA can be described as the following −

- A finite set Q of states (& the start state & the set of accepting/final states).
- A finite set ∑ which is called the input alphabet.
- A finite set Γ which is called the stack alphabet (& the initial stack symbol $).
- A finite set of transition instructions (or a transition function T).

$T : Q \times \sum \cup \{\wedge\} \times \Gamma \rightarrow \Gamma^* \times Q$

Or it is represented by 'transition' diagram as given below −



We can rewrite the transition using transition function:

$$T(p, a, A) = (push(B), q)$$

or more often as the transition instruction

$$(p, a, A, push(B), q)$$

Context-Free Grammar

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

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

Example

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

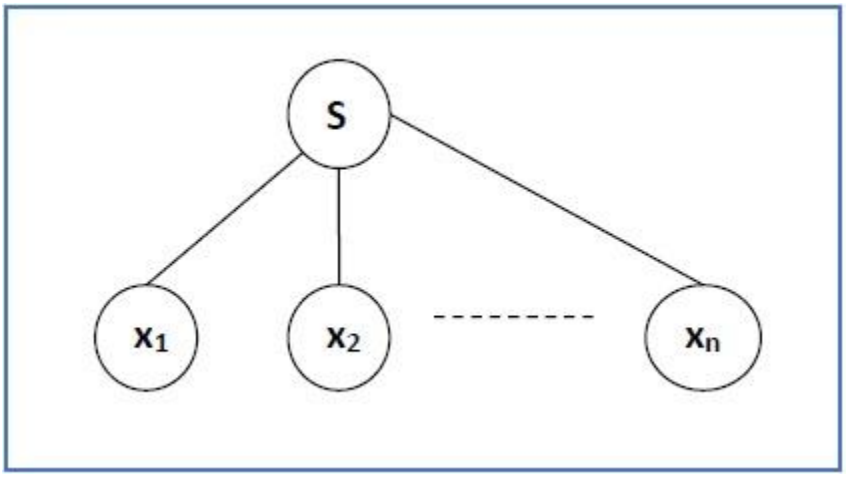
Generation of Derivation Tree

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

Representation Technique

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

If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows −



There are two different approaches to draw a derivation tree −

Top-down Approach −

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

Bottom-up Approach −

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

Derivation or Yield of a Tree

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

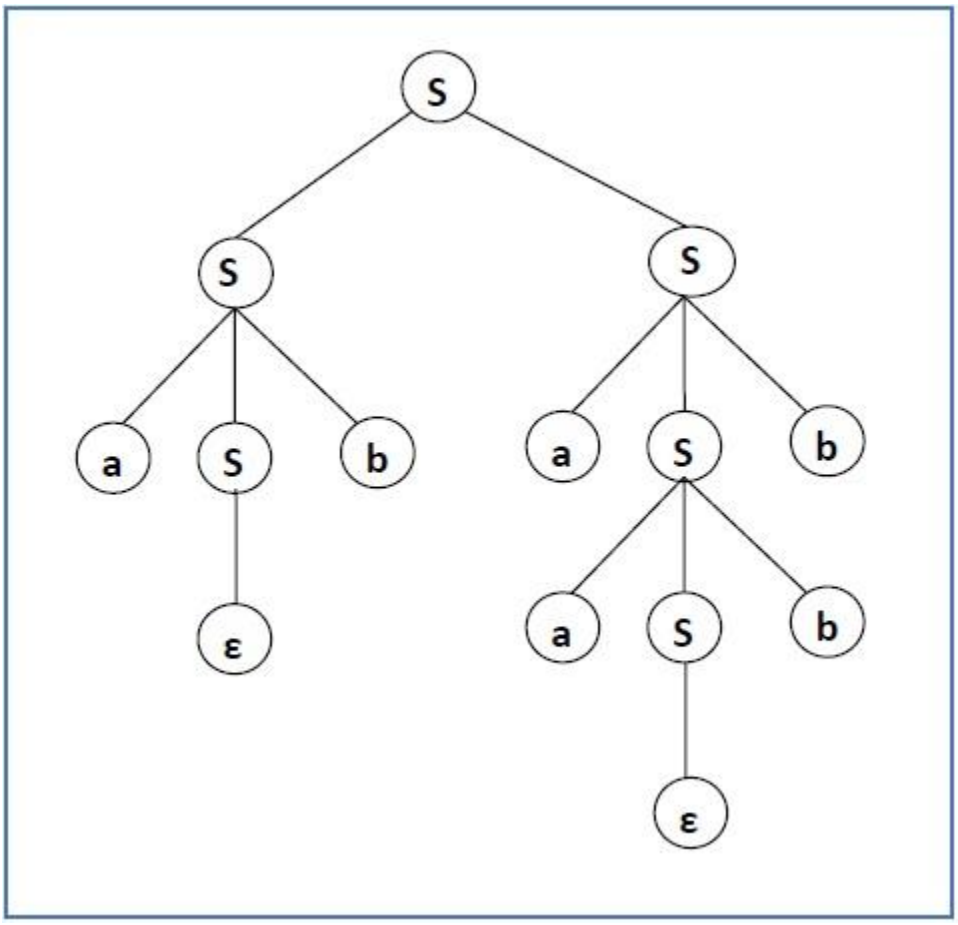
Example

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

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

One derivation from the above CFG is “abaabb”

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



Sentential Form and Partial Derivation Tree

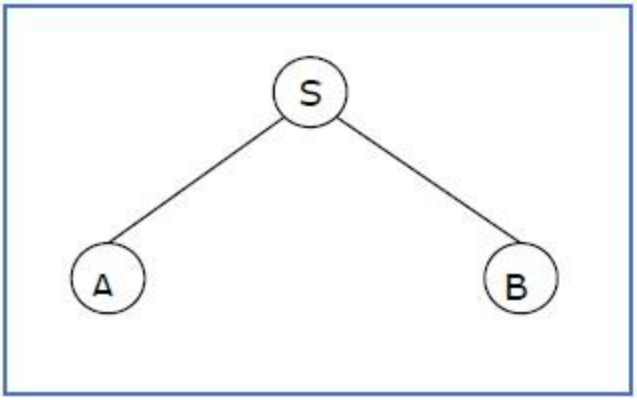
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are −

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following −



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

Leftmost and Rightmost Derivation of a String

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

Example

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

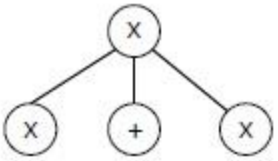
over an alphabet {a}.

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

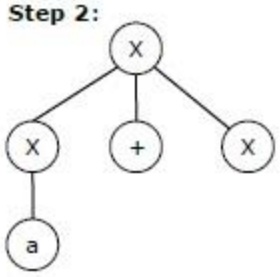
X → X+X → a+X → a + X*X → a+a*X → a+a*a

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

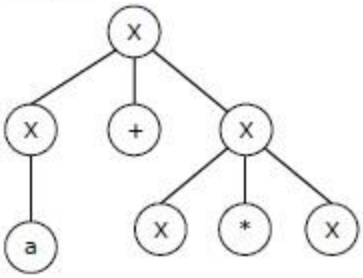
Step 1:



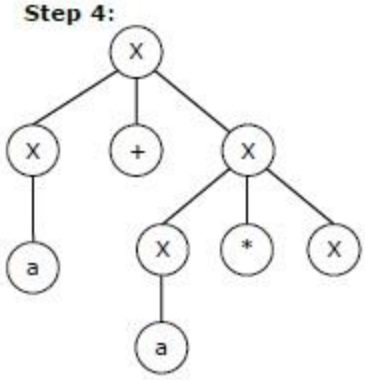
Step 2:



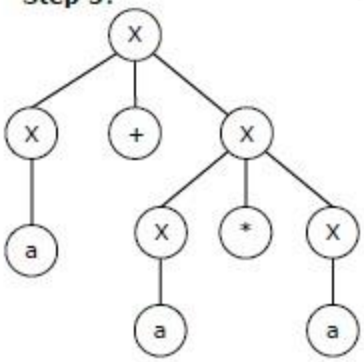
Step 3:



Step 4:



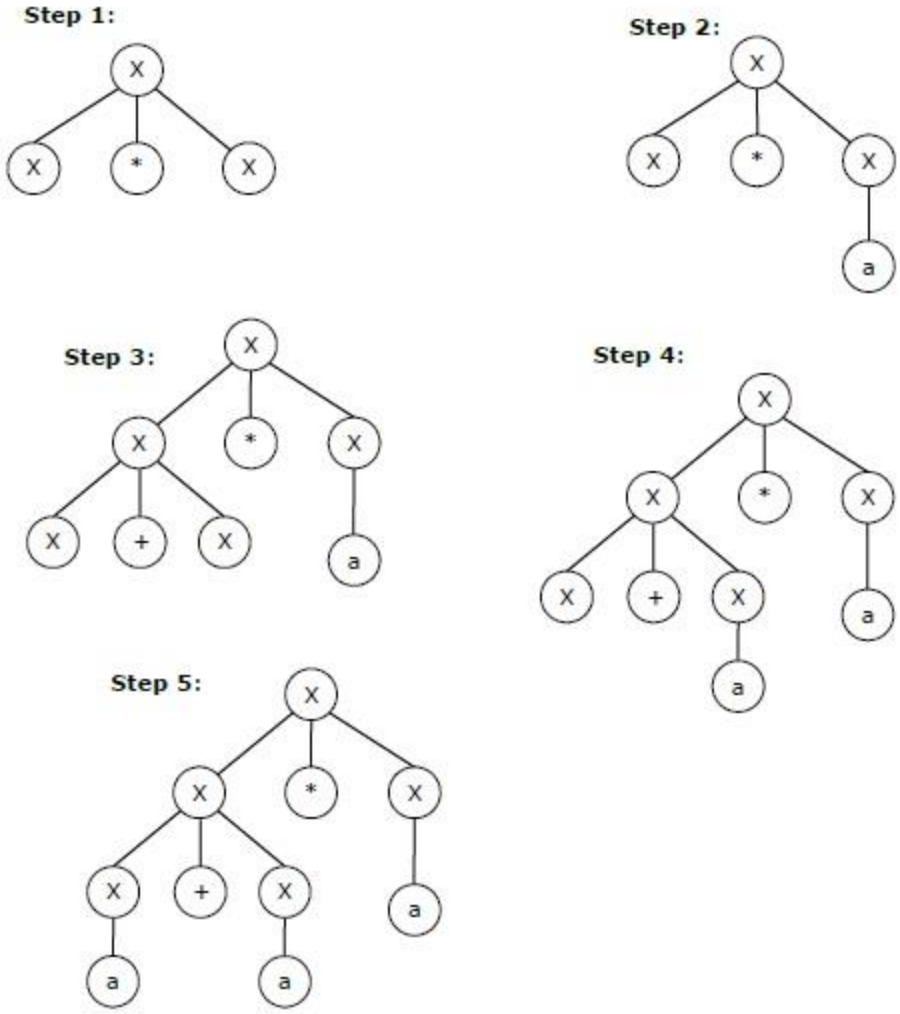
Step 5:



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

X → X*X → X*a → X+X*a → X+a*a → a+a*a

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



Left and Right Recursive Grammars

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

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

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form

What is GNF?

The Greibach normal form is referred to as GNF. A context-free grammar (CFG) is in Greibach normal form (GNF) if and only all of its production rules meet one of the criteria listed below:

- A non-terminal that generates a terminal. For instance, X → x.
- A start symbol that generates ε. For instance, S → ε.
- A non-terminal that generates a terminal followed by any number of non-terminals. For instance, S → xXSY.

Example

GA = {S → xXY | xY, X → xX| x, Y → yY | y}

GB = {S → xXY | xY, X → xX | ε, Y → yY | ε}

The production rules of the GA grammar satisfy the rules that are specified for GNF; thus, the GA grammar is in GNF. But the production rule of GB grammar does not satisfy the rules that are specified for GNF as X → ε and Y → ε contain ε (only the start symbols generate ε). Thus, the grammar GB is not present in GNF.

CFG into GNF Conversion Steps

Step 1: Conversion of the grammar into its CNF.

In case the given grammar is not present in CNF, first convert it into CNF.

Step 2: In case the grammar exits the left recursion, eliminate it.

Step 3: If any production rule is not present in GNF, convert the production rule given in the grammar into GNF form.

Example

S → AY | XX

X → x | SX

Y → y

A → x

Answer:

As the given grammar is already in CNF, and no recursion is left, we can skip the first and the second steps and directly go to the third step.

The production rule X → SX is not in GNF. Thus, substitute S → AY | XX in the production rule X → SX as:

S → AY | XX

X → x | AYX | XXX

Y → y

A → x

The production rule S → AY and Y → AYX is not in GNF. Thus, substitute A → x in the production rule S → AY and Y → AYX as:

S → xY | XX

X → x | xYX | XXX

Y → y

A → x

We will now remove left recursion (X → XXX), then get:

S → xY | XX

X → xZ | xYXZ

Z → XXZ | ε

Y → y

A → x

Removal of the null production Z → ε would get:

S → xY | XX

X → xZ | xYXZ | x | xYX

Z → XXZ | XX

Y → y

A → x

The production rule S → XX is not in GNF; thus, substitute X → xZ | xYXZ | x | xYX in production rule S → XX as:

S → xY | xZX | xYXZX | xX | xYXX

X → xZ | xYXZ | x | xYX

Z → XXZ

Z → xZX | xYXZX | xX | xYXX

Y → y

A → x

The production rule C → XXZ is not in GNF, so substitute X → xZ | xYXZ | x | xYX in production rule Z → XXZ as:

S → xY | xZX | xYXZX | xX | xYXX

X → xZ | xYXZ | x | xYX

Z → xZXZ | xYXZXZ | xXZ | xYXXZ

Z → xZX | xYXZX | xX | xYXX

Y → y

A → x

Hence, this is the GNF form for the grammar G.

Ambiguity in grammar

A grammar is said to be ambiguous if there exists more than one left most derivation or more than one right most derivation or more than one parse tree for a given input string.

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

Example

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

E = I

E = E+E

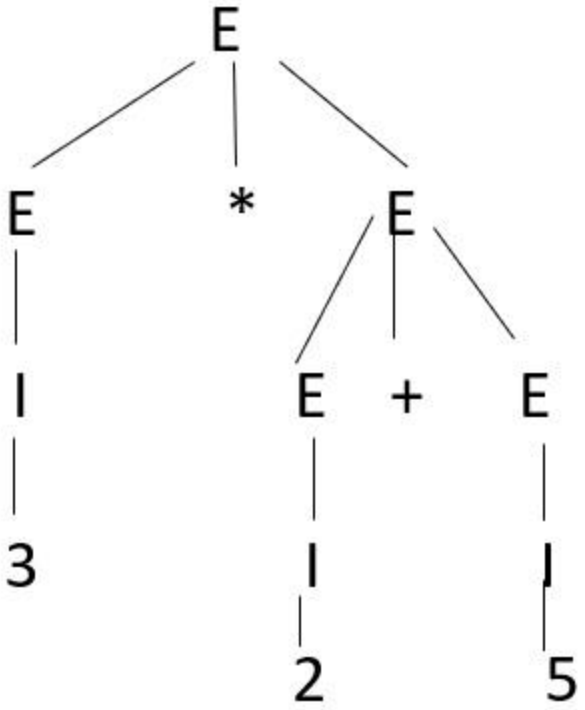
E = E*E

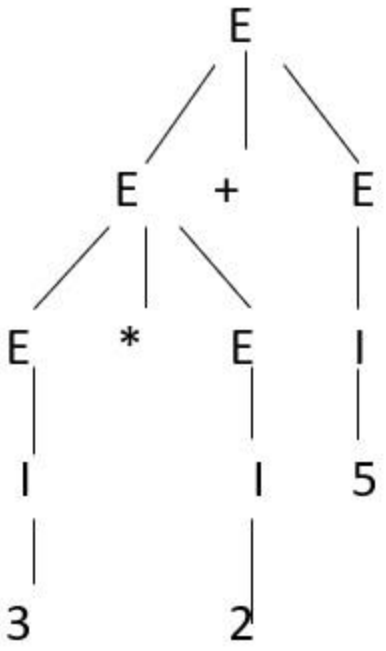
E = (E)

E = ε|0|1|2|3...9

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

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





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

Parse Trees (Derivation Trees)

Derivations mean replacing a given string's non-terminal by the right-hand side of the production rule. The sequence of applications of rules that makes the completed string of terminals from the starting symbol is known as derivation. The parse tree is the pictorial representation of derivations. Therefore, it is also known as derivation trees. The derivation tree is independent of the other in which productions are used.

A parse tree is an ordered tree in which nodes are labeled with the left side of the productions and in which the children of a node define its equivalent right parse tree also known as syntax tree, generation tree, or production tree.

A Parse Tree for a CFG G =(V,∑, P,S) is a tree satisfying the following conditions −

- Root has the label S, where S is the start symbol.
- Each vertex of the parse tree has a label which can be a variable (V), terminal (Σ), or ε.
- If A → $C_1,C_2…….C_n$ is a production, then $C_1,C_2…….C_n$ are children of node labeled A.
- Leaf Nodes are terminal (Σ), and Interior nodes are variable (V).
- The label of an internal vertex is always a variable.
- If a vertex A has k children with labels $A_1,A_2…….A_k$,then A →

$A_1,A_2…….A_k$ will be production in context-free grammar G.

Yield − Yield of Derivation Tree is the concatenation of labels of the leaves in left to right ordering.

Example1 − If CFG has productions.

S → a A S | a

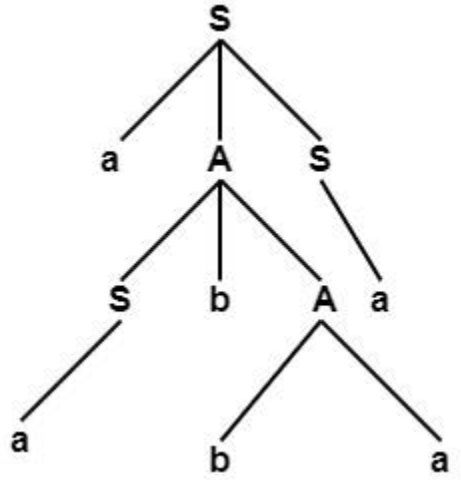
S → Sb A | SS | ba

Show that S ⇒ *aa bb aa & construct parse tree whose yield is aa bb aa.

Solution

$S \Rightarrow_{lm}$ lm a A S
⇒ a Sb A S
⇒ aa b AS
⇒ aa bba S
∴ S ⇒ * aa bb aa

Derivation Tree



Yield = Left to Right Ordering of Leaves = aa bb aa

Example2

Consider the CFG

$S \rightarrow bB \mid aA$

$A \rightarrow b \mid bS \mid aAA$

$B \rightarrow a \mid aS \mid bBB$

Find (a) Leftmost

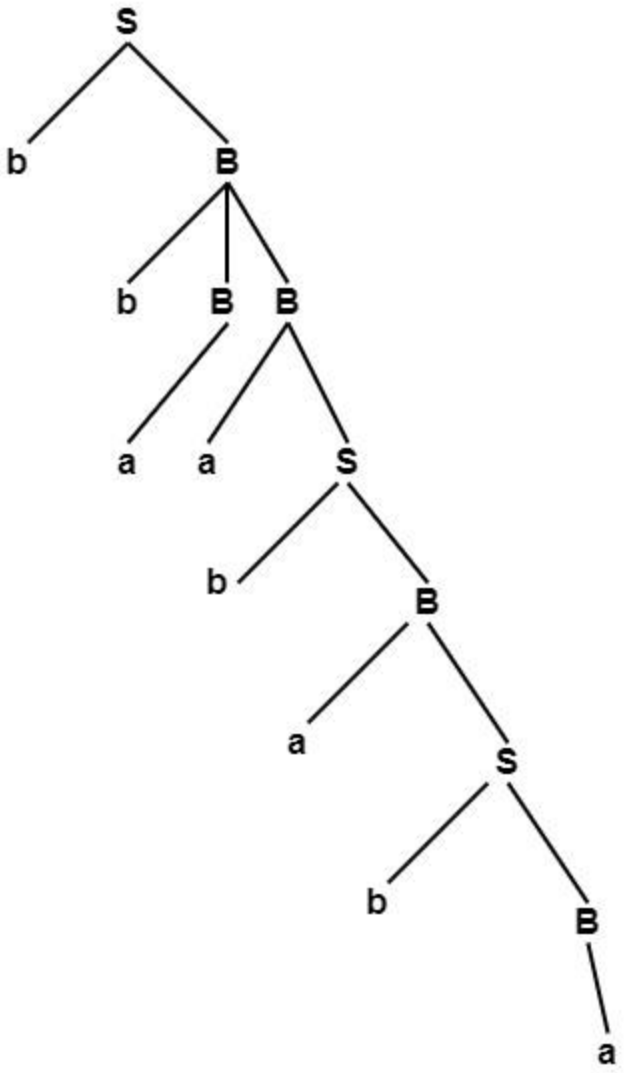
- Rightmost Derivation for string b aa baba. Also, find derivation Trees.

Solution

- **Leftmost Derivation**

S⇒b $\underline{B}$
⇒ bb $\underline{B}$B
⇒ bba$\underline{B}$
⇒ bbaa$\underline{S}$
⇒ bb aa$\underline{bB}$
⇒ bb aa b $\underline{aS}$
⇒ bb aa bab $\underline{B}$
⇒ bb aa ba ba

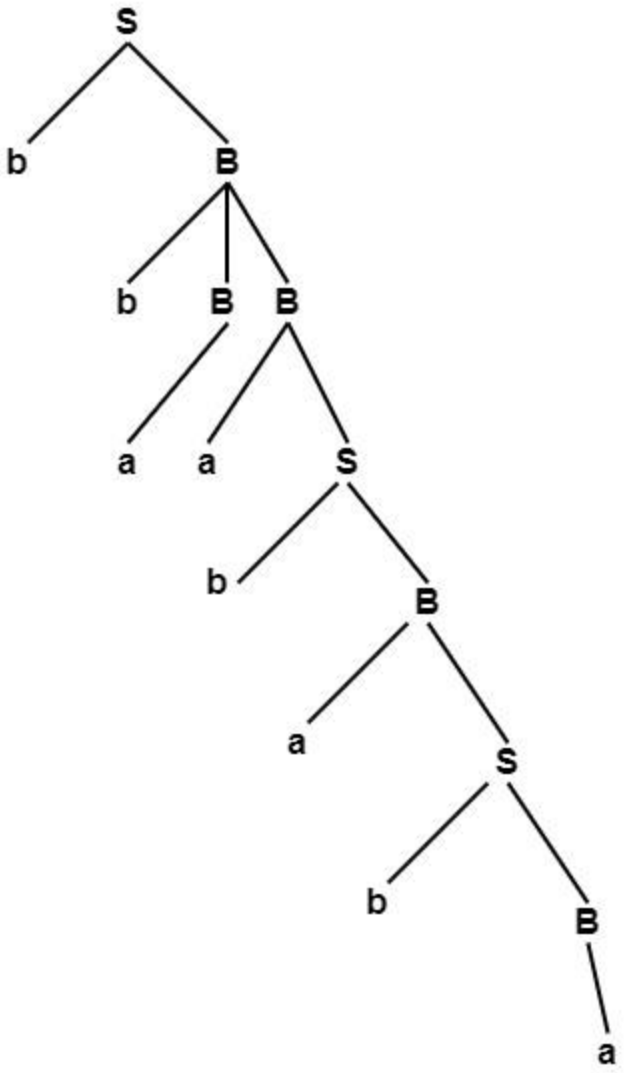
Derivation Tree for Leftmost Derivation



- **Rightmost Derivation**

S ⇒ b<u>B</u>
⇒ bb B<u>B</u>
⇒ bbBa<u>S</u>
⇒ bbBab<u>B</u>
⇒ bbBaba<u>S</u>
⇒ bbBababB
⇒ bb<u>B</u>abab a
⇒ bbaababa

Derivation Tree for Rightmost Derivation



Example3 − Consider the Grammar given below −

E⇒ E+E|E ∗∗ E|id
Find

- Leftmost
- Rightmost Derivation for the string.

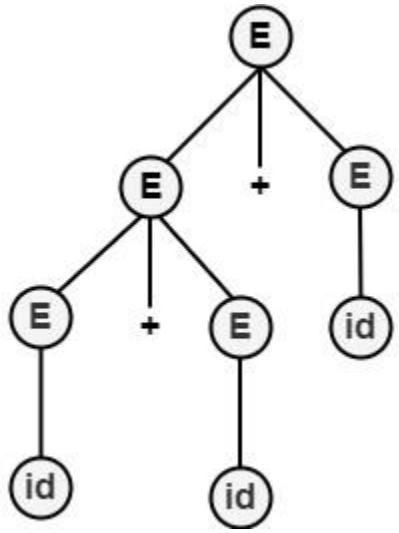
Solutio

- Leftmost Derivation

E ⇒ $\underline{\mathrm{E}}$+E
⇒ $\underline{\mathrm{E}}$+E+E
⇒ id+E+E

⇒ id+id+E

⇒ id+id+id



- **Rightmost Derivation**

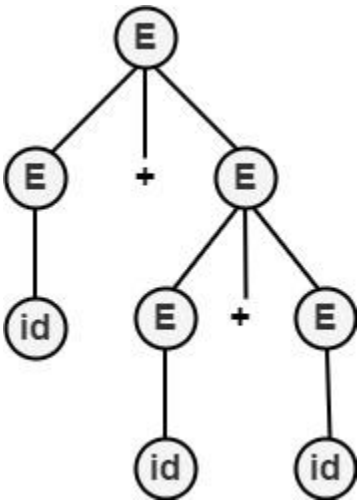
E ⇒ E+$\underline{\mathrm{E}}$

⇒ E+E+$\underline{\mathrm{E}}$

⇒ E+E+id

⇒ E+id+id

⇒ id+id+id



EQUIVALENCE OF PUSHDOWN AUTOMATA AND CFG

If a grammar G is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar G. A parser can be built for the grammar G. Also, if P is a pushdown automaton, an equivalent context-free grammar G can be constructed where L(G) = L(P)

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F)

Step 1 − Convert the productions of the CFG into GNF.

Step 2 − The PDA will have only one state {q}.

Step 3 − The start symbol of CFG will be the start symbol in the PDA.

Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 − For each production in the form A → aX where a is terminal and A, X are combination of terminal and non-terminals, make a transition δ (q, a, A).

Problem

Construct a PDA from the following CFG.

G = ({S, X}, {a, b}, P, S)

where the productions are –

S → XS | ε , A → aXb | Ab | ab

Solution

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ –

δ(q, ε , S) = {(q, XS), (q, ε)}

δ(q, ε , X) = {(q, aXb), (q, Xb), (q, ab)}

δ(q, a, a) = {(q, ε)}

δ(q, 1, 1) = {(q, ε)}

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F) such that the non- terminals of the grammar G will be {Xwx | w,x ∈ Q} and the start state will be Aq0,F.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule Xwx → a Xyzb in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule Xwx → XwyXyx in grammar G.

Step 3 − For w ∈ Q, add the production rule Xww → ε in grammar G.

Various Properties of context free languages (CFL)

Context-Free Language (CFL) is a language which is generated by a context-free grammar or Type 2 grammar(according to Chomsky classification) and gets accepted by a Pushdown Automata.
Some very much important properties of a context-free language is:

Regularity- context-free languages are Non-Regular PDA language.

Closure properties :

The context-free languages are closed under some specific operation, closed means after doing that operation on a context-free language the resultant language will also be a context-free language.Some such operation are:

1. Union Operation
2. Concatenation
3. Kleene closure
4. Reversal operation
5. Homomorphism
6. Inverse Homomorphism
7. Substitution
8. init or prefix operation
9. Quotient with regular language
10. Cycle operation
11. Union with regular language
12. Intersection with regular language
13. Difference with regular language

Context free language is not closed under some specific operation, not-closed means after doing that operation on a context-free language the resultant language not remains be a context-free language anymore.

Some such operation are:

1. Intersection
2. Complement
3. Subset
4. Superset
5. Infinite Union
6. Difference, Symmetric difference (xor, Nand, nor or any other operation which get reduced to intersection and complement