

Automa a stati finiti deterministico (DFA)

Un Automa a Stati Finiti Deterministico (DFA) è una quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

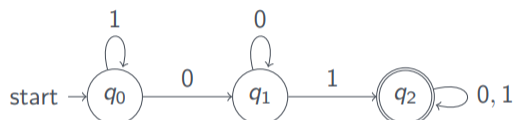
- Q è un insieme finito di **stati**
- Σ è un **alfabeto finito** (= simboli in input)
- δ è una **funzione di transizione** $(q, a) \mapsto q'$
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è un insieme di **stati finali**

Generalmente un automa si dice *accetti* un linguaggio se, nella descrizione del *diagramma di transizione* (il disegno classico coi cerchietti e le frecce dell'automa), una stringa/sottostringa/insieme di stringhe, i corrispondenti stati iniziali arrivano nei corrispondenti stati finali.

Concretamente si rappresenta così:

Esempio: costruiamo un automa A che accetta il linguaggio delle stringhe con 01 come sottostringa

- L'automa come **diagramma di transizione**:



- L'automa come **tabella di transizione**:

	0	1
→ q_0	q_1	q_0
q_1	q_1	q_2
* q_2	q_2	q_2

Tutti i linguaggi accettati dagli automi si definiscono linguaggi regolari.

Automa a stati finiti non deterministico (NFA)

Quando l'automa può scegliere liberamente in quali stati andare (trovandosi quindi contemporaneamente in più stati), si ha il *non determinismo*. Formalmente:

Un Automa a Stati Finiti Non Deterministico (NFA) è una quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q è un insieme finito di **stati**
- Σ è un **alfabeto finito** (= simboli in input)
- δ è una **funzione di transizione** che prende in input (q, a) e restituisce un **sottoinsieme di Q**
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è un insieme di **stati finali**

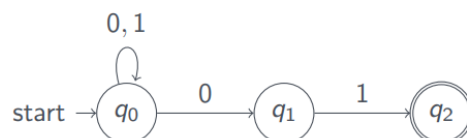
La computazione di un NFA ha due condizioni:

- Data una parola $w = w_1 w_2 \dots w_n$, una **computazione** di un NFA A con input w è una sequenza di stati $r_0 r_1 \dots r_n$ che rispetta **due condizioni**:
 - 1 $r_0 = q_0$ (inizia dallo stato iniziale)
 - 2 $\delta(r_i, w_{i+1}) = r_{i+1}$ per ogni $i = 0, \dots, n-1$ (rispetta la funzione di transizione)
- Diciamo che una computazione **accetta** la parola w se:
 - 3 $r_n \in F$ (la computazione **termina in uno stato finale**)
- A causa del nondeterminismo, **ci può essere più di una computazione** per ogni parola!

Confronto riassuntivo DFA/NFA

Deterministic Finite Automata	Non-Deterministic Finite Automata
Each transition leads to exactly one state called as deterministic	A transition leads to a subset of states i.e. some transitions can be non-deterministic.
Accepts input if the last state is in Final	Accepts input if one of the last states is in Final.
Backtracking is allowed in DFA.	Backtracking is not always possible.
Requires more space.	Requires less space.
Empty string transitions are not seen in DFA.	Permits empty string transition.
For a given state, on a given input we reach a deterministic and unique state.	For a given state, on a given input we reach more than one state.
DFA is a subset of NFA.	Need to convert NFA to DFA in the design of a compiler.
$\delta : Q \times \Sigma \rightarrow Q$ For example - $\delta(q_0, a) = \{q_1\}$	$\delta : Q \times \Sigma \rightarrow 2^Q$ For example - $\delta(q_0, a) = \{q_1, q_2\}$
DFA is more difficult to construct.	NFA is easier to construct.
DFA is understood as one machine.	NFA is understood as multiple small machines computing at the same time.

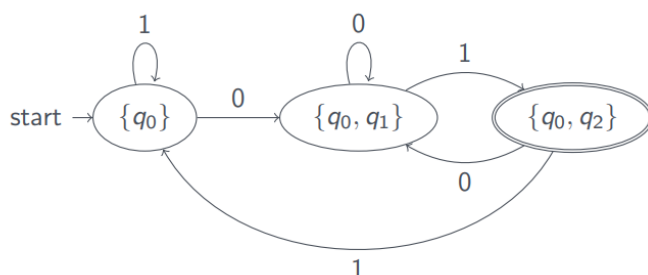
Essi sono in grado di riconoscere lo stesso linguaggio per mezzo della costruzione per sottoinsiemi (dove la funzione di transizione percorre tutte le possibili strade, si ha un insieme di stati e si ha almeno uno stato finale). Si costruisce la tabella di transizione per i soli stati raggiunti dallo stato iniziale e si opera correttamente la conversione.



Costruiamo δ_D per l'NFA qui sopra:

	0	1
\emptyset	\emptyset	\emptyset
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	\emptyset	$\{q_2\}$
$*\{q_2\}$	\emptyset	\emptyset
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	\emptyset	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

La tabella di transizione per D ci permette di ottenere il **diagramma di transizione**



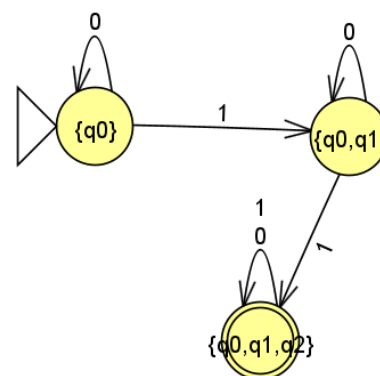
Normalmente, infatti, nella costruzione per sottoinsiemi omettiamo gli stati non raggiungibili, quindi normalmente quelli non raggiunti dallo stato iniziale e dalle successive unioni di stati.

Conversione da DFA ad NFA

- 1 Determinare il DFA equivalente all'NFA con la seguente tabella di transizione:

	0	1
→ q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_1\}$	$\{q_0, q_2\}$
* q_2	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$

- 2 Qual è il linguaggio accettato dall'automa?



- 1) Qui sopra a destra ho rappresentato l'automa DFA, segue la sua tabella di riferimento (in questo caso completa di tutti i possibili stati, di solito si rappresentano solo gli stati collegati dallo stato iniziale):

Q_d	0	1
Insieme vuoto	Insieme vuoto	Insieme vuoto
$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_1\}$	$\{q_1\}$	$\{q_0, q_2\}$
* $\{q_2\}$	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$
* $\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$
* $\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
* $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$

- 2) Accetta come linguaggio tutte le stringhe che contengono almeno due 1

Nota: l'automa si ottiene, guardando la tabella, per unione delle precedenti. Parto da q_0 e interpreto la prima riga. Sono in $\{q_0, q_1\}$ e da lì faccio l'unione dei risultati di q_0 e q_1 (q_0 va a 0, q_1 va a 0, quindi unisco e q_0, q_1 andrà a 0 su sé stesso). Poi vado verso q_0, q_1, q_2 : questo perché q_0 va ad 1 verso q_0, q_1 mentre q_1 va ad 1 con q_0, q_2 . Unisco i due risultati ed ottengo esattamente q_0, q_1, q_2 . A questo punto, unisco q_0, q_1, q_2 (avendo q_0 che va a q_0 per 0, q_1 che va a q_1 per 0 e q_1 che va a 0 per q_1, q_2) e i loro risultati portano

proprio a q_0, q_1, q_2 . Ad 1 si nota che ci sta già una transizione verso q_0, q_1, q_2 quindi non serve metterne altre e sarà quella finale.

Automi con ϵ -transizioni (ϵ -NFA)

Per poter normalmente operare la conversione qui sopra mostrata, utilizziamo degli automi con transizioni vuote, quindi gli ϵ -NFA.

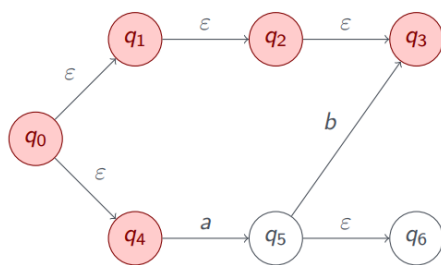
Un Automa a Stati Finiti Non Deterministico con ϵ -transizioni (ϵ -NFA) è una quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

dove:

- Q, Σ, q_0, F sono definiti come al solito
- δ è una **funzione di transizione** che prende in input:
 - uno stato in Q
 - un simbolo nell'alfabeto $\Sigma \cup \{\epsilon\}$
 e restituisce un sottoinsieme di Q

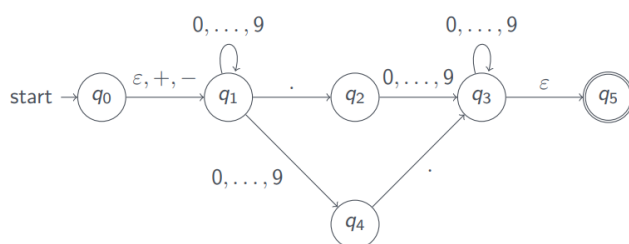
In essi si va a calcolare induttivamente la ϵ -chiusura (tutti gli stati con ϵ -transizioni raggiunti anche dallo stato iniziale), applicando anche la costruzione a sottoinsiemi per operare la normale conversione da ϵ -NFA a DFA.



$$ECLOSE(q_0) = \{q_0, q_1, q_4, q_2, q_3\}$$

E poi esempio di conversione DFA a ϵ -NFA:

Costruiamo un DFA D equivalente all' ϵ -NFA E che riconosce i numeri decimali:

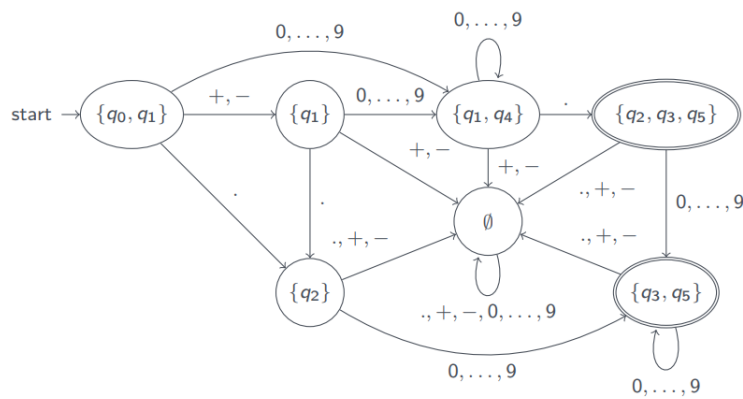


- Come prima cosa costruiamo la ϵ -chiusura di ogni stato:

$$\begin{array}{ll} ECLOSE(q_0) = \{q_0, q_1\} & ECLOSE(q_1) = \{q_1\} \\ ECLOSE(q_2) = \{q_2\} & ECLOSE(q_3) = \{q_3, q_5\} \\ ECLOSE(q_4) = \{q_4\} & ECLOSE(q_5) = \{q_5\} \end{array}$$

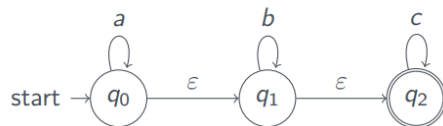
- Lo **stato iniziale** di D è $\{q_0, q_1\}$

■ Applicando le regole otteniamo il **diagramma di transizione**:



oppure:

- 1 Costruiamo un ε -NFA che riconosce le parole costituite da zero o più a , seguite da zero o più b , seguite da zero o più c



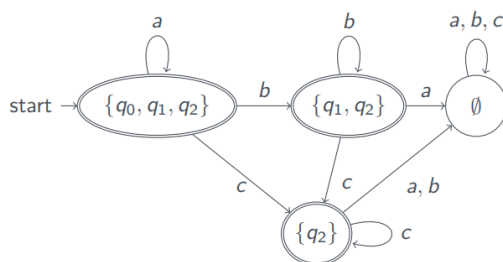
- 2 Calcolare la ε -chiusura di ogni stato

$$\text{ECLOSE}(q_0) = \{q_0, q_1, q_2\}$$

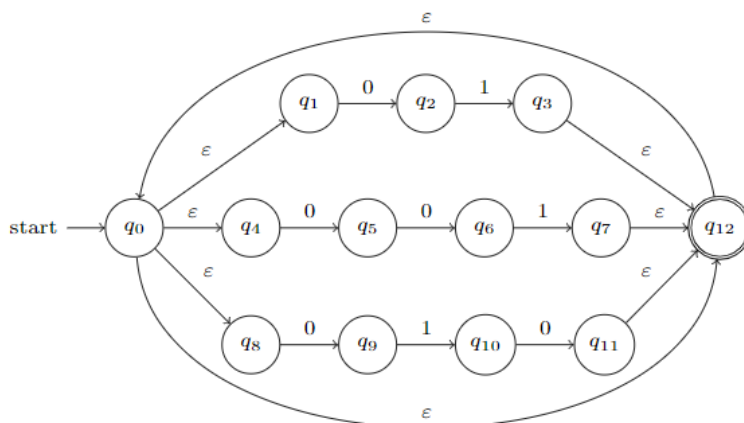
$$\text{ECLOSE}(q_1) = \{q_1, q_2\}$$

$$\text{ECLOSE}(q_2) = \{q_2\}$$

- 3 Convertire l' ε -NFA in DFA



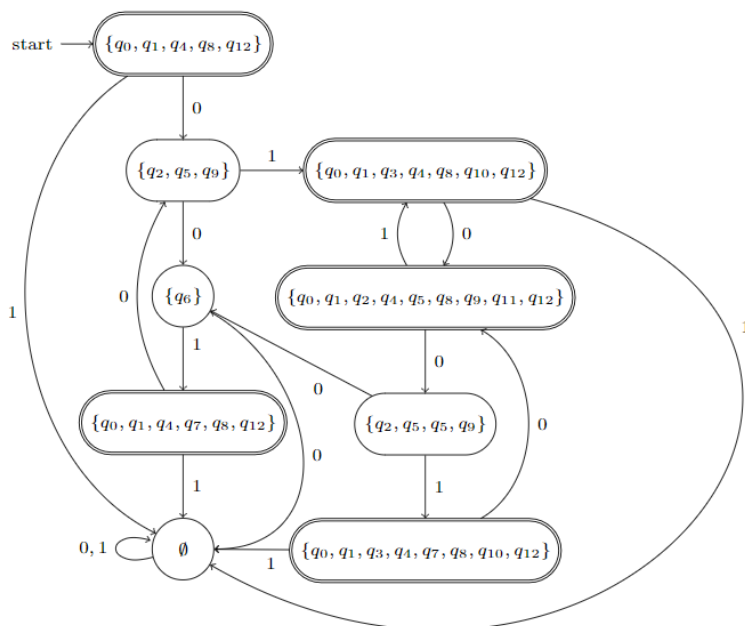
Esempio completo in un contesto complesso:



(b) Trasformare l' ϵ -NFA ottenuto al punto precedente in un DFA.

Considerando la ϵ -chiusura come stato iniziale: $q_0, q_1, q_4, q_8, q_{12}$

q_i	0	1
$\rightarrow^* \{q_0, q_1, q_4, q_8, q_{12}\}$	$\{q_2, q_5, q_9\}$	\emptyset
$\{q_2, q_5, q_9\}$	$\{q_6\}$	$\{q_0, q_1, q_3, q_4, q_8, q_{10}, q_{12}\}$
$\{q_6\}$	\emptyset	$\{q_0, q_1, q_4, q_7, q_{10}, q_{12}\}$
$^* \{q_0, q_1, q_3, q_4, q_8, q_{10}, q_{12}\}$	$\{q_0, q_1, q_2, q_4, q_5, q_8, q_9, q_{11}, q_{12}\}$	\emptyset
$^* \{q_0, q_1, q_4, q_7, q_8, q_{12}\}$	$\{q_2, q_5, q_9\}$	\emptyset
$^* \{q_0, q_1, q_2, q_4, q_5, q_8, q_9, q_{11}, q_{12}\}$	$\{q_2, q_5, q_6, q_9\}$	$\{q_6\}$
$\{q_2, q_5, q_6, q_9\}$	$\{q_0, q_1, q_3, q_4, q_7, q_8, q_{10}, q_{12}\}$	$\{q_6\}$
$^* \{q_0, q_1, q_3, q_4, q_7, q_8, q_{10}, q_{12}\}$	$\{q_0, q_1, q_2, q_4, q_5, q_8, q_9, q_{11}, q_{12}\}$	\emptyset



Come sempre si ha l'inclusione della closure nel caso degli stati q_3, q_7, q_{11} e con pazienza si ricava tutto.

Operazioni chiuse degli automi

Tutte le seguenti operazioni sono chiuse per i linguaggi regolari:

Se L e M sono linguaggi regolari, allora anche i seguenti linguaggi sono regolari:

- Unione: $L \cup M$
- Intersezione: $L \cap M$
- Concatenazione: $L.M$
- Complemento: \bar{L}
- Chiusura di Kleene: L^*

Molto utile capire come eseguire la dimostrazione che un linguaggio sia regolare (si costruisce con un automa seguendo implicitamente questa struttura):

Se L e M sono regolari, allora anche $L \cap M$ è un linguaggio regolare.

Dimostrazione. Sia L il linguaggio di

$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$

e M il linguaggio di

$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

Possiamo assumere che entrambi gli automi siano **deterministici**.
Costruiremo un automa che simula A_L e A_M in parallelo, e accetta se e solo se sia A_L che A_M accettano.

Dimostrazione (continua).

Se A_L va dallo stato p allo stato s leggendo a , e A_M va dallo stato q allo stato t leggendo a , allora $A_{L \cap M}$ andrà dallo stato (p, q) allo stato (s, t) leggendo a .

Formalmente

$$A_{L \cap M} = (Q_L \times Q_M, \Sigma, \delta_{L \cap M}, (q_L, q_M), F_L \times F_M),$$

dove

$$\delta_{L \cap M}((p, q), a) = (\delta_L(p, a), \delta_M(q, a))$$

Espressioni regolari

Un automa riconosce linguaggi regolari; essi vengono scritti tramite espressioni regolari.

Le **Espressioni Regolari** sono costruite utilizzando

- un insieme di **costanti** di base:
 - ϵ per la stringa vuota
 - \emptyset per il linguaggio vuoto
 - a, b, \dots per i simboli $a, b, \dots \in \Sigma$
- collegati da **operatori**:
 - $+$ per l'unione
 - \cdot per la concatenazione
 - $*$ per la chiusura di Kleene
- raggruppati usando le **parentesi**:
 - (\quad)

Con un ordine di applicazione:

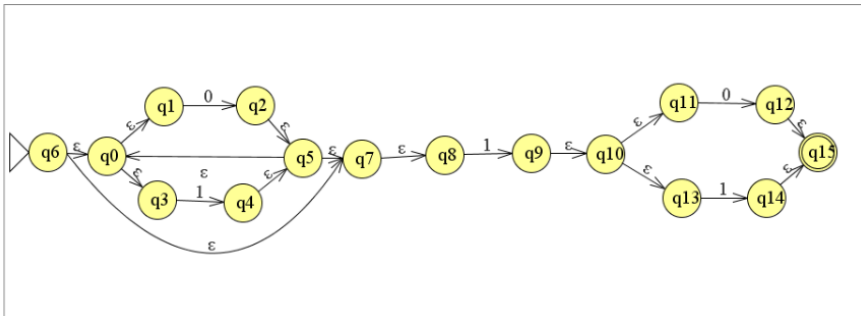
Come per le espressioni aritmetiche, anche per le espressioni regolari ci sono delle **regole di precedenza** degli operatori:

- 1 Chiusura di Kleene
- 2 Concatenazione (punto)
- 3 Unione (+)

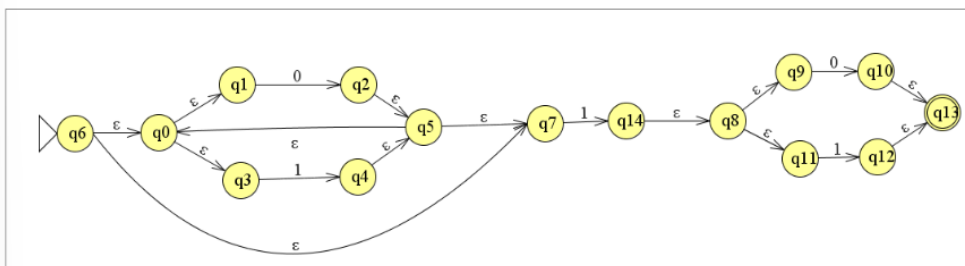
Conversione da RE a ϵ -NFA partendo dagli automi sopra descritti.

Esempi pratici:

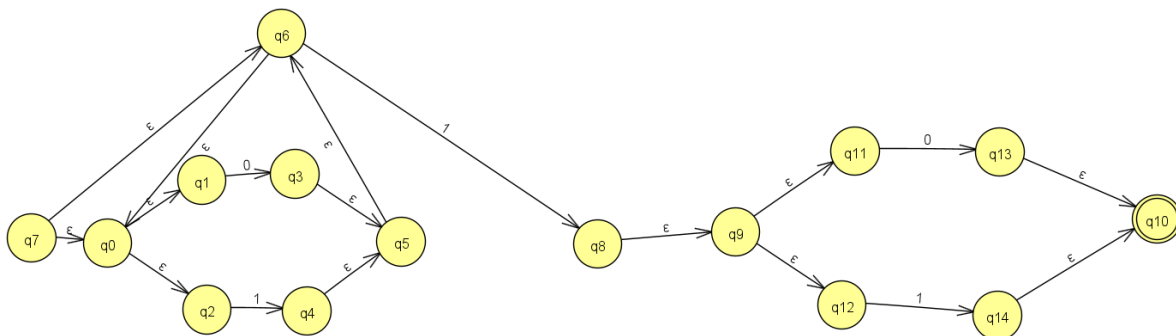
Trasformare $(0 + 1)^* 1(0 + 1)$ in ϵ -NFA



Trasformiamo $(0+1)^* 1(0+1)$ in ϵ -NFA

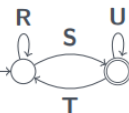


Trasformazione dell'espressione $(0 + 1)^* 1(0 + 1)$ in ϵ -NFA




Conversione da FA ad RE, ottenibile per mezzo della eliminazione degli stati.

- 1 l'automata deve avere un **unico stato finale**
 - se c'è più di uno stato finale, crea un nuovo stato finale q_f con ϵ -transizioni provenienti dai vecchi stati finali
- 2 **collassa le transizioni** tra la stessa coppia di stati
- 3 elimina tutti gli stati **tranne lo stato iniziale e lo stato finale**

- 4 se $q_f \neq q_0$ l'automata finale è
 

che è equivalente a $(R + SU^*T)^*SU^*$

- 5 se $q_f = q_0$ l'automata finale è
 
- che è equivalente a R^*

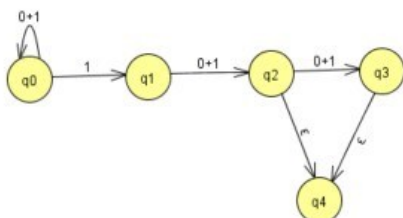
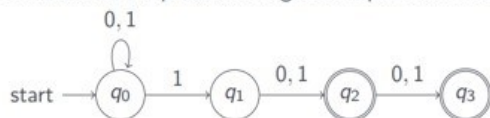
Si ricordi inoltre che:

- se lo stato iniziale presenta più stati entranti verso lo stato iniziale occorre crearne uno nuovo iniziale che non ha archi entranti, aggiungendo una ϵ -transizione verso il nuovo stato iniziale da parte del vecchio stato iniziale;
- se l'automa presenta più stati finali oppure presenta più transizioni uscenti dallo stato finale, si convertono tutti gli stati finali in stati non-finali, creando un nuovo stato finale e aggiungendo una ϵ -transizione verso il nuovo stato finale da parte dei vecchio stati finali.

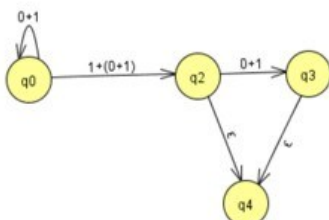
Un esempio completo:

Esercizi

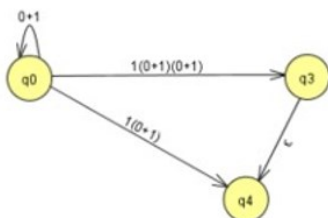
1 Costruiamo l'espressione regolare equivalente al seguente NFA:



Elimino q1 PRE: q0 SUCC: q2 $1(0+1)$



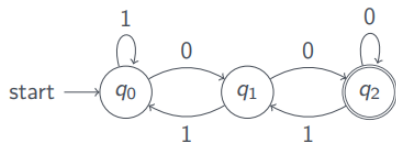
Elimino q2 PRE: q0 SUCC: q3 $1(0+1)(0+1)$
 PRE: q0 SUCC: q4 $1(0+1)$



Elimino q3 PRE: q0 SUCC: q4 $1(0+1)(0+1)$

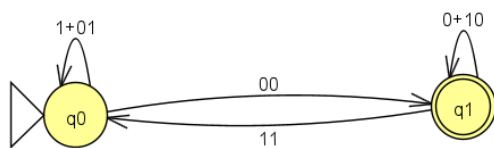


2 Costruiamo l'espressione regolare equivalente al seguente NFA:



	PRE	SUCC
Elimino q1	q0	q0 1+01
	q0	q2 00
	q2	q2 0+10
	q2	q0 11

Segue:



ER Finale: $((1+01)+00(0+10)^*11)^*00(0+10)^*$

Pumping Lemma

Come detto, definiamo linguaggio regolare quello accettabile da un automa.

Esistono però linguaggi in cui alcuni esiste di solito almeno un loop tale per cui, aumentando il numero di stringhe su di esso, il linguaggio si “sbilancia” a favore di una delle parti della stringa, avendo quindi una rappresentazione diversa da quella di partenza, dimostrando che il linguaggio così ottenuto non è regolare.

Viene infatti enunciato il pumping lemma, con le seguenti proprietà:

Sia L un **linguaggio regolare**. Allora

- esiste una **lunghezza** $k \geq 0$ tale che
- ogni parola $w \in L$ di lunghezza $|w| \geq k$
- può essere **spezzata** in $w = xyz$ tale che:
 - 1 $y \neq \epsilon$ (il secondo pezzo è non vuoto)
 - 2 $|xy| \leq k$ (i primi due pezzi sono lunghi al max k)
 - 3 $\forall i \geq 0, xy^iz \in L$ (possiamo “pompare” y rimanendo in L)

In forma sintetica le condizioni saranno sempre: $y \neq \epsilon$, $|xy| \leq k$ e $xy^iz \in L$

La dimostrazione avviene sempre per contraddizione o per assurdo, tipo così:

- 1 Sia L_{ab} il linguaggio delle stringhe sull'alfabeto $\{a, b\}$ dove il numero di a è uguale al numero di b . L_{ab} è regolare?

No, L_{ab} non è regolare:

- supponiamo per assurdo che lo sia
- sia k la lunghezza data dal Pumping Lemma
- consideriamo la parola $w = a^k b^k$
- sia $w = xyz$ una suddivisione di w tale che $y \neq \epsilon$ e $|xy| \leq k$:

$$w = \underbrace{aaa \dots a}_x \underbrace{a \dots a}_y \underbrace{ab \dots bb}_z$$
- poiché $|xy| \leq k$, le stringhe x e y sono fatte solo di a
- per il Pumping lemma, anche $xy^2z \in L_{ab}$, ma contiene più a che $b \Rightarrow$ assurdo

2 Il linguaggio $L_{rev} = \{ww^R : w \in \{a, b\}^*\}$ è regolare?

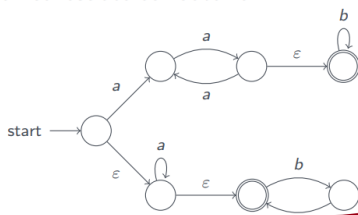
No, L_{rev} non è regolare:

- supponiamo per assurdo che lo sia
- sia k la lunghezza data dal Pumping Lemma
- consideriamo la parola $w = a^k b b a^k$
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$:
 $w = \underbrace{aaa \dots a}_x \underbrace{a}_{y} \underbrace{abb aaa \dots aaa}_z$
- poiché $|xy| \leq k$, le stringhe x e y sono fatte solo di a
- per il Pumping lemma, anche $xy^0z = xz \in L_{rev}$, ma non la posso spezzare in $ww^R \Rightarrow$ **assurdo**

3 Il linguaggio $L_{nm} = \{a^n b^m : n \text{ è dispari oppure } m \text{ è pari}\}$ è regolare?

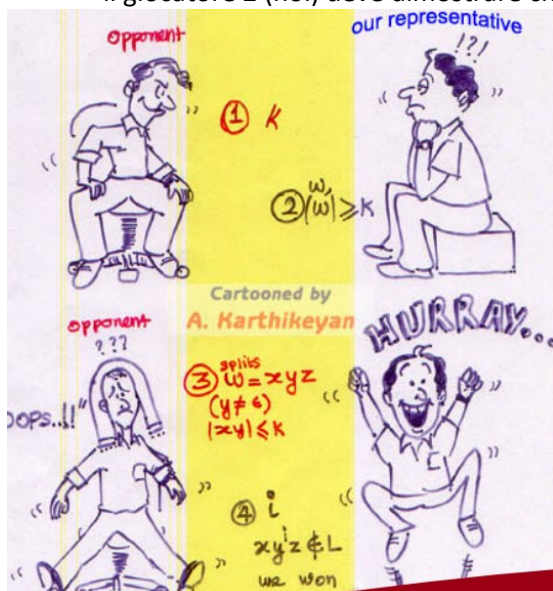
Sì, L_{nm} è regolare:

- è rappresentato dall'espressione regolare $a(aa)^*b^* + a^*(bb)^*$
- e riconosciuto dall'automa



Possiamo quindi vedere anche il gioco del pumping lemma, in cui:

- il giocatore 1 deve dimostrare che il linguaggio è regolare
- il giocatore 2 (noi) deve dimostrare che il linguaggio non è regolare



- L'avversario sceglie la lunghezza k
- Noi scegliamo una parola w
- L'avversario spezza w in xyz
- Noi scegliamo i tale che $xy^i z \notin L$
- allora **abbiamo vinto**

Esempi concreti pumping lemma

7. Sia $\Sigma = \{0, 1\}$, e considerate il linguaggio

$$D = \{w \mid w \text{ contiene un ugual numero di occorrenze di } 01 \text{ e di } 10\}$$

Mostrare che D è un linguaggio regolare.

Per assurdo come sempre assumiamo L sia regolare.

Quindi dovremmo avere sempre: $y \neq \emptyset$, $xy \leq p$ e $xy^iz \in L$

Consideriamo come parola $w = (01)^n(10)^n$

Sappiamo quindi che esisteranno due esponenti, $k > 0$, $j > 0$ tale che $(01)^k(10)^j$ entrambi minori di k , con $x = \epsilon$.

Chiaro è, a queste condizioni, che per qualsiasi i tale da pompare y , avremo $(01)^{k+i}(10)^{n-j-k}$ tale che il numero di occorrenze di 01 ed 10 si pareggia per ogni possibile pumping length.

Quindi il linguaggio è sempre regolare.

$$L = \{0^n 1^m 2^n \mid n, m \geq 0\}$$

Per assurdo come sempre assumiamo L sia regolare.

Quindi dovremmo avere sempre: $y \neq \emptyset$, $xy \leq p$ e $xy^iz \in L$

Scegliamo come possibile parola $w = 0^k 1^k 2^k$

Si può vedere quindi che i pumping su 1 portano il linguaggio ad essere sbilanciato, infatti già da qui avremmo un numero di esponenti $2p+1 > p$. Infatti seguendo la classica scia di dimostrazione, si noterebbe che una possibile stringa z avrebbe ad esempio $2^{p-(2n+1)}$ che a seguito di pumping sarebbe chiaramente sbilanciata. Pertanto il linguaggio non è regolare.

$$L = \{0^k 10^k \mid k \geq 0\}$$

Per assurdo come sempre assumiamo L sia regolare.

Quindi dovremmo avere sempre: $y \neq \emptyset$, $xy \leq p$ e $xy^iz \in L$

Scegliamo come possibile parola $w = 0^n 10^n$

Il numero di 0 è chiaramente sbilanciato rispetto agli 1, avendo una lunghezza $2n > p$ e si ragiona ugualmente a prima (z che ragiona per sottrazione di "p" dal resto degli esponenti).

A seguito di pumping si avrebbe chiaro sbilanciamento; pertanto il linguaggio non è regolare.

2 Pumping Lemma

Dimostrare che i seguenti linguaggi non sono regolari

1. $\{0^n 1^m 0^n \mid m, n \geq 0\}$
2. $\{w \in \{0, 1\}^* \mid w \text{ e' palindroma}\}$
3. $\{w \in \{0, 1\}^* \mid w \text{ non e' palindroma}\}$
4. $\{0^{pq} \mid p, q \in \mathbb{N}, p > 1, q > 1\}$
5. $\{0^n 1^m 0^{m+n} \mid n, m \in \mathbb{N}\}$
6. $\{0^n 0^{2^n} \mid n, m \in \mathbb{N}\}$

1)

1) Lo scopo: dimostrare che non è regolare

A è regolare $\rightarrow \exists k$ costante di pumping

2) Scelta parola w , $|w| \geq k$

$$n = k$$

$$w = 0^n 1^m 0^k$$

La parola deve essere almeno k , per regola, ma può essere anche oltre.

3) Suddivisione in xyz

$$w = xyz = 0 \dots 0 | 1^n 0^k$$

$$xy \quad z$$

$$|y| > 0 \quad 0^b 0^z, a > 0$$

4) Scelta di i

$$xy^i z \quad \exists A \quad \forall i$$

Caso di esempio:

$$i=0$$

$$xz = 0^b 1^m 0^k$$

$$000 \quad 1111000$$

$$xy \quad z$$

Quindi posso scegliere in xy, la y, dato che so che blocco è, ad esempio quello evidenziato.

OCCHIO: Non scrivere xyz, cioè $xy^i z$ all'esame

2)

1) Per assurdo, esiste l

2) scelta w

$$w = 0^k 1^k 0^k$$

3) scelte xyz

$$w = 0 \dots 0 \quad 1^k 0^k$$

$$xy \quad z$$

$$|y| = p, p > 0$$

$$|x| = k - p$$

$$w = 0^{k-p} 1^h 0^k$$

$$xy^i z \quad i=0 \quad 0^{k-p} 1^h 0^k$$

3)

Ragioniamo con il complementare (indicato da me per praticità $\sim A$)

Quindi

$$L(\sim A) \rightarrow L(A)$$

$$L(A) \text{ reg} \rightarrow L(\sim A) \text{ reg}$$

Non è regolare quindi, dato che il complemento è anch'esso non regolare

4)

Per assurdo ammettiamo che esista L.

Scegliamo w, $|w| \geq k$, $q=k$, $|xy| < k$, $z = 0^{k-p}$

$$w = 0^{k-p} 0^k$$

$$xy^i z \quad i=0, 0^{2k-p}$$

Di fatto come al solito $|xy|$ non è $\leq k$ e quindi non è regolare

$$5) 0^n 0^m 0^{m+n}$$

Per assurdo ammettiamo che esista L.

Scegliamo w, $|w| \geq k$, $n=k$

$$w = 0^k 0^m 0^{m+k}$$

$$xy^i z \quad i=2, \quad 0^k 0^{2m} 0^{m+k}$$

$$x = 0^k \quad y = 0^{2m} \quad z = 0^{m+k}$$

Basterà intuitivamente dare un qualsiasi valore ad i diverso da 1, quindi 0 o maggiore di 1 come banalmente 2 per dimostrare che la stringa pompata non è regolare.

$$6) 0^n 0^{2^n}$$

Per assurdo ammettiamo che esista L.

Scegliamo w , $|w| \geq k$, $n=k$

$$w=0^k0^{2^k}$$

Un esempio banale; con $i=0$ si dimostra che il linguaggio non è regolare.

Infatti, assumendo:

$$xy=0^k \quad z=0^{2^k}$$

$$\text{avremo } w=10^{2^k}$$

che chiaramente non rispetta $xy \leq k$

Lunghezza minima pumping

Il pumping lemma afferma che ogni linguaggio regolare ha una lunghezza del pumping p , tale che ogni stringa del linguaggio può essere iterata se ha lunghezza maggiore o uguale a p . La *lunghezza minima del pumping* per un linguaggio A è il più piccolo p che è una lunghezza del pumping per A . Per ognuno dei seguenti linguaggi, dare la lunghezza minima del pumping e giustificare la risposta.

- | | | |
|----------------------------|---------------------------------------------|-------------------|
| (a) 110^* | (g) $10(11^*0)^*0$ | (m) ε |
| (b) $1^*0^*1^*$ | (h) 101101 | (n) $1^*01^*01^*$ |
| (c) $0^*1^*0^*1^* + 10^*1$ | (i) $\{w \in \Sigma^* \mid w \neq 101101\}$ | (o) 1011 |
| (d) $(01)^*$ | (j) 0001^* | (p) Σ^* |
| (e) \emptyset | (k) 0^*1^* | |
| (f) $0^*01^*01^*$ | (l) $001 + 0^*1^*$ | |

- (a) 3: si necessita di avere almeno tre caratteri per poter cominciare ad eseguire il pumping. Infatti parole come 00, 11 o similari non sono accettate.
- (b) 3: parole come 11 (cioè xy^0z) non sono accettate per costruzione, prendendo ad esempio 101 come xyz si nota che devono esserci entrambi per poter cominciare a pompare.
- (c) 3: L'unione c'è ma non ci interessa, la proprietà vale comunque. Inoltre si nota che devo avere almeno due 1 ma l'idea è la stessa dell'esercizio precedente.
- (d) 1: l'insieme vuoto deve avere almeno una stringa con cui operare e quindi pompare all'infinito una parola.
- (e) 2: abbiamo bisogno di entrambi i caratteri per pompare.
- (f) 3: le parole devono essere divise e definite in 3 pezzi anche qui per formare una parola valida e pompare.
- (g) 4: in pratica la lunghezza deve essere 4 in quanto potrebbe esserci una suddivisione che non sbilancia la stringa avendo soli tre caratteri, avendo stringhe che pompate non sarebbero nel linguaggio. Sapendo che $(11^*0)^*$ è l'espressione minima, avremo quantomeno bisogno di 0 oppure 1 per cominciare a pompare.
- (h) 4: si vede infatti che ipotizzando 3 come lunghezza minima, potremmo avere una stringa del tipo 111 che rimane sempre regolare.
- (i) 3: se sappiamo (penso io) che la stringa precedente non fa parte del linguaggio, significa considerando l'alfabeto precedente che necessitiamo di almeno tre caratteri per poter avere una stringa pompabile, considerando il caso complementare a quello descritto sopra.
- (j) 4: potremmo banalmente avere la stringa 000 che non sarebbe pompabile.
- (k) 2: anche qui, scegliendo 1 non sarebbe pompabile; con 2 almeno avremmo il possibile caso 01, regolarmente pompabile.
- (l) 3: lasciando stare il caso dell'unione, comunque si nota che la stringa 001 non può avere 2 come lunghezza minima pompabile.
- (m) 1: la stringa vuota deve necessitare di almeno un carattere qualsiasi.
- (n) 3: si considera infatti che la minima stringa effettivamente pompabile sia 001
- (o) 5: di fatto 1011 potrebbe non essere accettata dal linguaggio come pompata, perché già integralmente parte del linguaggio stesso.
- (p) 2: considerando che la stringa vuota necessita di almeno un carattere e l'alfabeto la comprende di sicuro essendo star, potrebbe bastare per un generico alfabeto avere un solo altro carattere.

Grammatiche context-free

La grammatica G_1 :

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

- insieme di **regole di sostituzione** (o **produzioni**)
- **variabili**: A, B
- **terminali** (simboli dell'alfabeto): $0, 1, \#$
- **variabile iniziale**: A

Una grammatica genera stringhe nel seguente modo:

- 1 Scrivi la variabile iniziale
- 2 Trova una variabile che è stata scritta e una regola che inizia con quella variabile. Sostituisci la variabile con il lato destro della regola
- 3 Ripeti 2 fino a quando non ci sono più variabili

Esempio per G_1 :

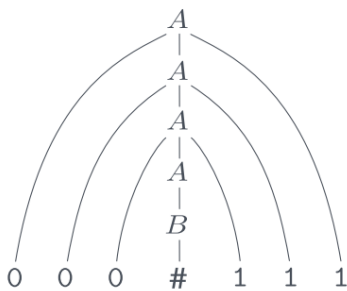
$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000\#111$

La sequenza di sostituzioni si chiama **derivazione** di $000\#111$

Da una derivazione si costruisce l'albero sintattico (parse tree), che descrive le derivazioni fattibili:

- la radice è la variabile iniziale
- i nodi interni sono variabili
- le foglie sono terminali

Ecco il parse tree per la grammatica sopra:



Più in generale, una grammatica CF si nota essere definita ricorsivamente da variabili fino ai terminali, in cui si usano le regole per descrivere un linguaggio con una serie di derivazioni. L'idea quindi è di definire sottostringhe collegate tra di loro ricorsivamente.

Se u, v, w sono stringhe di variabili e terminali e $A \rightarrow w$ è una regola:

- uAv produce uwv : $uAv \Rightarrow uwv$
- u deriva v : $u \Rightarrow^* v$ se:
 - $u = v$, oppure
 - esiste una sequenza u_1, u_2, \dots, u_k tale che $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$
- il linguaggio della grammatica è $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Un esempio concreto:

- Molti linguaggi sono **unione di linguaggi più semplici**
- **Idea:**
 - costruisci grammatiche separate per ogni componente
 - unisci le grammatiche con una nuova regola iniziale

$$S \rightarrow S_1 \mid S_2 \mid \dots \mid S_k$$

dove S_1, S_2, \dots, S_k sono le regole iniziali delle componenti

Esempio: grammatica per $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$

- Grammatica per $0^n 1^n$: $S_1 \rightarrow 0S_11 \mid \varepsilon$
- Grammatica per $1^n 0^n$: $S_2 \rightarrow 1S_20 \mid \varepsilon$
- **Unione:** $S \rightarrow S_1 \mid S_2$

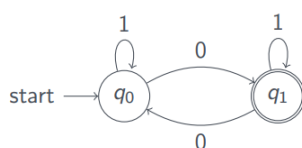
Se il linguaggio è regolare, esiste un DFA che lo riconosce e si segue questa idea con successiva applicazione pratica per trasformare un DFA in CFG:

- **Idea:** trasformiamo il DFA in grammatica:
 - una variabile R_i per ogni stato q_i
 - una regola $R_i \rightarrow aR_j$ per ogni transizione $\delta(q_i, a) = q_j$
 - una regola $R_i \rightarrow \varepsilon$ per ogni stato finale q_i
 - R_0 variabile iniziale, se q_0 è lo stato iniziale

Esempio

$\{w \in \{0, 1\}^* \mid w \text{ contiene un numero dispari di } 0\}$

- DFA per $\{w \in \{0, 1\}^* \mid w \text{ contiene un numero dispari di } 0\}$:



- Grammatica context-free:

$$R_0 \rightarrow 0R_1 \mid 1R_0$$

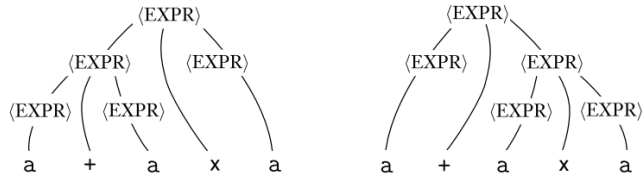
$$R_1 \rightarrow 0R_0 \mid 1R_1 \mid \varepsilon$$

Esistono anche le grammatiche ambigue, che possono portare a molteplici *leftmost derivations* (quindi più derivazioni a parità di regole partendo da sinistra):

G_5

$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$

Questa grammatica genera la stringa $a + a \times a$ in **due modi diversi!**



Attenzione

L'idea intuitiva, per capire che il linguaggio è corretto, è di attuare delle derivazioni partendo dalla leftmost fino alle rightmost, a quel punto si verifica se la propria CFG è corretta.

Esempio concreto:

Leftmost Derivation-

$S \rightarrow aB$

$\rightarrow aaBB$ (Using $B \rightarrow aBB$)

$\rightarrow aaaBBB$ (Using $B \rightarrow aBB$)

$\rightarrow aaabBB$ (Using $B \rightarrow b$)

$\rightarrow aaabbB$ (Using $B \rightarrow b$)

$\rightarrow aaabbaBB$ (Using $B \rightarrow aBB$)

$\rightarrow aaabbabB$ (Using $B \rightarrow b$)

$\rightarrow aaabbabbS$ (Using $B \rightarrow bS$)

$\rightarrow aaabbabbbA$ (Using $S \rightarrow bA$)

$\rightarrow aaabbabbba$ (Using $A \rightarrow a$)

$S \rightarrow aB / bA$

$S \rightarrow aS / bAA / a$

$B \rightarrow bS / aBB / b$

Forma normale di Chomsky

Di solito si usa scrivere la CFG in forma semplificata (utile nelle dimostrazioni pratiche in cui si chiede, *dato il linguaggio X che è context-free, dimostra che y è context-free; bisogna infatti trasformare la grammatica di quell'esercizio in Chomsky*), detta forma normale di Chomsky:

Una grammatica context-free è in **forma normale di Chomsky** se ogni regola è della forma

$$A \rightarrow BC$$

$$A \rightarrow a$$

dove a è un terminale, B, C non possono essere la variabile iniziale. Inoltre, ci può essere la regola $S \rightarrow \epsilon$ per la variabile iniziale S

in cui si segue questo ordine di regole:

Idea: possiamo trasformare una grammatica G in forma normale di Chomsky:

- 1 aggiungiamo una **nuova variabile iniziale**
- 2 eliminiamo le **ϵ -regole** $A \rightarrow \epsilon$
- 3 eliminiamo le **regole unitarie** $A \rightarrow B$
- 4 trasformiamo le regole rimaste nella forma corretta

(dove per il punto 5 sotto si intende, rimpiazza ogni variabile terminale sul lato destro di una regola con una nuova variabile e regola non terminale, tale che abbiamo a destra almeno 2 regole non terminali per il punto 4)

Quindi in generale:

1)

If the Start Symbol S occurs on some right side, create a new Start Symbol S' and a new Production $S' \rightarrow S$.

2)

Removal of Null Productions

In a CFG, a Non-Terminal Symbol ' A ' is a nullable variable if there is a production $A \rightarrow \epsilon$ or there is a derivation that starts at ' A ' and leads to ϵ . (Like $A \rightarrow \dots \rightarrow \epsilon$)

Procedure for Removal:

Step 1: To remove $A \rightarrow \epsilon$, look for all productions whose right side contains A

Step 2: Replace each occurrence of ' A ' in each of these productions with ϵ

Step 3: Add the resultant productions to the Grammar

3)

Any Production Rule of the form $A \rightarrow B$ where $A, B \in \text{Non Terminals}$ is called Unit Production

Procedure for Removal

Step 1: To remove $A \rightarrow B$, add production $A \rightarrow x$ to the grammar rule whenever $B \rightarrow x$ occurs in the grammar. [$x \in \text{Terminal}$, x can be Null]

Step 2: Delete $A \rightarrow B$ from the grammar.

Step 3: Repeat from Step 1 until all Unit Productions are removed.

4)

Replace each Production $A \rightarrow B_1 \dots B_n$ where $n > 2$, with $A \rightarrow B_1 C$ where $C \rightarrow B_2 \dots B_n$. Repeat this step for all Productions having two or more Symbols on the right side.

5)

If the right side of any Production is in the form $A \rightarrow aB$ where ' a ' is a terminal and A and B are non-terminals, then the Production is replaced by $A \rightarrow XB$ and $X \rightarrow a$. Repeat this step for every Production which is of the form $A \rightarrow aB$

Esempio completo step by step Chomsky:

Trasformiamo la grammatica G_6 in forma normale di Chomsky:

$$S \rightarrow ASA \mid aB$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b \mid \epsilon$$

In questo caso si vede che ci sono regole unitarie, ϵ -simboli e altro.

Per cominciare a trasformarla consideriamo (il testo barrato è quello eliminato):

1 aggiungiamo una nuova variabile iniziale $S_0 \notin V$ e la regola

$$S_0 \rightarrow S$$

In questo modo garantiamo che la variabile iniziale non compare mai sul lato destro di una regola

$$G' = (V', \Sigma, R', S_0)$$

dove si nota che S appare a destra e si introduce un nuovo stato iniziale.

$$S' \rightarrow S$$

$$S \rightarrow ASA \mid aB$$

$$A \rightarrow B \mid S \mid \epsilon$$

$$B \rightarrow b \mid \epsilon$$

dove metto ϵ anche in A perché la regola successiva va in B che va a sua volta in ϵ .
Successivamente:

2 Eliminiamo le ϵ -regole $A \rightarrow \epsilon$:

- se $A \rightarrow \epsilon$ è una regola dove A non è la variabile iniziale
- per ogni regola del tipo $R \rightarrow uAv$, aggiungiamo la regola

$$R \rightarrow uv$$

- **attenzione:** nel caso di più occorrenze di A, consideriamo tutti i casi: per le regole come $R \rightarrow uAvAw$, aggiungiamo

$$R \rightarrow uvAw \mid uAvw \mid uvw$$

- nel caso di regole $R \rightarrow A$ aggiungiamo $R \rightarrow \epsilon$ solo se non abbiamo già eliminato $R \rightarrow \epsilon$
- Ripeti finché non hai eliminato tutte le ϵ -regole

rimuovo le ϵ -regole, quindi $B \rightarrow \epsilon$ ed $A \rightarrow \epsilon$:

Rimuovendo $B \rightarrow \epsilon$ (quindi vuol dire che considero tutte le stringhe dove B è nullo)

$S' \rightarrow S$
 $S \rightarrow ASA|aB|a$
 $A \rightarrow B|S|\epsilon$
 $B \rightarrow b$

e poi rimuovo $A \rightarrow \epsilon$ (tutti i casi con ASA dove a è nullo e tolgo la ϵ):

$S' \rightarrow S$
 $S \rightarrow ASA|aB|a|AS|SA|S$
 $A \rightarrow B|S$
 $B \rightarrow b$

applicando poi la terza parte:

3 Eliminiamo le regole unitarie $A \rightarrow B$:

- se $A \rightarrow B$ è una regola unitaria
- per ogni regola del tipo $B \rightarrow u$, aggiungiamo la regola

$$A \rightarrow u$$

a meno che $A \rightarrow u$ non sia una regola unitaria eliminata in precedenza

- Ripeti finché non hai eliminato tutte le regole unitarie

avendo come regole unitarie (regola che va verso un'altra regola non terminale), quindi:

$S \rightarrow S$ $S' \rightarrow S$ $A \rightarrow B$ $A \rightarrow S$

Partiamo rimuovendo $S \rightarrow S$, che non fa nulla in pratica:

$S' \rightarrow S$
 $S \rightarrow ASA|aB|a|AS|SA$
 $A \rightarrow B|S$
 $B \rightarrow b$

quindi eliminiamo $S' \rightarrow S$ (quindi sostituisco S con ASA|aB|a|AS|SA)

$S' \rightarrow ASA|aB|a|AS|SA$
 $S \rightarrow ASA|aB|a|AS|SA$

$A \rightarrow B|S$
 $B \rightarrow b$

poi eliminiamo $A \rightarrow B$ (quindi sostituisco B con b):

$S' \rightarrow ASA|aB|a|AS|SA$
 $S \rightarrow ASA|aB|a|AS|SA$
 $A \rightarrow b|S$
 $B \rightarrow b$

ed infine eliminiamo $A \rightarrow S$ (quindi sostituisco S con $ASA|aB|a|AS|SA$):

$S' \rightarrow ASA|aB|a|AS|SA$
 $S \rightarrow ASA|aB|a|AS|SA$
 $A \rightarrow b|ASA|aB|a|AS|SA$
 $B \rightarrow b$

4 Trasformiamo le regole rimaste nella forma corretta:

- se $A \rightarrow u_1 u_2 \dots u_k$ è una regola tale che:

- ogni u_i è una variabile o un terminale
- $k \geq 3$

- sostituisci la regola con la catena di regole

$$A \rightarrow u_1 A_1, \quad A_1 \rightarrow u_2 A_2, \quad A_2 \rightarrow u_3 A_3, \quad \dots \quad A_{k-2} \rightarrow u_{k-1} u_k$$

- rimpiazza ogni terminale u_i sul lato destro di una regola con una nuova variabile U_i , e aggiungi la regola

$$U_i \rightarrow u_i$$

- ripeti per ogni regola non corretta

Ora dobbiamo trovare le produzioni che hanno più di 2 variabili a destra:

$S' \rightarrow ASA$ $S \rightarrow ASA$ $A \rightarrow ASA$

In pratica, vedendo che tutte hanno AS oppure SA come variabile rimpiazzabile, posso creare una nuova regola che le sostituisce, ottenendo:

$S' \rightarrow AX|aB|a|AS|SA$
 $S \rightarrow AX|aB|a|AS|SA$
 $A \rightarrow b|AX|aB|a|AS|SA$
 $B \rightarrow b$
 $X \rightarrow SA$

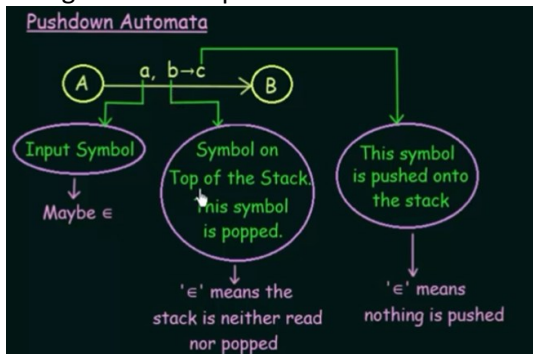
Adesso "a" è stato terminale e quindi dobbiamo cambiare tutte le produzioni che contengono "a" con una nuova regola (le espressioni sono $S' \rightarrow aB$, $S \rightarrow ab$, $A \rightarrow aB$), aggiungendo come regola $y \rightarrow a$:

$S' \rightarrow AX|YB|a|AS|SA$
 $S \rightarrow AX|YB|a|AS|SA$
 $A \rightarrow b|AX|YB|a|AS|SA$
 $B \rightarrow b$
 $X \rightarrow SA$
 $Y \rightarrow a$

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:

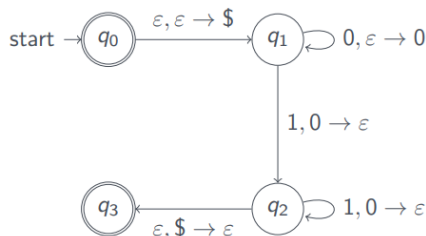


Partendo da una tabella di transizione:

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

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

- $P = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{0, \$\}, \delta, q_0, \{q_0, q_3\})$
- 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 $w_i \in \Sigma \cup \{\epsilon\}$
- 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

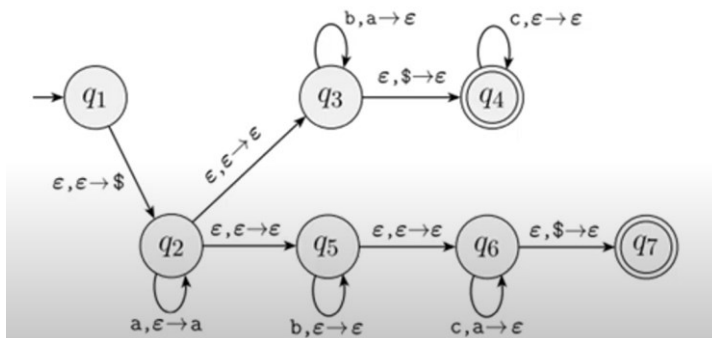
- 1 $r_0 = q_0$ e $s_0 = \epsilon$ (inizia dallo stato iniziale e pila vuota)
- 2 per ogni $i = 0, \dots, m-1$, $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ con $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\epsilon$ e $t \in \Gamma^*$ (rispetta la funzione di transizione)
- 3 $r_m \in F$ (la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



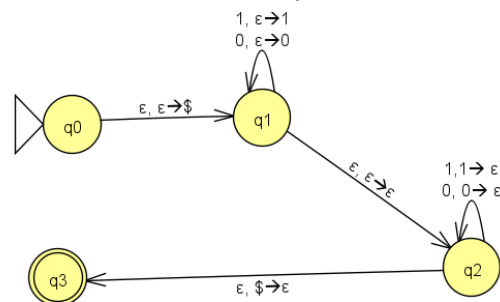
- 2 Costruisci un PDA per il linguaggio $\{ww^R \mid w \in \{0, 1\}^*\}$, dove w^R indica la parola w scritta al contrario

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'automata controlla idealmente se quello che c'era nello stack è uguale al resto della stringa, popando 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 $A \rightarrow u$ 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:

$$S \rightarrow aTb \mid b$$

$$T \rightarrow Ta \mid \varepsilon$$

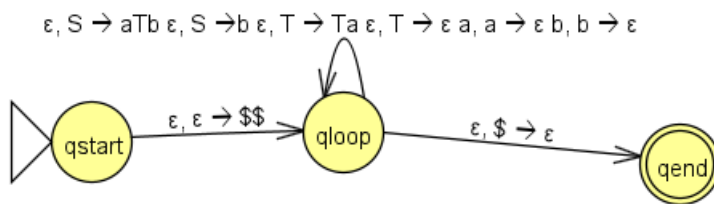
Dobbiamo metterci 3 stati:

q_{start}

q_{loop}

q_{end}

e successivamente diciamo “se c’è simbolo di input nella pila, poi lo rimuovo”:



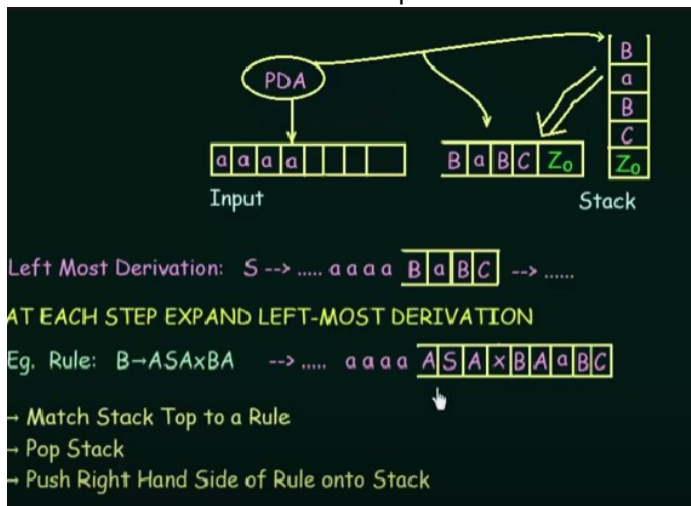
Per esempio scegliamo di attuare una derivazione

$S \rightarrow aTb \rightarrow aTab \rightarrow aab$

	a	T
	T	a
	b	b
	\$	\$

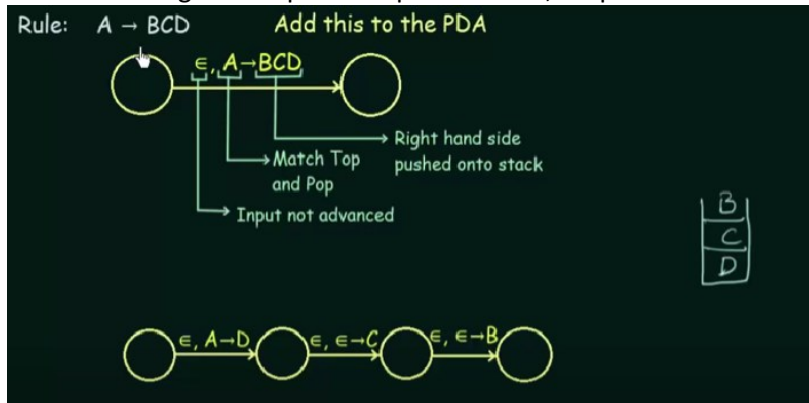
\$
\$

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:

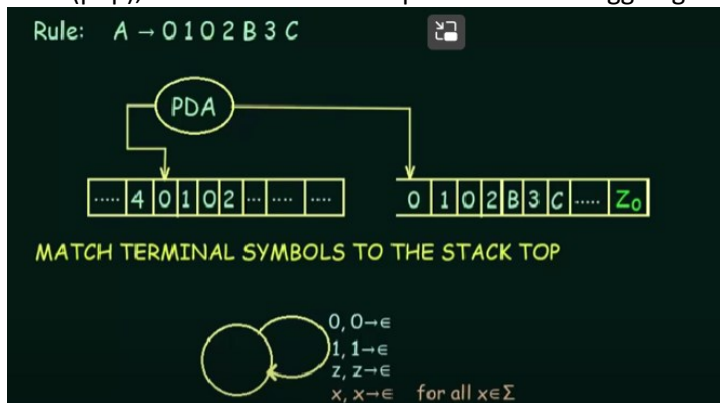


Quindi vogliamo partire sempre da sinistra con le derivazioni, facendo in modo di avere almeno una regola di quel tipo sulla cima dello stack.

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 essere tolto (pop), facendo avanzare l'input finché non raggiungiamo la fine dell'input:



Vediamo un altro esempio concreto (sempre trovato su YouTube, convertendo la CFG seguente in PDA).

$S \rightarrow aBc \mid ab$
 $B \rightarrow SB \mid \epsilon$

In pratica bisogna introdurre nell'ordine inverso le regole.

Quindi, notiamo che le uniche regole di lettura sono "a,b,c", messe nell'ordine inverso, quindi di rimozione che sarà fattibile dallo stack.

In tutti gli altri casi ragioniamo in questo modo.

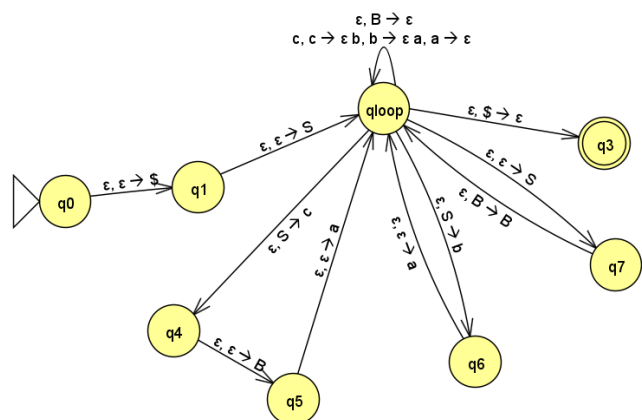
Prendiamo ad esempio $S \rightarrow aBc$.

Prendiamo prima c, quindi $\epsilon, S \rightarrow c$, poi mettiamo B, quindi $\epsilon, \epsilon \rightarrow B$, dunque mettiamo a, quindi $\epsilon, \epsilon \rightarrow a$.

Per tutte le altre regole si procede nella maniera similare.

Similmente per $S \rightarrow ab$, avremo prima $\epsilon, S \rightarrow b$ e successivamente il push di A senza pop, quindi $\epsilon, \epsilon \rightarrow a$.

Così per tutte le altre.



Conversione PDA-CFG

Si parte da una grammatica che:

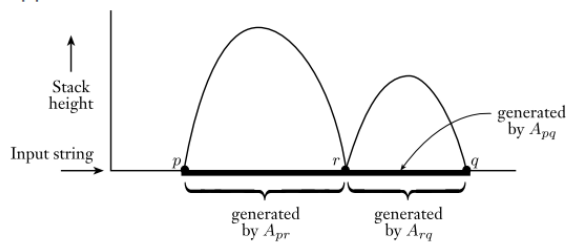
- una variabile A_{pq} per ogni coppia di stati p, q di P
- A_{pq} genera tutte le stringhe che portano da p con pila vuota a q con pila vuota

semplificando la pila tale che:

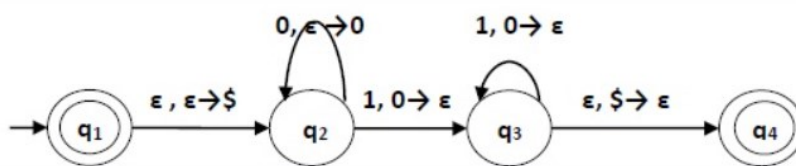
- 1 Ha un unico stato accettante q_f
- 2 Svuota la pila prima di accettare
- 3 Ogni transizione inserisce un unico simbolo sulla pila (push) oppure elimina un simbolo dalla pila (pop), ma non fa entrambe le cose contemporaneamente

sapendo quindi che a livello teorico abbiamo due casi:

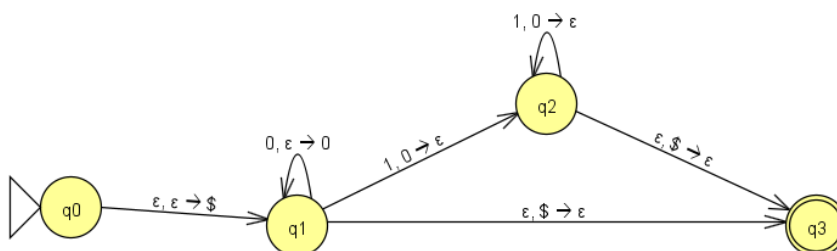
- Per andare da p con pila vuota a q con pila vuota:
 - la prima mossa deve essere un push
 - l'ultima mossa deve essere un pop
- Ci sono due casi:
 - 1 il simbolo inserito all'inizio viene eliminato alla fine
 - 2 oppure no:



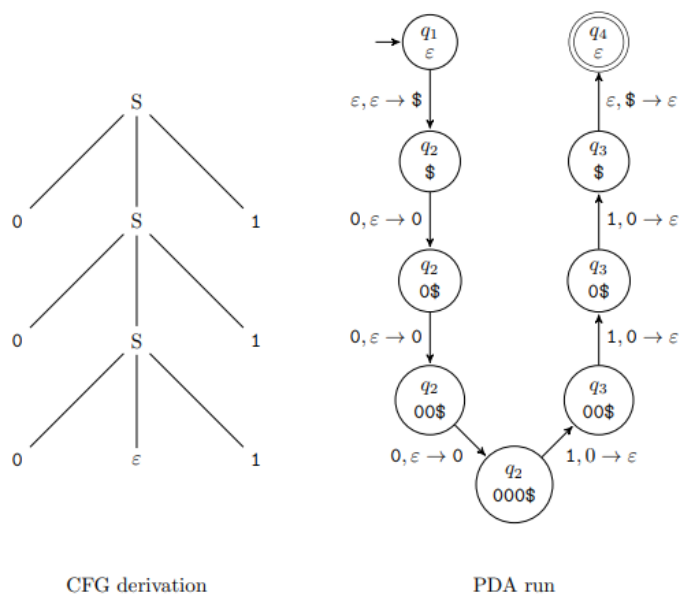
Trasformiamo il PDA per il linguaggio $\{0^n 1^n \mid n \geq 0\}$ in grammatica:



In poche parole mette la transizione che da q_2 va a q_4 inserendo $\epsilon, \$ \rightarrow \epsilon$ perché potrebbe non esserci nessun simbolo



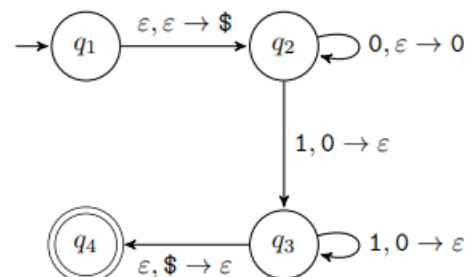
Quindi potremmo avere concretamente una situazione di questo tipo:



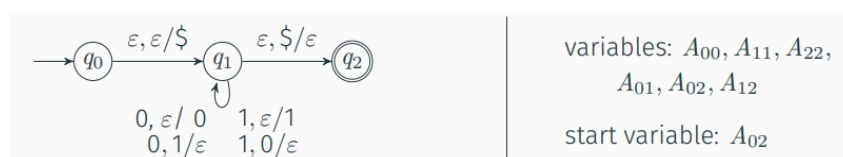
La grammatica prodotta è (riporto l'automa identico al nostro, ma comunque per chiarire col discorso lettere a fianco della CFG):

- 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

$A_{11} \rightarrow \epsilon$
 $A_{22} \rightarrow \epsilon$
 $A_{33} \rightarrow \epsilon$
 $A_{44} \rightarrow \epsilon$
 $A_{11} \rightarrow A_{11}A_{11} \mid A_{12}A_{21} \mid A_{13}A_{31} \mid A_{14}A_{41}$
 $A_{12} \rightarrow A_{11}A_{12} \mid A_{12}A_{22} \mid A_{13}A_{32} \mid A_{14}A_{42}$
 $A_{13} \rightarrow A_{11}A_{13} \mid A_{12}A_{23} \mid A_{13}A_{33} \mid A_{14}A_{43}$
 \dots
 $A_{42} \rightarrow A_{41}A_{12} \mid A_{42}A_{22} \mid A_{43}A_{32} \mid A_{44}A_{42}$
 $A_{43} \rightarrow A_{41}A_{13} \mid A_{42}A_{23} \mid A_{43}A_{33} \mid A_{44}A_{43}$
 $A_{44} \rightarrow A_{41}A_{14} \mid A_{42}A_{24} \mid A_{43}A_{34} \mid A_{44}A_{44}$
 $A_{23} \rightarrow 0A_{22}1 \mid 0A_{23}1$
 $A_{14} \rightarrow \epsilon A_{23} \epsilon$
 $S \rightarrow A_{14}$



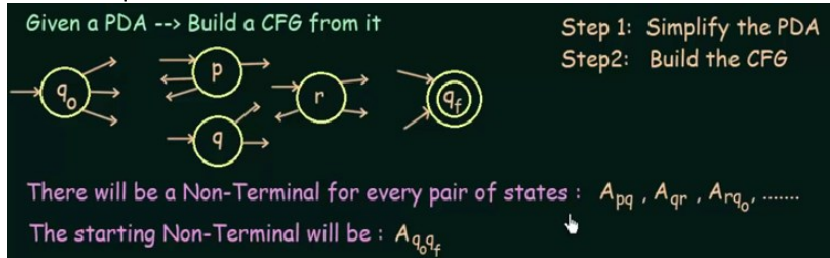
Altro esempio utile (dei pochi che ho trovato in giro su questa cosa):



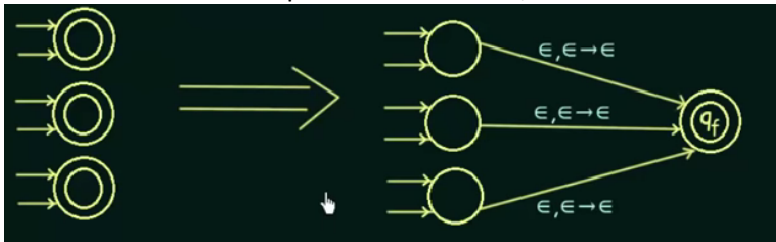
productions:

$A_{02} \rightarrow A_{01}A_{12}$
 $A_{01} \rightarrow A_{01}A_{11}$
 $A_{12} \rightarrow A_{11}A_{12}$
 $A_{11} \rightarrow A_{11}A_{11}$
 $A_{11} \rightarrow 0A_{11}1$
 $A_{11} \rightarrow 1A_{11}0$
 $A_{02} \rightarrow A_{11}$
 $A_{00} \rightarrow \epsilon, A_{11} \rightarrow \epsilon,$
 $A_{22} \rightarrow \epsilon$

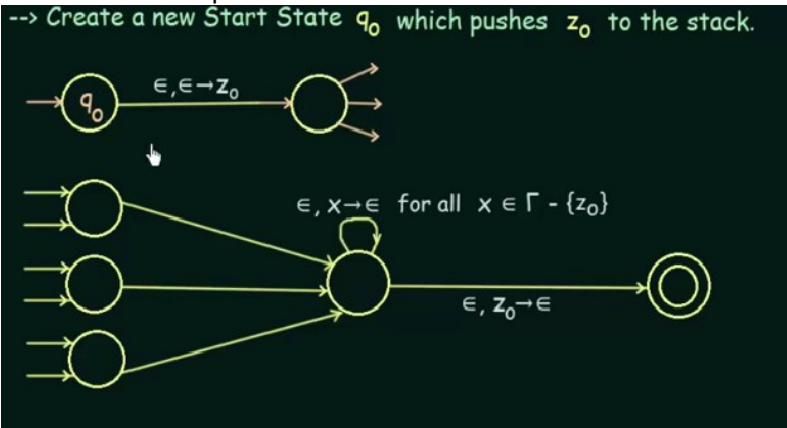
In sintesi quindi:



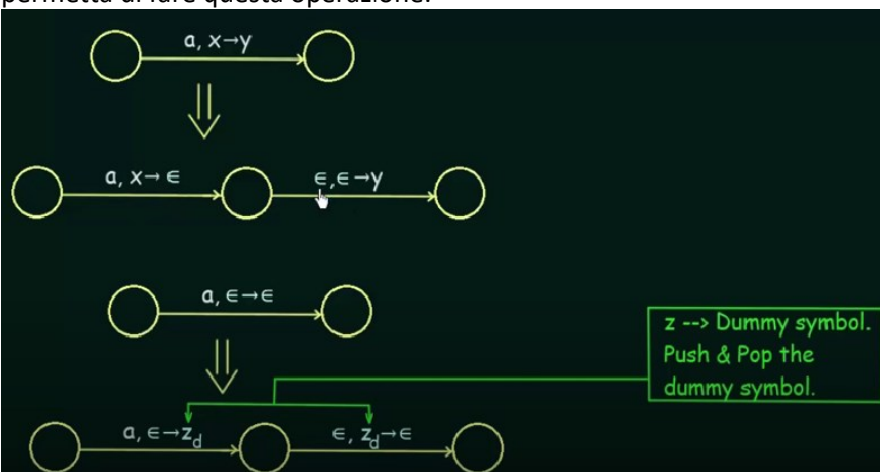
Cominciamo con la semplificazione del PDA, facendo in modo di avere un solo stato finale:



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:



Pumping Lemma per Linguaggi Context-Free (PL per CFL)

Theorem (Pumping Lemma per Linguaggi Context-free)

Sia L un *linguaggio context-free*. Allora

- *esiste una lunghezza* $k \geq 0$ *tale che*
- *ogni parola* $w \in L$ *di lunghezza* $|w| \geq k$
- *può essere spezzata* in $w = uvxyz$ *tale che:*
 - 1 $|vy| > 0$ (*il secondo o il quarto pezzo non sono la stringa vuota*)
 - 2 $|vxy| \leq k$ (*il blocco centrale è lungo al max k*)
 - 3 $\forall i \geq 0, uv^i xy^i z \in L$ (*possiamo "pompare" contemporaneamente v e y rimanendo in L*)

Il discorso è che qui il PL ha più condizioni, ma il risultato è identico al PL per i linguaggi regolari.

Qui semplicemente si divide la stringa in 5 parti e si esegue il pumping su 2 di queste, precisamente su "v" ed "y".

Nel caso lo chiedesse qualche esercizio, vediamo il gioco del PL per linguaggi CF:

- L'avversario sceglie la lunghezza k
- Noi scegliamo una parola w
- L'avversario spezza w in $uvxyz$
- Noi scegliamo i tale che $uv^i xy^i z \notin L$
- allora **abbiamo vinto**

Possiamo vedere queste derivazioni come alberi sintattici e, dato "b" numero massimo di simboli nel lato destro delle regole, ogni nodo avrà al massimo "b" figli. Dunque avremo in generale, per un albero di altezza "h", una stringa di lunghezza $\leq b^h$.

L'idea matematica è che, prendendo una lunghezza di pumping " k "= $b^{|V|+1}$ avremo che ogni albero sintattico per " w " ha altezza $\geq |V| + 1$. Dunque almeno una variabile si ripete e nelle derivazioni continuiamo a sostituire ricorsivamente i sottoalberi sintattici, tali da avere sempre un sottoalbero corretto.

Esempi concreti

- Il linguaggio $L_1 = \{a^n b^n \mid n \geq 0\}$
- Il linguaggio $L_2 = \{a^n b^n c^n \mid n \geq 0\}$
- Il linguaggio $L_3 = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$
- Il linguaggio $L_4 = \{ww^R \mid w \in \{0, 1\}^*\}$
- Il linguaggio $L_5 = \{ww \mid w \in \{0, 1\}^*\}$

1) Assumiamo che il linguaggio L sia context-free per assurdo.

Pertanto L dovrà avere una pumping length, che chiamiamo "p"

Le condizioni da rispettare sono:

- $uv^i xy^i z$ vero per ogni $i \geq 0$
- $|vy| > 0$
- $|vxy| \leq p$

Prendiamo poi una stringa $w = a^p b^p$

La dividiamo in 5 parti tali da avere $uvxyz$

Prendiamo $p=3$ e avremo quindi:

aaabbb

- $u = a$

- $v = aa$
- $x = b$
- $y = b$
- $z = b$

Pompriamo con $i=3$ su uv^2xy^2z ottenendo

aaaaabbbb

Si nota quindi che il numero di a è diverso da quello delle b.

Quindi il linguaggio non è CF.

2) Assumiamo che il linguaggio L sia context-free per assurdo.

Pertanto L dovrà avere una pumping length, che chiamiamo "p"

Le condizioni da rispettare sono:

- uv^ixy^iz vero per ogni $i \geq 0$
- $|vy| > 0$
- $|vxy| \leq p$

Prendiamo poi una stringa $w = a^pb^pc^p$

La dividiamo in 5 parti tali da avere $uvxyz$

Prendendo ad esempio $p=5$ avremo:

aaaaabbbbcccc

```
-- --- ----- -- --
u  v  x          y  z
```

A queste condizioni applicando ad esempio uv^2xy^2z

aaaaaaaabbbbcccccc

Come si può vedere a seguito del pumping, il numero di a, b e c risulta essere sbilanciato. Dunque

Dunque il linguaggio non è CF.

3) Assumiamo che il linguaggio L sia context-free per assurdo.

Pertanto L dovrà avere una pumping length, che chiamiamo "p"

Le condizioni da rispettare sono:

- uv^ixy^iz vero per ogni $i \geq 0$
- $|vy| > 0$
- $|vxy| \leq p$

Prendiamo poi una stringa $w = a^qb^rc^s$

Assumendo per l'appunto tutti gli esponenti ≥ 0 , prendiamo

una stringa w costituita come:

aaabbbbcccc $q=3; r=4; s=5$

- $u = a$
- $v = aa$
- $x = bb$
- $y = bb$
- $z = ccccc$

Pompriamo tipo con $p=2$, avendo quindi uv^2xy^2z e:

aaaaabbbbcccccc

Il numero di b non è $\leq k$ e quindi il linguaggio non è CF.

4) Assumiamo che il linguaggio L sia context-free per assurdo.

Pertanto L dovrà avere una pumping length, che chiamiamo "p"

Le condizioni da rispettare sono:

- uv^ixy^iz vero per ogni $i \geq 0$
- $|vy| > 0$
- $|vxy| \leq p$

Prendiamo poi una stringa $w = 0^p1^k0^p$

Assumendo per l'appunto tutti gli esponenti ≥ 0 , prendiamo una stringa w costituita come:

0000011100000 $q=5; r=3;$

- $u = 00$
- $v = 000$
- $x = 1110$
- $y = 00$
- $z = 00$

Prendiamo un pumping del tipo uv^2xy^2z avendo ad esempio:

000000001110000000

Si vede quindi che il numero di 0 non è pareggiato da entrambe le parti come invece dovrebbe essendo stringa palindroma, dunque il linguaggio non è CF.

5) Assumiamo che il linguaggio L sia context-free per assurdo. Pertanto L dovrà avere una pumping length, che chiamiamo " p "

Le condizioni da rispettare sono:

- uv^ixy^iz vero per ogni $i \geq 0$
- $|vy| > 0$
- $|vxy| \leq p$

Prendiamo poi una stringa $w = 0^p 1^k$

Assumendo per l'appunto tutti gli esponenti ≥ 0 , prendiamo una stringa w costituita come:

0000111111 $p=4; k=6;$

- $u = 00$
- $v = 00$
- $x = 111$
- $y = 11$
- $z = 00$

Come pumping prendiamo uv^3xy^3z , avendo quindi:

000000001111111100

Come si vede il linguaggio ha un numero di 0 diverso dal numero di 1 e dunque il linguaggio non può essere CF.