AMBIGUITY IN GRAMMAR

A grammar is said to be ambiguous if there exists more than one leftmost derivatio or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

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

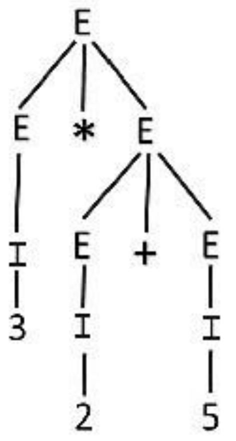
Example :

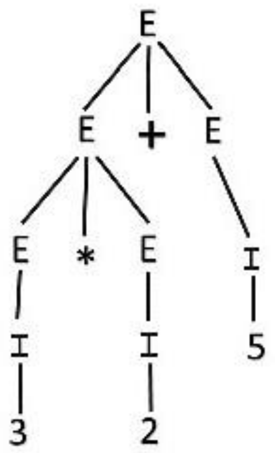
Let us consider a grammar G with the production rule

1. E → I
2. E → E + E
3. E → E * E
4. E → (E)
5. I → ε | 0 | 1 | 2 | ... | 9

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:





Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example :

Check whether the given grammar G is ambiguous or not.

1. E → E + E
2. E → E - E
3. E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. E → E + E
2. → id + E
3. → id + E - E
4. → id + id - E
5. → id + id- id

Second Leftmost derivation

1. E → E - E
2. → E + E - E
3. → id + E - E
4. → id + id - E
5. → id + id - id

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

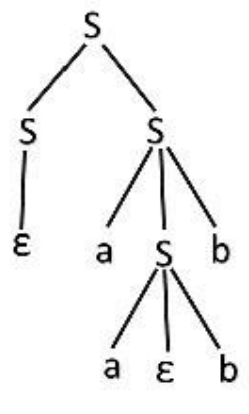
Example :

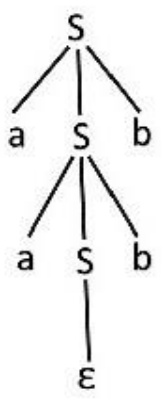
Check whether the given grammar G is ambiguous or not.

1. S → aSb | SS
2. S → ε

Solution:

For the string "aabb" the above grammar can generate two parse trees





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

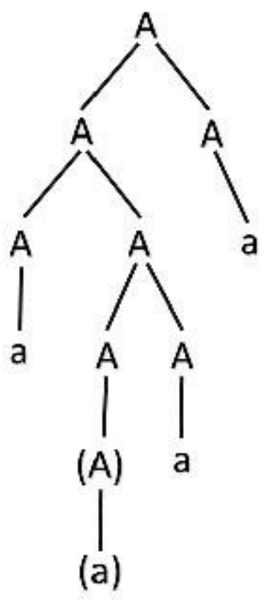
Example 4:

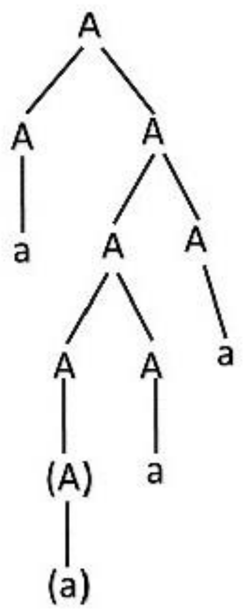
Check whether the given grammar G is ambiguous or not.

1. A → AA
2. A → (A)
3. A → a

Solution:

For the string "a(a)aa" the above grammar can generate two parse trees:





Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

CONTEXT-FREE GRAMMAR (CFG)

CFG stands for context-free grammar. It is is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

G = (V, T, P, S)

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

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

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

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

Example :

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

Solution:

As we know the regular expression for the above language is

1. r.e. = a*

Production rule for the Regular expression is as follows:

1. S → aS rule 1
2. S → ε rule 2

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

1. S
2. aS
3. aaS rule 1
4. aaaS rule 1
5. aaaaS rule 1
6. aaaaaS rule 1

7. aaaaaaS rule 1
8. aaaaaaε rule 2
9. aaaaaa

The r.e. = a* can generate a set of string {ε, a, aa, aaa, ...}. We can have a null string because S is a start symbol and rule 2 gives S → ε.

Example :

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

Solution:

The CFG can be given by,

1. Production rule (P): 2. S → 0S | 1S
3. S → ε

The rules are in the combination of 0's and 1's with the start symbol. Since (0+1)* indicates {ε, 0, 1,01, 10, 00, 11, }. In this set, ε is a string, so in the rule, we can set the rule S → ε.

Example :

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

Solution:

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

The grammar could be:

1. S → aSa rule 1
2. S → bSb rule 2
3. S → c rule 3

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

1. S → aSa
2. S → abSba from rule 2
3. S → abbSbba from rule 2
4. S → abbcbba from rule 3

Thus any of this kind of string can be derived from the given production rules.

Example 4:

Construct a CFG for the language L = $a^n b^{2n}$ where n>=1.

Solution:

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

The grammar could be:

1. S → aSbb | abb

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

1. S → aSbb
2. S → aabbbb

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing, we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be placed with production rule.

1. Leftmost Derivation:

In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.

Example:

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

Input

1. a - b + a

The leftmost derivation is:

1. E = E + E
2. E = E - E + E
3. E = a - E + E
4. E = a - b + E
5. E = a - b + a

2. Rightmost Derivation:

In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

Example

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

Input

1. a - b + a

The rightmost derivation is:

1. E = E - E
2. E = E - E + E
3. E = E - E + a
4. E = E - b + a
5. E = a - b + a

When we use the leftmost derivation or rightmost derivation, we may get the same string. This type of derivation does not affect on getting of a string.

Examples of Derivation:

Example :

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

1. S → AB | ε
2. A → aB
3. B → Sb

Solution:

Leftmost derivation:

$\boxed{}$ B

a $\boxed{¿§}$ B

A $\boxed{\varepsilon}$ bB

ab $\boxed{Sb}$

ab $\boxed{\varepsilon}$ b

abb

Rightmost derivation:

$\boxed{}$

A $\boxed{\varepsilon}$ b

$\boxed{}$

$\boxed{}$

Example :

Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

1. S ⟶ aB | bA
2. S ⟶ a | aS | bAA
3. S ⟶ b | aS | aBB

Solution:

Leftmost derivation:

1. S
2. aB S ⟶ aB
3. aaBB B ⟶ aBB

4. aabB $B \to b$
5. aabbS $B \to bS$
6. aabbaB $S \to aB$
7. aabbabS $B \to bS$
8. aabbabbA $S \to bA$
9. aabbabba $A \to a$

Rightmost derivation:

1. S
2. aB $S \to aB$
3. aaBB $B \to aBB$
4. aaBbS $B \to bS$
5. aaBbbA $S \to bA$
6. aaBbba $A \to a$
7. aabSbba $B \to bS$
8. aabbAbba $S \to bA$
9. aabbabba $A \to a$

Example :

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

1. $S \to A1B$
2. $A \to 0A \mid \varepsilon$
3. $B \to 0B \mid 1B \mid \varepsilon$

Solution:

Leftmost derivation:

1. S
2. A1B
3. 0A1B
4. 00A1B
5. 001B
6. 0010B
7. 00101B
8. 00101

Rightmost derivation:

1. S

2. A1B
3. A10B 4.
A101B 5. A101
6. 0A101
7. 00A101
8. 00101

DERIVATION TREE

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG. It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules. The derivation tree is also called a parse tree.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

A parse tree contains the following properties:

1. The root node is always a node indicating start symbols.
2. The derivation is read from left to right.
3. The leaf node is always terminal nodes.
4. The interior nodes are always the non-terminal nodes.

Example :

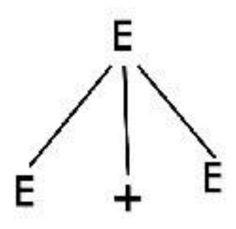
Production rules:

1. E = E + E
2. E = E * E
3. E = a | b | c

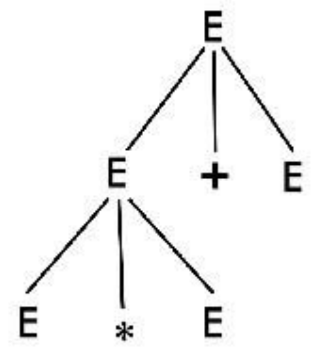
Input

1. a * b + c

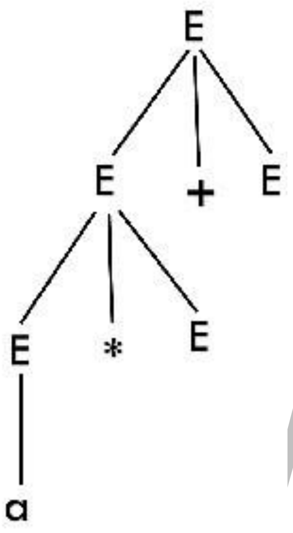
Step 1:



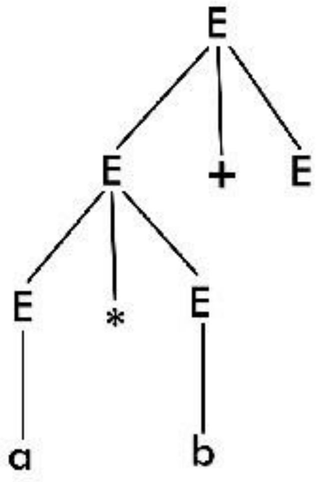
Step 2:



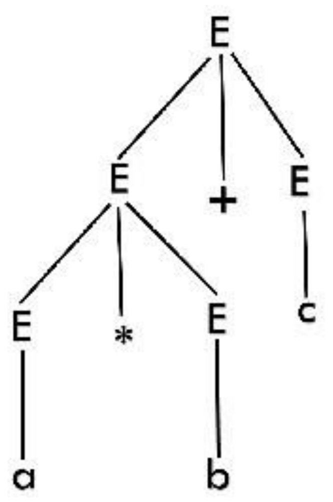
Step 2:



Step 4:



Step 5:



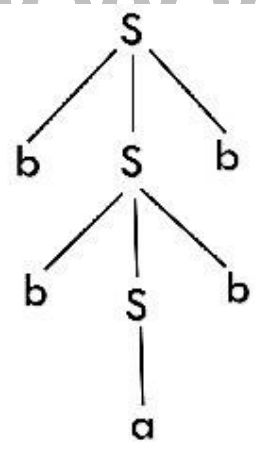
Example :

Draw a derivation tree for the string'bab" from the CFG given by

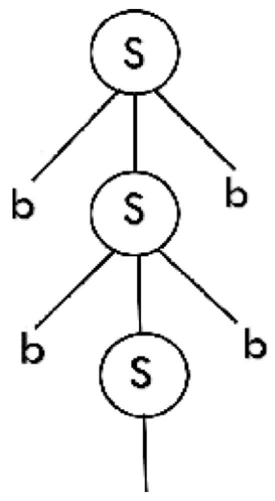
1. S → bSb | a | b

Solution:

Now, the derivation tree for the string "bbabb" is as follows:



The above tree is a derivation tree drawn for deriving a string bbabb. By simply reading the leaf nodes, we can obtain the desired string. The same tree can also be denoted by,



Example :

Construct a derivation tree for the string aabbabba for the CFG given by,

1. S → aB | bA
2. A → a | aS |bAA
3. B → b | bS |aBB

Solution:

To draw a tree, we will first try to obtain derivation for the string aabbabba

a $\boxed{\text{aBB}}$

aa $\boxed{\text{bS}}$ B

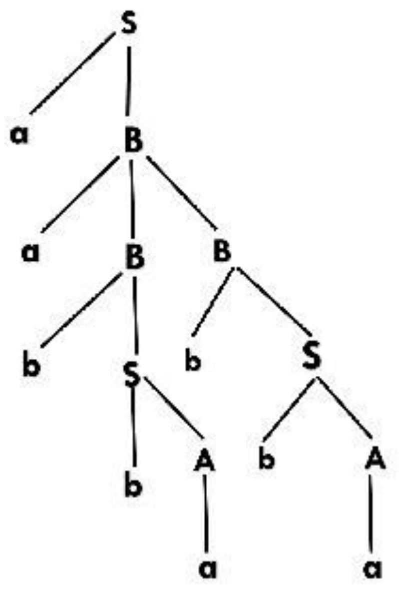
aab $\boxed{\text{bA}}$ B

aabb $\boxed{\text{a}}$ B

aabba $\boxed{\text{bS}}$

aabbab $\boxed{\text{bA}}$

aabbabb $\boxed{\text{a}}$


S
a
B
a
B
B
b
S
b
S
b
A
b
A
a
a

$\boxed{aB}$ B

a $\boxed{Sb}$ B

a bB

a BAB

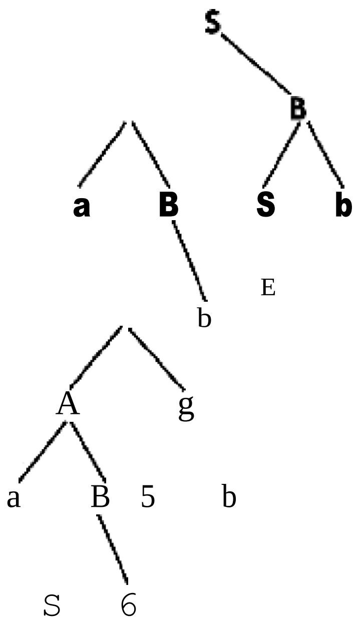
aa BbB

aa bBbB

aab bB

aab bbB

Now, the derivation tree for the string "aabbbb" is as follows:



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, q_0, 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 −

$\delta(q, \varepsilon, S) = \{(q, XS), (q, \varepsilon)\}$

$\delta(q, \varepsilon, X) = \{(q, aXb), (q, Xb), (q, ab)\}$

$\delta(q, a, a) = \{(q, \varepsilon)\}$

$\delta(q, 1, 1) = \{(q, \varepsilon)\}$

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = $(Q, \sum, S, \delta, q_0, I, F)$ such that the non- terminals of the grammar G will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

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 $X_{wx} \rightarrow a\, X_{yz}b$ in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule $X_{wx} \rightarrow X_{wy}X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

LANGUAGES OF PDA

A language can be accepted by Pushdown automata using two approaches:

1. Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by the final state can be defined as:

1. L(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ F}

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by empty stack can be defined as:

1. N(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ Q}

Equivalence of Acceptance by Final State and Empty Stack

- If L = N(P1) for some PDA P1, then there is a PDA P2 such that L = L(P2). That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language L = L (P1) for some PDA P1 then there is a PDA P2 such that L = N(P2). That means language accepted by final state PDA is also acceptable by empty stack PDA.

Example:

Construct a PDA that accepts the language L over {0, 1} by empty stack which accepts all the string of 0's and 1's in which a number of 0's are twice of number of 1's.

Solution:
There are two parts for designing this PDA:

- If 1 comes before any 0's
- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The δ can be

1. δ(q0, 1, Z) = (q0, 11, Z) Here Z represents that stack is empty
2. δ(q0, 0, 1) = (q0, ε)

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from q0 to q1. [Note that state q1 indicates that first 0 is read and still second 0 has yet to read].

Being in q1, if 1 is encountered then POP 0. Being in q1, if 0 is read then simply read that second 0 and move ahead. The δ will be:

1. δ(q0, 0, Z) = (q1, 0Z)
2. δ(q1, 0, 0) = (q1, 0)
3. δ(q1, 0, Z) =(q0, ε) (indicate that one 0 and one 1 is already read, so simply read the second 0)
4. δ(q1, 1, 0) = (q1, ε)

Now, summarize the complete PDA for given L is:

1. δ(q0, 1, Z) = (q0, 11Z)
2. δ(q0, 0, 1) = (q1, ε)
3. δ(q0, 0, Z) = (q1, 0Z)
4. δ(q1, 0, 0) = (q1, 0)
5. δ(q1, 0, Z) = (q0, ε)
6. δ(q0, ε, Z) = (q0, ε) ACCEPT state

PUSH DOWN AUTOMATA(PDA)

- Pushdown automata is a way to implement a CFG in the same 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.
- Pushdown automata is simply an NFA augmented with an "external stack memory". The addition of stack is used to provide a last-in-first-out memory manage ment capability to Pushdown automata. Pushdown automata can store an unbounded amount of information on the stack. It can access a li mited amount of information on the stack. A PDA can push an ack and pop off an element from element onto the top of the st element into thethe top of the stack. To read an elements must be popped off and are lost. stack, the to n FA. Any language which can be acceptable by FA can also be o accepts a class of language which
- A PDA is more powerful tha acceptable byeven cann ot be accepted by PDA. PDA als FA. Thus PDA is much superior to FA. more

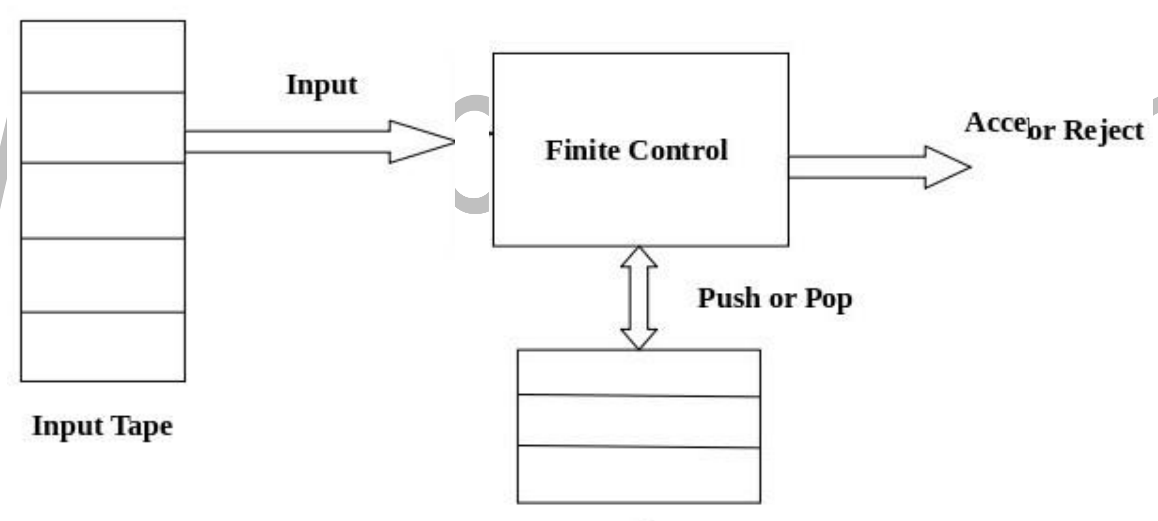


Fig: Pushdown A utomata

[Source: "J.E.Hopcroft, R.Motwani and J.D Ullman, Introduction to Automata Theory, Languages and Computations, Second Edition, Pearson Education, 2003]

PDA Components:

Input tape: The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.

Finite control: The finite control has some pointer which points the current symbol which is to be read.

Stack: The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

Formal definition of PDA:

The PDA can be defined as a collection of 7 components:

Q: the finite set of states

∑: the input set

Γ: a stack symbol which can be pushed and popped from the stack

q0: the initial state

Z: a start symbol which is in Γ.

F: a set of final states

δ: mapping function which is used for moving from current state to next state.

Instantaneous Description (ID)

ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected.

An instantaneous description is a triple (q, w, α) where:

q describes the current state.

w describes the remaining input.

α describes the stack contents, top at the left. Turnstile Notation:

⊢ sign describes the turnstile notation and represents one move.

⊢ * sign describes a sequence of moves.

For example,

(p, b, T) ⊢ (q, w, α)

In the above example, while taking a transition from state p to q, the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α.

Example 1:

Design a PDA for accepting a language $\{a^n b^{2n} \mid n>=1\}$.

Solution: In this language, n number of a's should be followed by 2n number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon
as we read 'b' then for every single 'b' only one 'a' should get popped from the stack.

The ID can be constructed as follows:

1. δ(q0, a, Z) = (q0, aaZ)
2. δ(q0, a, a) = (q0, aaa)

Now when we read b, we will change the state from q0 to q1 and start popping corresponding 'a'. Hence,

1. δ(q0, b, a) = (q1, ε)

Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state q1 only.

1. δ(q1, b, a) = (q1, ε)

After reading all b's, all the corresponding a's should get popped. Hence when we read ε as input symbol then there should be nothing in the stack. Hence the move will be:

1. δ(q1, ε, Z) = (q2, ε)

Where

PDA = ({q0, q1, q2}, {a, b}, {a, Z}, δ, q0, Z, {q2})

We can summarize the ID as: 1. δ(q0, a, Z) =

(q0, aaZ)

2. δ(q0, a, a) = (q0, aaa)
3. δ(q0, b, a) = (q1, ε)
4. δ(q1, b, a) = (q1, ε)
5. δ(q1, ε, Z) = (q2, ε)

Now we will simulate this PDA for the input string "aaabbbbbb".

1. δ(q0, aaabbbbbb, Z) ⊢ δ(q0, aabbbbbb, aaZ)
2. ⊢ δ(q0, abbbbbb, aaaaZ)
3. ⊢ δ(q0, bbbbbb, aaaaaaZ)
4. ⊢ δ(q1, bbbbb, aaaaaZ)
5. ⊢ δ(q1, bbbb, aaaaZ)
6. ⊢ δ(q1, bbb, aaaZ)
7. ⊢ δ(q1, bb, aaZ)
8. ⊢ δ(q1, b, aZ)
9. ⊢ δ(q1, ε, Z)
10. ⊢ δ(q2, ε)
11. ACCEPT