PDA (Push down automata)

↳ Machine models for the CFL class of Languages.

↳ More powerful than DFA, NFA because CFL ⊇ RL (regular languages)

PDA ⟶ Finite state machine (FSM) + an additional Stack

This gives more features, power and so PDA is more powerful than DFA/NFA which are just simple FSM.

DFA representation ⟶ $(Q, \Sigma, \delta, q_0, F)$

PDA ⟶ $(Q, \Sigma, \Gamma, \perp, \delta, q_0, F)$

where $Q, \Sigma, \delta, q_0, F$ have usual meanings as they are for the FSM Part in the PDA. and δ will be slightly different.

→ Additionally Γ (tau) is another finite alphabet set that we use in the stack. All stack elements are single character elements from Γ.

→ and $\perp$ → bottom marker character of your stack.

Starting Configuration :-

At the starting of your machine, you will be in the start state $q_0 \in Q$, your stack will contain a single bottom marker element $\perp$.

→ Transition function (δ) :-

in a DFA, δ is a function of current state and the next input character you are reading from your input string $x \in L$.

from there you transition to different state

$\delta(q, a) = q'$

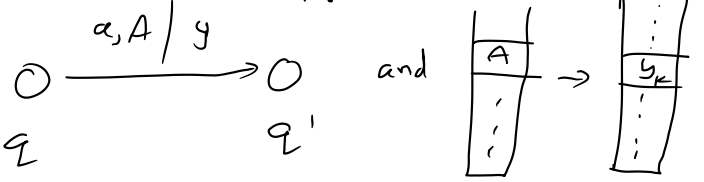
In a PDA,

δ will utilize your stack and updates the stack at every transition.

δ is a function of your current state q, next input char 'a', top most stack element 'A'.

→ From there, transitions to a new state, and pops off the current top element of your stack and pushes some series of characters $\gamma = \underset{\text{top}}{\gamma_1} \cdots \underset{\text{bottom}}{\gamma_k}$

$$\delta(q, a, A) = (q', \gamma)$$



Final acceptance criteria :-

we have 2 different types of PDA,

(i) Acceptance by final state | (ii) Acceptance by empty stack.

↓ ↓

After reading the whole string, making transitions, updating stack if you end in a final state.

Accepted. $x \in L$,

After reading the whole string, our stack will be emptied.

↓

Our job is to construct our machine in such a way that for either type of machines, our string 'x' after reading every character ends in a final state.

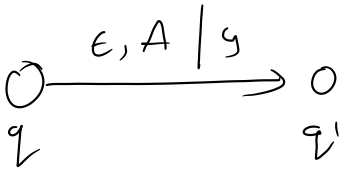
(or) if you constructed (ii) type machine, our stack will be empty at the end.

For every PDA, we have to mention which type of PDA we are constructing. Type (i) (or) Type (ii).

Note:

(1) PDA is a non-deterministic machine. You can specify multiple transitions for same 'a', different 'A', (or) same (a, A) ----.

(2) ϵ-transitions are also allowed, they look like this.



$\delta(q, a, A) = (q', y)$

(3) Since PDA is non deterministic, similar to NFA. we can have multiple paths of transitioning. we accept $x \in L$ if there exists atleast one path which after consuming whole string x will end satisfying our acceptance criteria for that machine.

(4) In NFAs, we are still able to keep track of all the states we can exist in for reading till certain part of x. and we did not write all possible paths every time.

→ But in PDA, all paths are different and we cannot simply keep track of all states we can exist in at every time because the stack configuration can/will be different for every path.

(5) If in some path you end in a situation where your stack becomes empty before you finish reading your string 'x'.

→ That path is terminated because no top stack element in your stack to make a next transition.

→ Similarly if there is no transition for your current state (q, a, A) → terminate path.

(6) The stack can only be empty when the whole string is read and acceptance criteria is either success/failure (decided REL or not) in either type machine. (otherwise, Path terminated)

We saw the starting situation, transitions, final acceptance criteria

Now lets see some PDA Construction examples. (from next page, just brief discussion below)

→ We already looked at DFA Construction and multiple examples. We know the kind of things we can do using a DFA.

→ Now, we can additionally do more with a stack. like do some counting.

ex:- we saw in CFG that $L = \{a^n b^n \mid \forall n \geq 0\}$ is not a regular language because we cannot count a's and then see b's using a DFA.

→ But we can construct a PDA that uses stack smartly to represent this language. in the stack.

→ We can push, pop stuff (characters from T) anytime based on δ rules. So we can do more. examples in next page.

① $L_1 = \{a^n b^n \mid \forall n \geq 0\}$ → we know this is not a regular language.

PDA for L_1 (acceptance by empty stack PDA)

→ the idea is,

→ for reading a's, stay in current state, keep pushing a's into stack.

→ when you start reading b's, move to different state and for every 'b' you are reading, stay in this 2nd state and replace top most stack element 'a' with empty string.

→ So a → Push to stack

b → Pop one 'a' from stack.

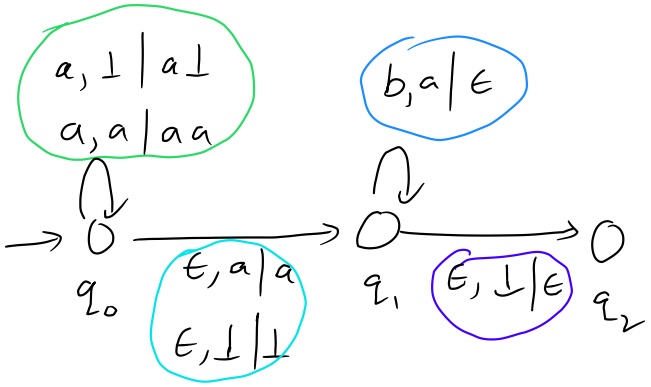
So initially we start with our stack containing only

and after reading whole input string we will again be left with 

from there do one ϵ-transition to Pop off that $\perp$ and empty the stack. → string is read, empty stack at end. ✓

(or) just do a ϵ-transition to a different final state when $\perp$ is exposed and this way you can construct a type (i) PDA.

Type -2 PDA for L_1



in q_0, if you see 'a' char just push it into stack

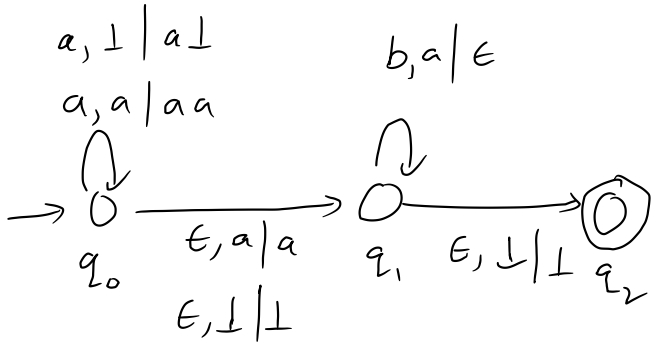
after reading all a's use this non-deterministic ϵ-transition to move to q_1.

for all b's you read pop off one 'a' at top of stack.

at end ⊥ will be exposed. Pop it off.

Type-1 PDA for L_1

→ Simple change. at end instead of popping ⊥. just make q_2 as final state. (also you can pop if you want).



Note

(1) in acceptance by final state PDA, you only make sure you can end in a final state. (doesn't matter what stack contains at that point)

(2) in acceptance by empty stack PDA, you only make sure after reading whole string 'w'. your stack is empty.

(dont care if there is no final state)

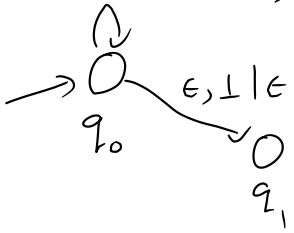
(2) $L_2 = \{ x \mid x$ Contains equal no. of a's and b's $\}$

→ in previous example we had to make sure we read all a's before b's. thats why before reading b's we moved to a different state.

→ Here we dont care about order, we can do every thing using a single state.

→ our stack will contain a $\perp$, and a's, b's count we keep track using +, - characters.

a, +| ++ , b,+| ϵ
a, ⊥ |+⊥ b,⊥|-⊥
a,- |ϵ b,- |--



PDA by empty stack.

$Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{+, -, \perp\}$

→ how we are using stack here.

→ initially start with  every time.

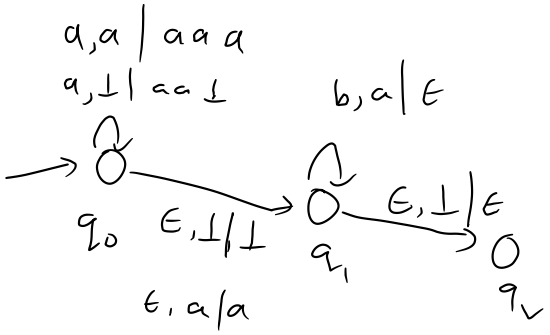
→ when you read 'a', if you have '-' on top, pop it off
if you have + (or) ⊥ at top, push a '+'

→ when you read 'b', if you have '+' on top, pop it off
if you have - (or) ⊥ at top, push a '-'

→ at end if you have same no. of a's, b's, all will be popped, you will be left with  again.

$L_3 = \{a^n b^{2n} \mid \forall n \geq 0\}$

→ similar to L_1, but every time push 2 a's instead of one into the stack.



PDA by empty stack.

$L_4 = \{x \mid \#a(x) = 2 \cdot \#b(x)\}$ → x contains twice as many a's as b's.

→ L_4 is not a content free language. no PDA.

$L_5 = \{a^{k_1 n} b^{k_2 n} \mid n \geq 0,\ k_1, k_2 \in \mathbb{N}\}$

→ idea: for reading an 'a' → push k_2 no. of a's to stack.

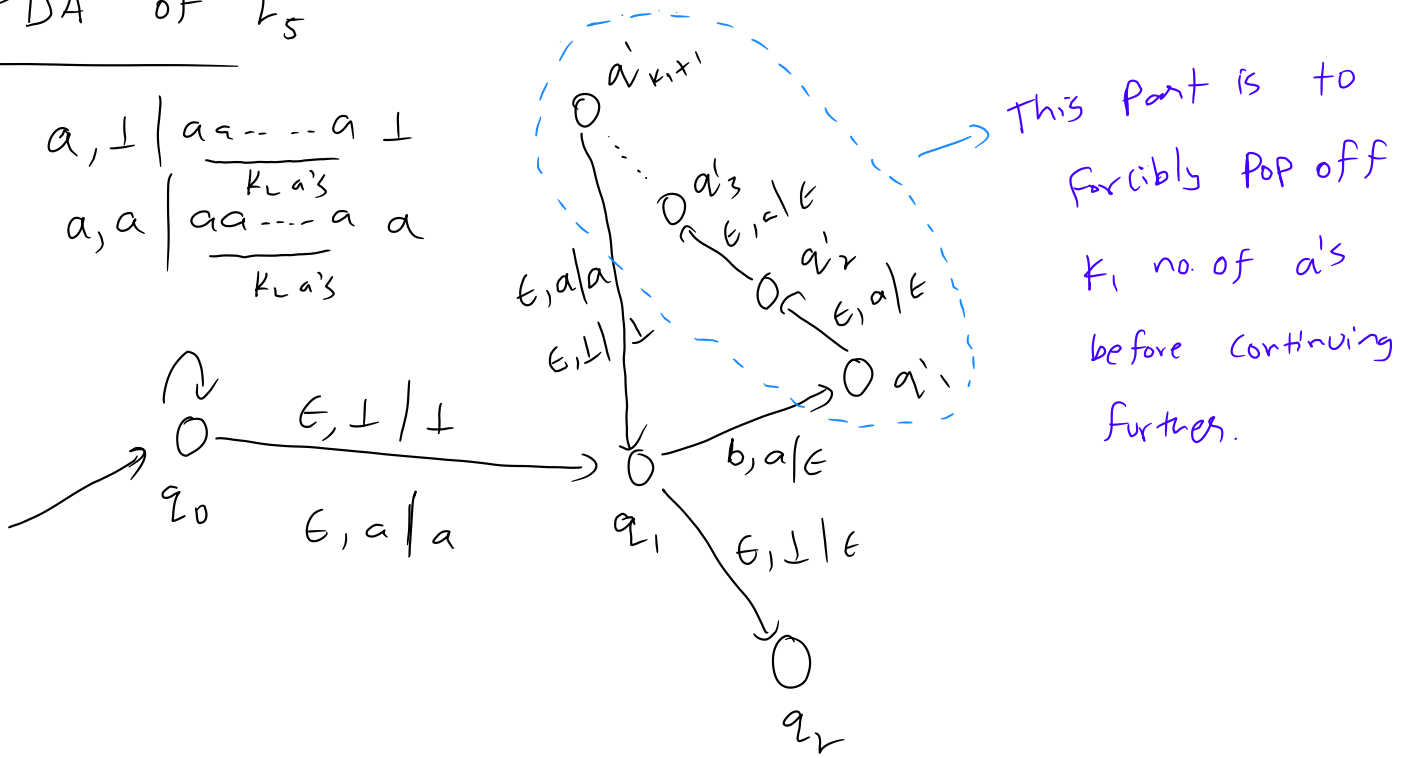
for reading a 'b' → pop k_1 no. of a's from stack.

so we are pushing $k_2 (k_1 n)$ no. of a's

popping $k_1 (k_2 n)$ no. of b's

we will be left with ⊥ at the end.

PDA of L_5


a, ⊥ | aa---a ⊥
K₁ a's
a, a | aa---a a
K₁ a's
q0
q1
q2
ε, ⊥ | ⊥
ε, a | a
b, a | ε
ε, ⊥ | ε
a'K1+1
a'3
a'2
a'1
ε, a | ε
ε, a | a
ε, ⊥ | ⊥
This part is to forcibly pop off K1 no. of a's before continuing further.