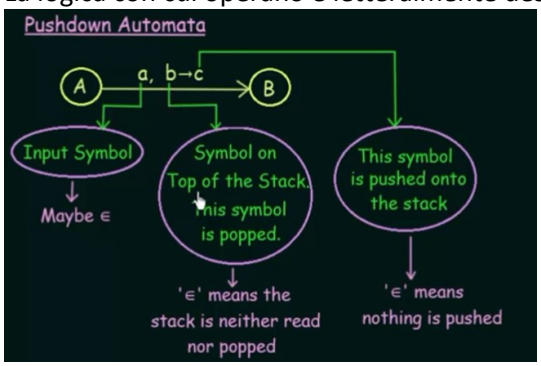
Pushdown automata/PDA/automi a pila

Essi hanno memoria infinita e dispongono di operazioni di *push* e *pop* dallo stack, che ragiona con la leftmost-derivation (la parte più a sinistra) che corrisponde alla cima dello stack.
La logica con cui operano è letteralmente descrivibile con questa immagine:

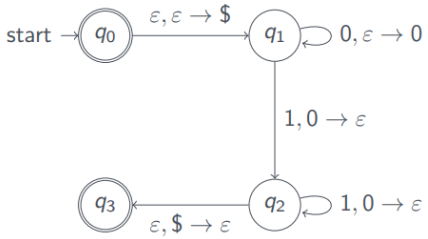


Partendo da una tabella di transizione:

- $P = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{0, \$\}, \delta, q_0, \{q_0, q_3\})$
- con δ descritta dalla tabella:

Input:	0			1			ε		
Pila:	0	$	ε	0	$	ε	0	$	ε
q_0									$\{(q_1, \$)\}$
q_1			$\{(q_1, 0)\}$	$\{(q_2, \varepsilon)\}$					
q_2				$\{(q_2, \varepsilon)\}$				$\{(q_3, \varepsilon)\}$	
q_3									

- $P = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{0, \$\}, \delta, q_0, \{q_0, q_3\})$
- con δ descritta dal diagramma di transizione:



Un PDA accetta un linguaggio se:

Data una parola w, un PDA accetta la parola se:

- possiamo scrivere $w = w_1 w_2 \dots w_m$ dove $w_i \in \Sigma \cup \{\varepsilon\}$
- esistono una sequenza di stati $r_0, r_1, \dots, r_m \in Q$ e
- una sequenza di stringhe $s_0, s_1, s_2, \dots, s_m \in \Gamma^*$

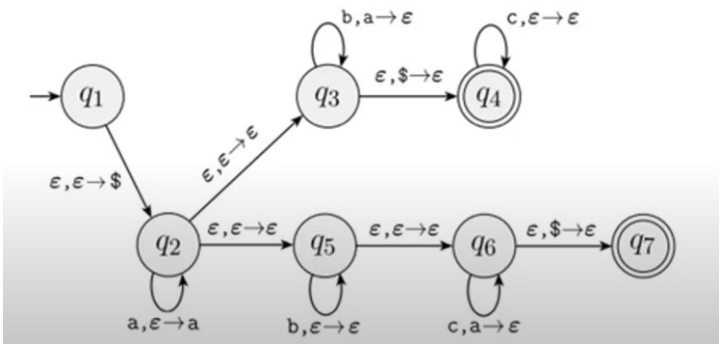
tali che

1. $r_0 = q_0$ e $s_0 = \varepsilon$ (inizia dallo stato iniziale e pila vuota)
2. per ogni $i = 0, \dots, m-1$, $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ con $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$ (rispetta la funzione di transizione)
3. $r_m \in F$ (la computazione termina in uno stato finale)

Similmente, sappiamo che un PDA accetta la parola *per pila vuota* se:

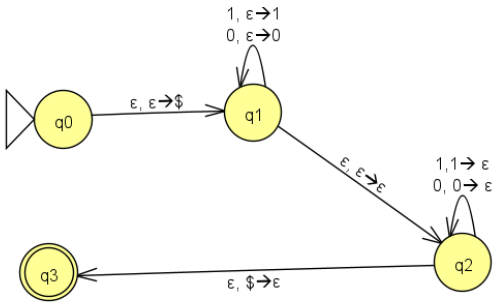
- consuma tutto l'input
- termina con la pila vuota (quindi rimuovendo anche lo stato iniziale)
- Vediamo poi due esempi pratici:

1 Costruisci un PDA per il linguaggio $\{a^i b^j c^k \mid i = j \text{ oppure } j = k\}$



2 Costruisci un PDA per il linguaggio $\{ww^R \mid w \in \{0,1\}^*\}$, dove w^R indica la parola w scritta al contrario

L'osservazione è che, essendo la stringa palindroma, il primo passaggio butta lo stato iniziale che sarà l'ultimo ad essere rimosso, lo stato successivo andrà in se stesso con entrambi 0/1 (permesso perché è un automa non deterministico). A questo punto l'automa controlla idealmente se quello che c'era nello stack è uguale al resto della stringa, poppando a/b se sono nella cima dello stack e poi togliamo lo stato iniziale, accettando la stringa.
Un esempio di stringa accettata: *abba*, in quanto subito *b* è cima dello stack e viene tolto, assieme ad *a*.
Per lo stesso motivo, non viene accettata *abab*, in quanto non si rispetta l'ordine di pop.
Segue il PDA qui a lato:



Conversione CFG-PDA

L'idea è:

Idea.

- Se L è context free, allora esiste una CFG G che lo genera
- Mostriamo come trasformare G in un PDA equivalente P
- P è fatto in modo da simulare i passi di derivazione di G
- P accetta w se esiste una derivazione di w in G

1. Inserisci il simbolo marcatore $ e la variabile iniziale S sulla pila
2. Ripeti i seguenti passi:
   1. Se la cima della pila è la variabile A: scegli una regola $A \to u$ e scrivi u sulla pila
   2. Se la cima della pila è un terminale a: leggi il prossimo simbolo di input.
      - se sono uguali, procedi
      - se sono diversi, rifiuta
   3. Se la cima della pila è $: vai nello stato accettante

Prendiamo il seguente esempio:

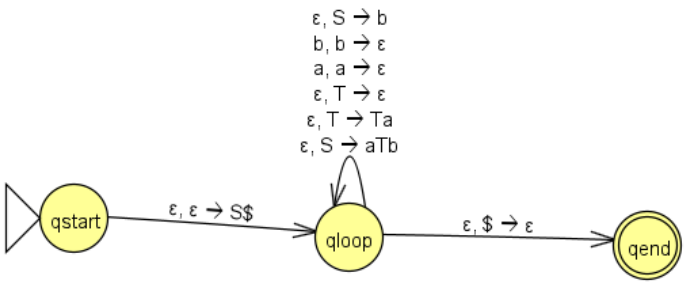
Trasformiamo la seguente CFG in PDA:

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$

Dobbiamo metterci 3 stati:

q_{start}, q_{loop}, q_{end}

e successivamente diciamo "se c'è simbolo di input nella pila, poi lo rimuovo":



L'idea quindi è di seguire la leftmost derivation, quindi avremmo:

- ε, S → aTb (prima cosa fatta)
- l'altra transizione possibile di S
- le due transizione possibili di T
- a e b (in pop) perché simboli terminali

Essendo grammatica CF, si scelgono liberamente le regole da applicare.

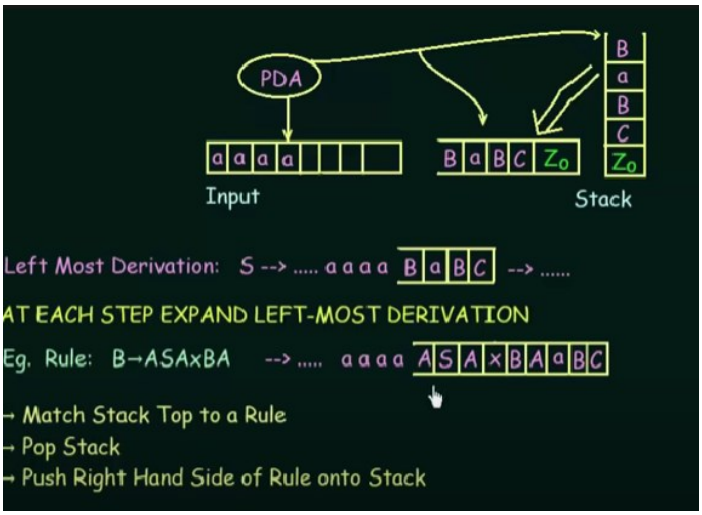
Per esempio scegliamo di attuare una derivazione
S → aTb → aTab → aab

~~S~~
$

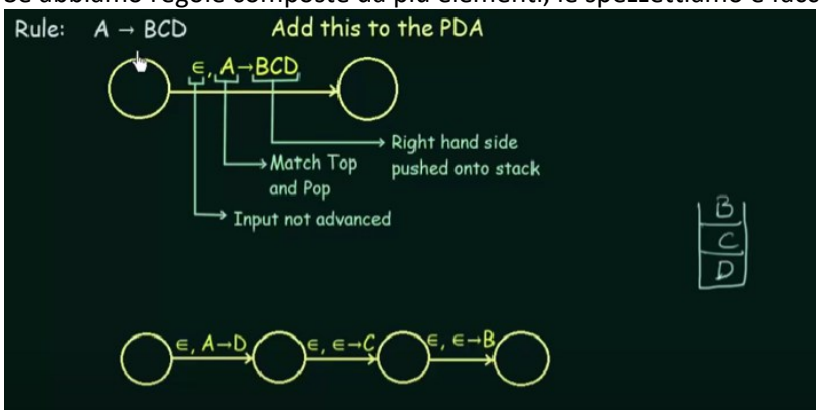
~~a~~
T
b
$

~~T~~
~~a~~
~~b~~
~~$~~

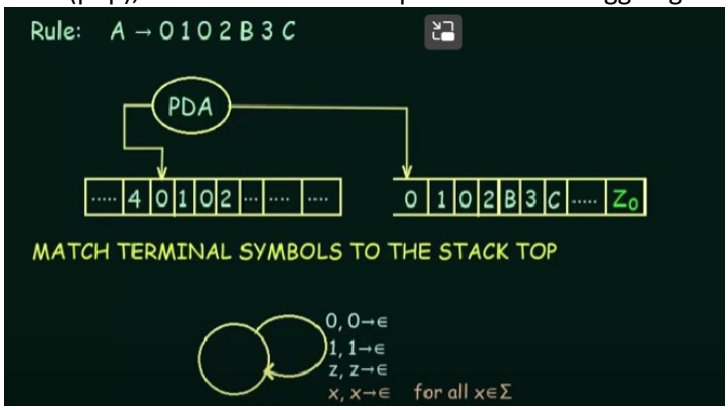
In questo caso, quindi, seguiamo la leftmost derivation, buttiamo dentro (push) tutti i simboli nell'ordine detto dopodiché, sapendo che rimarranno nella pila solo "a" e "b", andremo ad eseguirlo il pop se presenti.
In sintesi abbiamo una roba del tipo:



Quindi vogliamo partire sempre da sinistra con le derivazioni, facendo in modo di avere almeno una regola di quel tipo sulla cima dello stack.
Se abbiamo regole composte da più elementi, le spezzettiamo e facciamo *push* in maniera ordinata:



Successivamente, con questa logica, l'elemento che ho buttato dentro per ultimo sarà il primo ad esseree tolto (pop), facendo avanzare l'input finché non raggiungiamo la fine dell'input:



Conversione PDA-CFG

Si parte da una grammatica che:

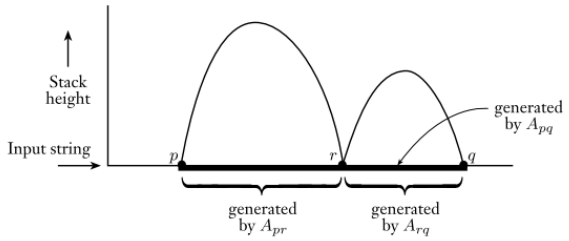
- una variabile A_{pq} per ogni coppia di stati p, q di P
- A_{pq} genera tutte le stringhe che portano da p con pila vuota a q con pila vuota

semplificando la pila tale che:

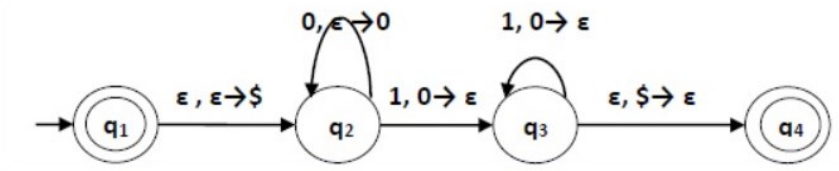
1. Ha un unico stato accettante q_f
2. Svuota la pila prima di accettare
3. Ogni transizione inserisce un unico simbolo sulla pila (push) oppure elimina un simbolo dalla pila (pop), ma non fa entrambe le cose contemporaneamente

sapendo quindi che a livello teorico abbiamo due casi:

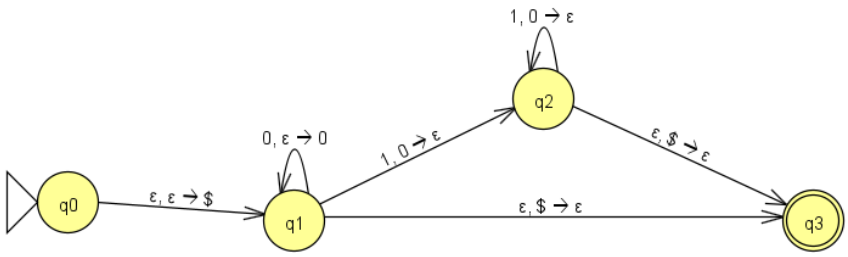
- Per andare da p con pila vuota a q con pila vuota:
  - la prima mossa deve essere un push
  - l'ultima mossa deve essere un pop
- Ci sono due casi:
  1. il simbolo inserito all'inizio viene eliminato alla fine
  2. oppure no:



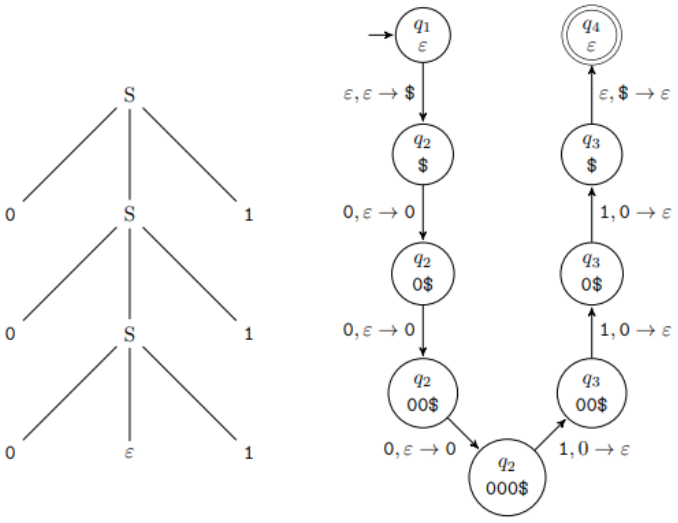
Trasformiamo il PDA per il linguaggio $\{0^n1^n \mid n>=0\}$ in grammatica:



In poche parole mette la transizione che da q2 va a q4 inserendo ε, $ → ε perché potrebbe non esserci nessun simbolo



Quindi potremmo avere concretamente una situazione di questo tipo:

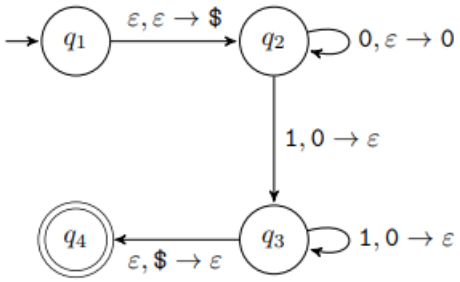


CFG derivation PDA run

La grammatica prodotta è (riporto l'automa identico al nostro, ma comunque per chiarire col discorso lettere a fianco della CFG):

- si parte considerando i 4 stati che vanno ad ε
- si inseriscono poi tutti gli stati che vanno, per combinazione, tutti gli uni con gli altri (letteralmente è una proprietà commutativa, l'immagine sotto chiarisce)
- nel caso di A_{23} abbiamo l'unione degli stati uscenti
- come ultimo abbiamo la regola dello stato finale

$$
\begin{aligned}
&A_{11} \to \varepsilon \\
&A_{22} \to \varepsilon \\
&A_{33} \to \varepsilon \\
&A_{44} \to \varepsilon \\
&A_{11} \to A_{11}A_{11} \mid A_{12}A_{21} \mid A_{13}A_{31} \mid A_{14}A_{41} \\
&A_{12} \to A_{11}A_{12} \mid A_{12}A_{22} \mid A_{13}A_{32} \mid A_{14}A_{42} \\
&A_{13} \to A_{11}A_{13} \mid A_{12}A_{23} \mid A_{13}A_{33} \mid A_{14}A_{43} \\
&\quad \dots \\
&A_{42} \to A_{41}A_{12} \mid A_{42}A_{22} \mid A_{43}A_{32} \mid A_{44}A_{42} \\
&A_{43} \to A_{41}A_{13} \mid A_{42}A_{23} \mid A_{43}A_{33} \mid A_{44}A_{43} \\
&A_{44} \to A_{41}A_{14} \mid A_{42}A_{24} \mid A_{43}A_{34} \mid A_{44}A_{44} \\
&A_{23} \to \mathtt{0}A_{22}\mathtt{1} \mid \mathtt{0}A_{23}\mathtt{1} \\
&A_{14} \to \varepsilon A_{23} \varepsilon \\
&S \to A_{14}
\end{aligned}
$$



Altro esempio utile (dei pochi che ho trovato in giro su questa cosa):

variables: $A_{00}, A_{11}, A_{22}, A_{01}, A_{02}, A_{12}$

start variable: A_{02}

productions:

$A_{02} \to A_{01}A_{12}$

$A_{01} \to A_{01}A_{11}$

$A_{12} \to A_{11}A_{12}$

$A_{11} \to A_{11}A_{11}$

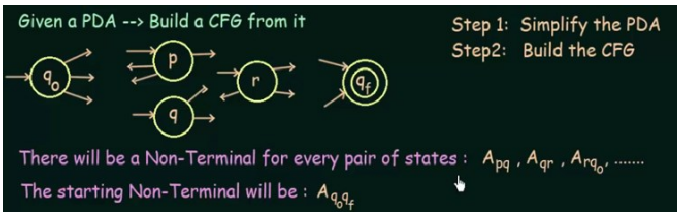
$A_{11} \to 0A_{11}1$

$A_{11} \to 1A_{11}0$

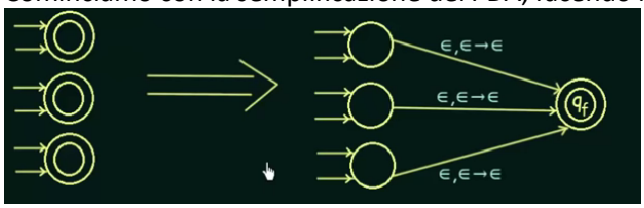
$A_{02} \to A_{11}$

$A_{00} \to \varepsilon,\ A_{11} \to \varepsilon,$
$A_{22} \to \varepsilon$

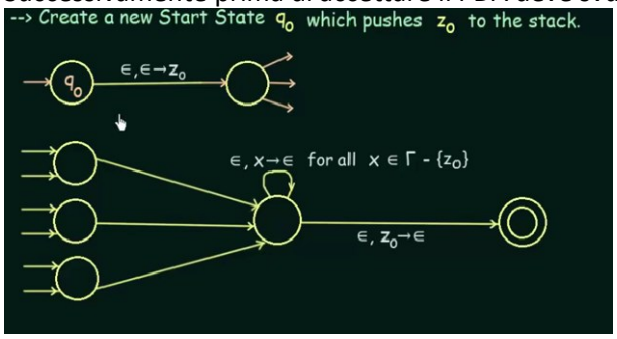
In sintesi quindi:



Cominciamo con la semplificazione del PDA, facendo in modo di avere un solo stato finale:



Successivamente prima di accettare il PDA deve svuotare il suo stack:



Attenzione che le transizioni o fanno push oppure pop, ma non devono fare entrambe le cose. Come si vede, non possiamo avere un caso in cui non ci sia pop/push, quindi piazziamo un nuovo simbolo che permetta di fare questa operazione: