Pushdown 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 management capability to Pushdown automata. Pushdown automata can store an unbounded amount of information on the stack. It can access a limited amount of information on the stack. A PDA can push an element onto the top of the stack and pop off an element from the top of the stack. To read an element into the stack, the top elements must be popped off and are lost.
- A PDA is more powerful than FA. Any language which can be acceptable by FA can also be acceptable by PDA. PDA also accepts a class of language which even cannot be accepted by FA. Thus PDA is much more superior to FA.

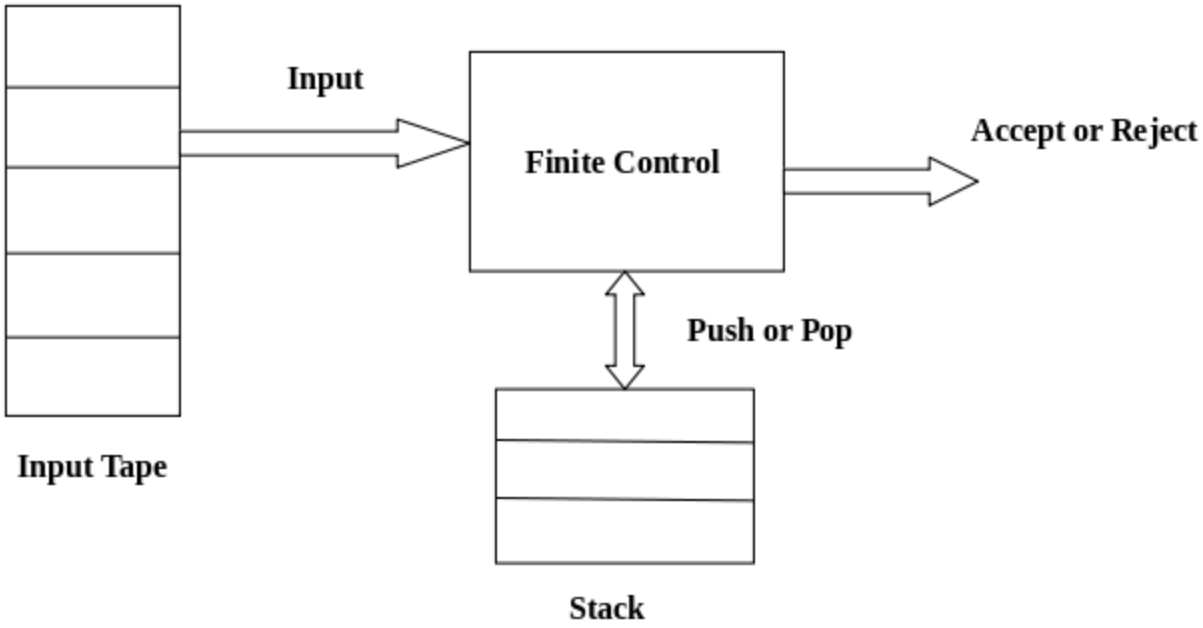


Fig: Pushdown Automata

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

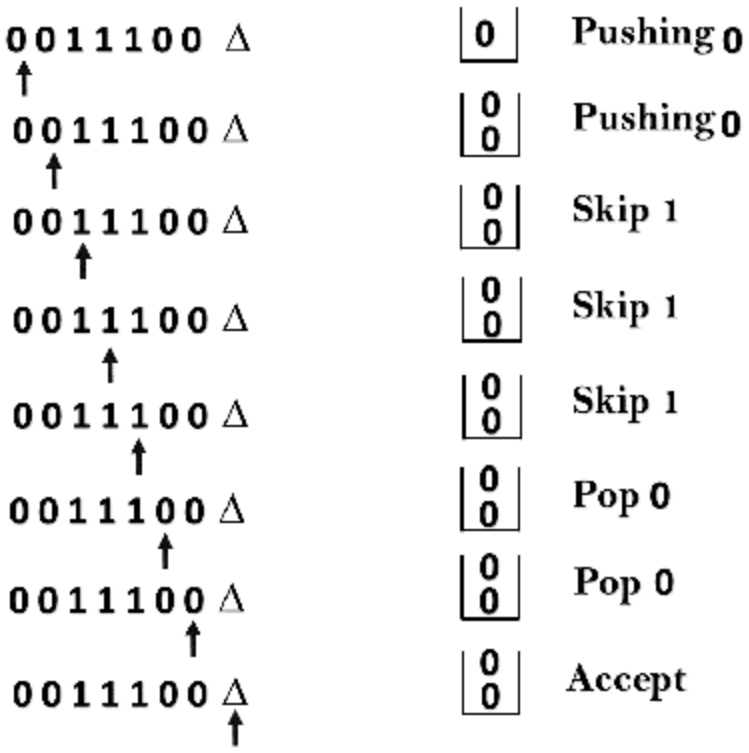
Example 2:

Design a PDA for accepting a language $\{0^n1^m0^n \mid m, n>=1\}$.

Solution: In this PDA, n number of 0's are followed by any number of 1's followed n number of 0's. Hence the logic for design of such PDA will be as follows:

Push all 0's onto the stack on encountering first 0's. Then if we read 1, just do nothing. Then read 0, and on each read of 0, pop one 0 from the stack.

For instance:



This scenario can be written in the ID form as:

1. δ(q0, 0, Z) = δ(q0, 0Z)
2. δ(q0, 0, 0) = δ(q0, 00)
3. δ(q0, 1, 0) = δ(q1, 0)
4. δ(q0, 1, 0) = δ(q1, 0)
5. δ(q1, 0, 0) = δ(q1, ε)
6. δ(q0, ε, Z) = δ(q2, Z) (ACCEPT state)

Now we will simulate this PDA for the input string "0011100".

1. δ(q0, 0011100, Z) ⊢ δ(q0, 011100, 0Z)
2. ⊢ δ(q0, 11100, 00Z)
3. ⊢ δ(q0, 1100, 00Z)
4. ⊢ δ(q1, 100, 00Z)
5. ⊢ δ(q1, 00, 00Z)
6. ⊢ δ(q1, 0, 0Z)

7. ⊢ δ(q1, ε, Z)
8. ⊢ δ(q2, Z)
9. ACCEPT

PDA Acceptance

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:

$$L(PDA) = \{w \mid (q0, w, Z) \vdash^* (p, \varepsilon, \varepsilon), q \in F\}$$

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:

$$N(PDA) = \{w \mid (q0, w, Z) \vdash^* (p, \varepsilon, \varepsilon), q \in Q\}$$

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

δ(q0, 1, Z) = (q0, 11, Z) Here Z represents that stack is empty
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:

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

Now, summarize the complete PDA for given L is:

δ(q0, 1, Z) = (q0, 11Z)
δ(q0, 0, 1) = (q1, ε)
δ(q0, 0, Z) = (q1, 0Z)
δ(q1, 0, 0) = (q1, 0)
δ(q1, 0, Z) = (q0, ε)
δ(q0, ε, Z) = (q0, ε) ACCEPT state

Non-deterministic Pushdown Automata

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

Example:

Design PDA for Palindrome strips.

Solution:

Suppose the language consists of string L = {aba, aa, bb, bab, bbabb, aabaa, ...]. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

1. $\delta(q1, a, Z) = (q1, aZ)$
2. $\delta(q0, b, Z) = (q1, bZ)$
3. $\delta(q0, a, a) = (q1, aa)$
4. $\delta(q1, a, b) = (q1, ab)$
5. $\delta(q1, a, b) = (q1, ba)$
6. $\delta(q1, b, b) = (q1, bb)$

Pushing the symbols onto the stack

7. $\delta(q1, a, a) = (q2, \varepsilon)$
8. $\delta(q1, b, b) = (q2, \varepsilon)$
9. $\delta(q2, a, a) = (q2, \varepsilon)$
10. $\delta(q2, b, b) = (q2, \varepsilon)$
11. $\delta(q2, \varepsilon, Z) = (q2, \varepsilon)$

Popping the symbols on reading the same kind of symbol

Simulation of abaaba

1. δ(q1, abaaba, Z) Apply rule 1

2. ⊢ δ(q1, baaba, aZ) Apply rule 5
3. ⊢ δ(q1, aaba, baZ) Apply rule 4
4. ⊢ δ(q1, aba, abaZ) Apply rule 7
5. ⊢ δ(q2, ba, baZ) Apply rule 8
6. ⊢ δ(q2, a, aZ) Apply rule 7
7. ⊢ δ(q2, ε, Z) Apply rule 11
8. ⊢ δ(q2, ε) Accept

CFG to PDA Conversion

The first symbol on R.H.S. production must be a terminal symbol. The following steps are used to obtain PDA from CFG is:

Step 1: Convert the given productions of CFG into GNF.

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

Step 3: The initial symbol of CFG will be the initial symbol in the PDA.

Step 4: For non-terminal symbol, add the following rule:

δ(q, ε, A) = (q, α)

Where the production rule is A → α

Step 5: For each terminal symbols, add the following rule:

δ(q, a, a) = (q, ε) **for** every terminal symbol

Example 1:

Convert the following grammar to a PDA that accepts the same language.

S → 0S1 | A
A → 1A0 | S | ε

Solution:

The CFG can be first simplified by eliminating unit productions:

S → 0S1 | 1S0 | ε

Now we will convert this CFG to GNF:

S → 0SX | 1SY | ε
X → 1
Y → 0

The PDA can be:

R1: δ(q, ε, S) = {(q, 0SX) | (q, 1SY) | (q, ε)}
R2: δ(q, ε, X) = {(q, 1)}
R3: δ(q, ε, Y) = {(q, 0)}
R4: δ(q, 0, 0) = {(q, ε)}
R5: δ(q, 1, 1) = {(q, ε)}

Example 2:

Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.

S → 0BB
B → 0S | 1S | 0

Solution:

The PDA can be given as:

A = {(q), (0, 1), (S, B, 0, 1), δ, q, S, ?}

The production rule δ can be:

R1: δ(q, ε, S) = {(q, 0BB)}
R2: δ(q, ε, B) = {(q, 0S) | (q, 1S) | (q, 0)}
R3: δ(q, 0, 0) = {(q, ε)}
R4: δ(q, 1, 1) = {(q, ε)}

Testing 0104 i.e. 010000 against PDA:

δ(q, 010000, S) ⊢ δ(q, 010000, 0BB)
⊢ δ(q, 10000, BB) R1
⊢ δ(q, 10000,1SB) R3
⊢ δ(q, 0000, SB) R2
⊢ δ(q, 0000, 0BBB) R1
⊢ δ(q, 000, BBB) R3
⊢ δ(q, 000, 0BB) R2

⊢ δ(q, 00, BB) R3
⊢ δ(q, 00, 0B) R2
⊢ δ(q, 0, B) R3
⊢ δ(q, 0, 0) R2
⊢ δ(q, ε) R3
ACCEPT

Thus 0100 is accepted by the PDA.

Example 3:

Draw a PDA for the CFG given below:

S → aSb
S → a | b | ε

Solution:

The PDA can be given as:

P = {(q), (a, b), (S, a, b, z0), δ, q, z0, q}

The mapping function δ will be:

R1: δ(q, ε, S) = {(q, aSb)}
R2: δ(q, ε, S) = {(q, a) | (q, b) | (q, ε)}
R3: δ(q, a, a) = {(q, ε)}
R4: δ(q, b, b) = {(q, ε)}
R5: δ(q, ε, z0) = {(q, ε)}

Simulation: Consider the string aaabb

δ(q, εaaabb, S) ⊢ δ(q, aaabb, aSb) R3
⊢ δ(q, εaabb, Sb) R1
⊢ δ(q, aabb, aSbb) R3
⊢ δ(q, εabb, Sbb) R2
⊢ δ(q, abb, abb) R3
⊢ δ(q, bb, bb) R4
⊢ δ(q, b, b) R4
⊢ δ(q, ε, z0) R5
⊢ δ(q, ε)
ACCEPT

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

Decision Properties:

1. Test for Membership: Decidable.
2. Test for Emptiness: Decidable
3. Test for finiteness: Decidable

Rest the decision properties are undecidable in context-free language.

Deterministic property :

The context-free language can be:

1. DCFL-Deterministic (which can be recognized by deterministic pushdown automata) context-free language
2. NDCFL-Non-deterministic (can’t be recognized by DPDA but NPDA) context free language.