

LINGUAGGI FORMALI ED AUTOMI

Risposta alle domande più comuni degli esami

Operazioni sui linguaggi

- Dati due linguaggi A e B: A^c , $A \cap B$, $A \cup B$, $A \cdot B$, $B \cdot A$, A^* , A^+
- Elencare e definire formalmente le operazioni insiemistiche e tipiche dei linguaggi formali

Linguaggi

- Definizione di linguaggio ricorsivo
- Definizione di linguaggio ricorsivamente numerabile
- Esempio di linguaggio ricorsivamente numerabile ma non ricorsivo (dimostrare che è ricorsivamente numerabile)
- Se L è ricorsivo, è anche ricorsivamente numerabile? (procedura o controesempio)
- Se A e B sono ricorsivi, $A \cup B$ è ricorsivo? (procedura o controesempio)
- Dati A ricorsivo e B ricorsivamente numerabile, dare un programma che dimostri che $A \cap B$ è ricorsivamente numerabile
- Dati A ricorsivo e B ricorsivamente numerabile, dare un programma che dimostri che $A \cup B$ è ricorsivamente numerabile
- Se L è ricorsivo, L^c è ricorsivo? (procedura o controesempio)
- Definire formalmente linguaggio dell'arresto ristretto
- Dimostrare che il linguaggio dell'arresto ristretto è ricorsivamente numerabile (pseudocodice)

Grammatiche

- Definizione di Grammatica
- La parola $X \in L(G)$?
- Tipologie di grammatiche (distinzione)
- Dare una grammatica a/che genera un linguaggio
- Disegnare l'albero di derivazione data una grammatica
- Stabilire l'ambiguità (parole e grammatica, con definizione)
- Costruire una Grammatica di tipo X per il linguaggio L_y
- Costruire una G di tipo 3 per il linguaggio $L(G1) \cdot L(G2)$

Automi

- Da NFA a DFA
- Stati indistinguibili
- Fornire una grammatica equivalente all'automa
- Ricavare espressione regolare per $L(A)$ (evidenziando metodo)
- Disegnare automa che riconosce una determinata parola
- Definizione formale di NFA e DFA
- Definire formalmente \approx (indistinguibilità) e l'automa $A \approx$ equivalente ad A, corollario

Riconoscitore a pila

- Definizione formale (prestare attenzione a δ)
- Dare il grafo di computazione di A per la parola X
- Ricavare una grammatica di tipo 2 equivalente ad A
- Costruire un riconoscitore a pila data una grammatica di tipo 2
- Ricavare un riconoscitore a pila per il linguaggio L
- Dare un riconoscitore a pila per una parola

TEOREMI

Operazioni su linguaggi

- Dati due linguaggi A e B: $A^c, A \cap B, A \cup B, A \cdot B, B \cdot A, A^*, A^+$

Di solito queste domande propongono sempre due linguaggi e viene chiesto di eseguire un'operazione su di essi. Esempi:

(a) Che linguaggio è $A \cap B$?

Soluzione: Considerando che A è l'insieme delle parole binarie che terminano con 0 e B l'insieme delle parole binarie che terminano con 1, $A \cap B = \emptyset$. Infatti, non esiste nessuna parola binaria che abbia come suffisso contemporaneamente il simbolo 0 e 1.

(b) $A \cup B = \{0, 1\}^*$?

Soluzione: Considerando che $\{0, 1\}^*$ è l'insieme di tutte le parole binarie di lunghezza finita compresa la parola vuota ε , la risposta è NO. L'unione dei linguaggi A e B produce l'insieme delle parole binarie che terminano o con 0 o con 1, ma ε non soddisfa nessuna delle due condizioni perchè non contiene alcun simbolo, quindi non è compresa nell'unione $A \cup B$.

(c) $A \cdot B = B \cdot A$?

Soluzione: In genere, il prodotto di due linguaggi non è commutativo. Ma questo non basta per giustificare la risposta che è NO. È sufficiente osservare che un linguaggio, ad esempio $A \cdot B$, contiene una parola che l'altro linguaggio, $B \cdot A$, non contiene. Consideriamo la parola 01 essa è chiaramente una parola appartenente ad $A \cdot B$, ma terminando con 1 non può chiaramente appartenere a $B \cdot A$, che contiene solo parole con suffisso 0.

- Elencare e definire formalmente le operazioni insiemistiche e tipiche dei linguaggi formali

1) **UNIONE:** $A \cup B = \{w \in \Sigma^* \mid w \in A \text{ OR } w \in B\}$. A e B sono sottoinsiemi di $A \cup B$

2) **INTERSEZIONE:** $A \cap B = \{w \in \Sigma^* \mid w \in A \text{ AND } w \in B\}$ $A \cap B$ è sottoinsieme di A e B

3) **COMPLEMENTO:** $A^c = \{w \in \Sigma^* \mid w \notin A\}$ inoltre $A \cap A^c = \emptyset$

4) **PRODOTTO:** $AB = \{w \mid w = x \cdot y \text{ con } x \in A \text{ e } y \in B\}$, non è operazione commutativa

5) **POTENZA:** $A^k = A \cdot A \cdot A \cdot \dots \cdot A$ (per k volte)

6) **CHIUSURA DI KLEENE:** si indica in 2 modi diversi:

a) $\underline{L}^* = \{w \mid w = x_1 \cdot x_2 \cdot \dots \cdot x_n \text{ dove } x_i \in L \text{ con } 1 \leq i < n, n \geq 1\} \cup \{\varepsilon\}$
linguaggio formato dalle parole ottenute moltiplicando, in tutte le permutazioni possibili, parole di L, unitamente alla parola ε

b) $\underline{L}^+ = \{w \mid w = x_1 \cdot x_2 \cdot \dots \cdot x_n \text{ dove } x_i \in L \text{ con } 1 \leq i < n, n \geq 1\}$
 L_n è il linguaggio formato dalle parole ottenute moltiplicando, in tutte le permutazioni possibili, parole di L prese "n" alla volta.

$$L^* = L^+ \cup L^0$$

$$L^+ \neq L^* \setminus \{\varepsilon\}$$

Linguaggi

- Definizione di linguaggio ricorsivo

Soluzione: Informalmente, un linguaggio ricorsivo è un linguaggio che ammette un sistema riconoscitivo, cioè un sistema che è in grado di stabilire l'appartenenza o meno delle parole al linguaggio stesso. Tale risposta non può essere completamente accettata perché non è formalizzata e priva della possibilità di essere utilizzata a livello progettuale nella costruzione di programmi che mostrano proprietà sui linguaggi ricorsivi.

Pertanto, la risposta che deve essere data è la definizione *formale* di linguaggio ricorsivo.

Definizione:

Un linguaggio $L \subseteq \Sigma^*$ si dice ricorsivo quando esiste un algoritmo implementato da un programma w tale che, per ogni $x \in \Sigma^*$, si ha:

$$F_w(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

dove con F_w si indica la semantica del programma w .

- Definizione di linguaggio ricorsivamente numerabile

Soluzione: Informalmente, un linguaggio ricorsivamente numerabile è un linguaggio che ammette un sistema generativo, un sistema cioè che consente di elencare le parole del linguaggio. Tale sistema però non permette un completo riconoscimento del linguaggio stesso. Questa risposta non può essere completamente accettata perché non è formalizzata e priva della possibilità di essere utilizzata a livello progettuale nella costruzione di programmi che mostrano proprietà sui linguaggi ricorsivamente enumerabili.

Pertanto, la risposta che deve essere data è la definizione *formale* di linguaggio ricorsivamente numerabile.

Definizione:

Un linguaggio $L \subseteq \Sigma^*$ si dice ricorsivamente numerabile quando esiste una procedura implementata da un programma w tale che, per ogni $x \in \Sigma^*$, si ha:

$$F_w(x) = \begin{cases} 1 & x \in L \\ \uparrow & x \notin L \end{cases}$$

dove con F_w si indica la semantica del programma w e con \uparrow un programma che non termina.

- Esempio di linguaggio ricorsivamente numerabile ma non ricorsivo (dimostrare che è ricorsivamente numerabile)

L'esempio perfetto è il linguaggio dell'arresto ristretto

Il **linguaggio dell'arresto ristretto** è una variante del problema dell'arresto classico, ma in questo caso si considera l'arresto di una macchina di Turing su un input specifico e entro un numero limitato di passi. Formalmente, il linguaggio dell'arresto ristretto può essere definito come segue:

Definizione Formale

Sia M una macchina di Turing, w un input per M , e k un numero naturale che rappresenta il numero massimo di passi. Il **linguaggio dell'arresto ristretto** è l'insieme delle triple $\langle M, w, k \rangle$ tali che la macchina di Turing M , quando viene eseguita sull'input w , si ferma entro k passi.

Formalmente:

$$H_{\text{ristretto}} = \{ \langle M, w, k \rangle \mid M \text{ è una macchina di Turing che si ferma su } w \text{ entro } k \text{ passi} \}$$

Definiamo un linguaggio H che contiene le tuple (M, w) dove M è una procedura che termina con input w

Come dimostrare che è ricorsivamente numerabile:

Bisogna dimostrare che esiste una procedura N che accetta ogni coppia di (M, w) dove M si ferma con input w . Ottengo

$$N(M, w) = \begin{cases} \langle M, w \rangle \in H & N = \text{accetta} \\ \langle M, w \rangle \notin H & N = \text{non termina} \end{cases}$$

Il che significa che è ricorsivamente numerabile

Come dimostrare che non è ricorsivo:

Dimostrazione per assurdo. Supponiamo che esista una procedura R che decide H , ovvero può determinare dati (M, w) se M si ferma con w (definizione di linguaggio ricorsivo)

$$R(M, w) = \begin{cases} \langle M, w \rangle \in H & R = \text{accetta} \\ \langle M, w \rangle \notin H & R = \text{non accetta} \end{cases}$$

Costruiamo ora una procedura D che funziona in questo modo

$$D(M) = \begin{cases} \text{if } R(M, w) = \text{accetta} & D = \text{non termina} \\ \text{if } R(M, w) = \text{non accetta} & D = \text{accetta} \end{cases}$$

Questo è un paradosso, perché se do D in input a se stessa ottengo che se D si ferma non dovrebbe fermarsi, e se invece non termina dovrebbe farlo. Di conseguenza non può esistere una procedura del genere e il linguaggio non è ricorsivo

Soluzione prof:

Soluzione: Sia u il programma interprete, ovvero il programma che soddisfa $F_u(w\$x) = F_w(x)$ (cioè, u su input $w\$x$ mi restituisce il risultato del programma w su input x). Si definisce il linguaggio dell'arresto ristretto l'insieme:

$$D = \{x \in \{0, 1\}^* \mid F_u(x\$x) \downarrow\},$$

dove \downarrow indica che il programma termina. Il linguaggio D è un esempio di linguaggio ricorsivamente numerabile ma non ricorsivo. Per quanto riguarda la prima affermazione, si consideri la seguente procedura:

```
RICNUM( $x \in \{0, 1\}^*$ )  
 $y = F_u(x\$x)$   
return 1
```

Si noti che se $x \in D$ allora $F_u(x\$x) \downarrow$ e $\text{RICNUM}(x) = 1$, mentre se $x \notin D$ allora $F_u(x\$x) \uparrow$ e $\text{RICNUM}(x) \uparrow$. Quindi RICNUM dimostra che D è ricorsivamente numerabile. Inoltre, è anche possibile dimostrare che D non è ricorsivo.

- Se L è ricorsivo, è anche ricorsivamente numerabile? (procedura o controesempio)

**Un linguaggio L è ricorsivo per definizione se esiste una procedura M che può decidere L
ovvero**

M(L)=	M accetta ogni $w \in L$	M = accetta e termina
	M rifiuta ogni $w \notin L$	M = rifiuta e termina

Mentre è ricorsivamente numerabile se esiste N che

N(L)=	$w \in L$	N = accetta e termina
	$w \notin L$	N = non termina

Dato che sappiamo che L è ricorsivo e quindi esiste M, definiamo una nuova macchina M' che funziona così:

M'(L)=	Se M accetta w	M' accetta w
	Se M rifiuta w	M non termina

Poiché M si arresta sempre e decide L, la macchina M' accetterà tutti e soli gli elementi di L, ma non è necessario che M' rifiuti in modo definitivo per gli input non appartenenti a L (può semplicemente non arrestarsi, soddisfacendo la definizione di linguaggio ricorsivamente numerabile).

In altre parole, ogni linguaggio ricorsivo ha una macchina che si arresta sempre (quindi, decide il linguaggio), ma anche una macchina che accetta solo le stringhe in L e potrebbe non fermarsi per le altre stringhe, il che è sufficiente per dimostrare che il linguaggio è anche ricorsivamente numerabile.

- Se A e B sono ricorsivi, $A \cup B$ è ricorsivo? (programma o controesempio)

Supponiamo di avere due macchine M_a e M_b che decidono A e B

$M_a(w)=$	$w \in A$	$M_a =$ accetta w
	$w \notin A$	$M_a =$ rifiuta w

$M_b(w)=$	$w \in B$	$M_b =$ accetta w
	$w \notin B$	$M_b =$ rifiuta w

Ora costruiamo la macchina per $A \cup B$ usando queste

	$w \in A$	$M_{A \cup B} =$ accetta w	
$M_{A \cup B}(w) =$	$M_a(w) =$	$w \in B$	$M_{A \cup B} =$ accetta w
	$w \notin A$	$M_{A \cup B} = M_b(w) =$	
		$w \notin B$	$M_{A \cup B} =$ rifiuta w

La macchina è perciò in grado di determinare se una parola appartiene ad $A \cup B$ oppure no, quindi $A \cup B$ è ricorsivo

- Dati A ricorsivo e B ricorsivamente numerabile, dare un programma che dimostri che $A \cap B$ è ricorsivamente numerabile

Supponiamo di avere due macchine M_a e M_b per A ricorsivo e B ricorsivamente numerabile

	$w \in A$	$M_a =$ accetta w
$M_a(w) =$		
	$w \notin A$	$M_a =$ rifiuta w
	$w \in B$	$M_b =$ accetta w
$M_b(w) =$		
	$w \notin B$	$M_b =$ no stop

Ora costruiamo la macchina per $A \cap B$ usando queste

	$w \notin A$	$M_{A \cap B} =$ rifiuta w	
$M_{A \cap B}(w) =$	$M_a(w) =$	$w \in B$	$M_{A \cap B} =$ accetta w
	$w \in A$	$M_{A \cap B} = M_b(w) =$	
		$w \notin B$	$M_{A \cap B} =$ no stop

La macchina è perciò in grado di determinare se una parola appartiene ad $A \cap B$ oppure no, quindi $A \cap B$ è ricorsivo

- Dati A ricorsivo e B ricorsivamente numerabile, dare un programma che dimostri che $A \cup B$ è ricorsivamente numerabile

Supponiamo di avere due macchine M_A e M_B per A ricorsivo e B ricorsivamente numerabile

$w \in A$ $M_A = \text{accetta } w$

$M_A(w) =$

$w \notin A$ $M_A = \text{rifiuta } w$

$w \in B$ $M_B = \text{accetta } w$

$M_B(w) =$

$w \notin B$ $M_B = \text{no stop}$

Ora costruiamo la macchina per $A \cup B$ usando queste

$w \notin A$ $M_{A \cup B} = \text{accetta } w$

$M_{A \cup B}(w) =$ $M_A(w) =$ $w \in B$ $M_{A \cup B} = \text{accetta } w$

$w \in A$ $M_{A \cup B} = M_B(w) =$

$w \notin B$ $M_{A \cup B} = \text{no stop}$

La macchina è perciò in grado di determinare se una parola appartiene ad $A \cap B$ oppure no, quindi $A \cup B$ è ricorsivo

- Se L è ricorsivo, L^c è ricorsivo? (procedura o controesempio)

Stabilire se L^c è ricorsivo sapendo che L lo è è molto semplice. Dato che L è ricorsivo esisterà una procedura M_L tale che

$w \in L$ $M_L = \text{true} (1)$

$M_L(w) =$

$w \notin L$ $M_L = \text{false} (0)$

Per trovare la procedura che determina L^c basta invertire l'output

$M_L = \text{true}$ $w \in L$ e quindi $w \notin L^c (0)$

$M_L = \text{false}$ $w \notin L$ e quindi $w \in L^c (1)$

Dato che possiamo determinare esattamente se una parola appartiene o non appartiene a L^c esso è ricorsivo

- Definire formalmente linguaggio dell'arresto ristretto

Il **linguaggio dell'arresto ristretto** è una variante del problema dell'arresto classico, ma in questo caso si considera l'arresto di una macchina di Turing su un input specifico e entro un numero limitato di passi. Formalmente, il linguaggio dell'arresto ristretto può essere definito come segue:

Definizione Formale

Sia M una macchina di Turing, w un input per M , e k un numero naturale che rappresenta il numero massimo di passi. Il **linguaggio dell'arresto ristretto** è l'insieme delle triple $\langle M, w, k \rangle$ tali che la macchina di Turing M , quando viene eseguita sull'input w , si ferma entro k passi.

Formalmente:

$$H_{\text{ristretto}} = \{ \langle M, w, k \rangle \mid M \text{ è una macchina di Turing che si ferma su } w \text{ entro } k \text{ passi} \}$$

- Dimostrare che il linguaggio dell'arresto ristretto è ricorsivamente numerabile (pseudocodice)

Definiamo un linguaggio H che contiene le tuple (M, w) dove M è una procedura che termina con input w

Come dimostrare che è ricorsivamente numerabile:

Bisogna dimostrare che esiste una procedura N che accetta ogni coppia di (M, w) dove M si ferma con input w . Ottengo

$$\langle M, w \rangle \in H \quad N = \text{accetta}$$

$$N(M, w) =$$

$$\langle M, w \rangle \notin H \quad N = \text{non termina}$$

Il che significa che è ricorsivamente numerabile

Grammatiche

- Definizione di Grammatica

1. Simboli terminali (T):

- Questi sono i simboli "finali" del linguaggio, cioè i simboli che compaiono effettivamente nelle stringhe del linguaggio generato dalla grammatica.
- Esempi di simboli terminali potrebbero essere lettere, numeri o qualsiasi altro carattere che non viene ulteriormente sostituito durante la generazione delle stringhe.

2. Simboli non terminali (V):

- Questi simboli rappresentano categorie o variabili che possono essere espanse o sostituite durante la derivazione di stringhe nel linguaggio.
- I simboli non terminali sono utilizzati nelle regole di produzione per generare stringhe di simboli terminali.
- Il simbolo iniziale della grammatica (vedi sotto) è sempre un simbolo non terminale.

3. Simbolo iniziale (S):

- Questo è un simbolo speciale tra i simboli non terminali, che rappresenta il punto di partenza per la generazione delle stringhe.
- La derivazione di qualsiasi stringa nel linguaggio inizia dal simbolo iniziale.

4. Regole di produzione (P):

- Le regole di produzione definiscono come i simboli non terminali possono essere sostituiti o espansi in altri simboli non terminali e/o terminali.
- Una regola di produzione ha la forma generale $\alpha \rightarrow \beta$, dove α è una stringa contenente almeno un simbolo non terminale, e β è una stringa contenente simboli terminali e/o non terminali.
- Queste regole descrivono come si possono formare stringhe di simboli terminali a partire dal simbolo iniziale.



Una grammatica G è formalmente rappresentata come una quadrupla:

$$G = (T, V, S, P)$$

Dove:

- T è l'insieme dei simboli terminali.
- V è l'insieme dei simboli non terminali.
- $S \in V$ è il simbolo iniziale.
- P è l'insieme delle regole di produzione.

Questi quattro elementi lavorano insieme per definire come le stringhe di un linguaggio possono essere generate o riconosciute dalla grammatica.

- La parola $X \in L(G)$?

Per determinare se una parola appartiene a una data grammatica bisogna provare a generare tale parole utilizzando le regole di produzione della stessa, e vedere se si riesce ad ottenerla
Sia la grammatica

$$G = (T = \{ (,) \}, V = \{ S \}, P = \{ S \rightarrow (S), S \rightarrow ()S, S \rightarrow () \}).$$

(a) Rispondere alle seguenti domande **giustificando la risposta**:

- La parola “ $()()((()))$ ” appartiene a $L(G)$?

Soluzione: Sì, infatti una parola appartiene al linguaggio generato da una grammatica, in questo caso G , quando è possibile individuare una derivazione per essa in G . Per giustificare la risposta dobbiamo quindi dare l'intera derivazione della parola:
 $S \Rightarrow ()S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()((S)) \Rightarrow ()()((()))$.

- La parola “ (S) ” appartiene a $L(G)$?

Soluzione: No, una parola per appartenere al linguaggio generato da una grammatica, in questo caso G , non deve solo ammettere una derivazione per essa in G , ma deve anche essere composta da soli simboli terminali che in questo caso sono definiti dall'insieme $T = \{ (,) \}$. La parola (S) non appartiene a T^* , infatti, ammette come fattore il simbolo $S \in V$. Nessuna parola contenente una variabile o metasimbolo di G appartiene a $L(G)$.

- Tipologie di grammatiche (distinzione)

- **G di Tipo 0:** le regole di produzione sono completamente arbitrarie
- **G di Tipo 1:** ogni regola di produzione $\alpha \rightarrow \beta$ deve essere tale che $|\beta| \geq |\alpha|$. Inoltre:
 - si aggiunge in M il metasimbolo S'
 - si aggiunge in P la regola $S' \rightarrow \epsilon$ e $S' \rightarrow S$
- **G di tipo 2:** ogni regola di produzione $\alpha \rightarrow \beta$ deve essere tale che α sia un metasimbolo
 - ~~$\beta \in (\Sigma \cup M)^*$~~ ma ora $\beta \in (\Sigma \cup M)^*$ (consente di aggiungere $A \rightarrow \epsilon$)
- **G di tipo 3:** ogni regola di produzione $\alpha \rightarrow \beta$ deve essere tale che si presenti in una delle seguenti due forme:
 - $\alpha \rightarrow x$ \Leftrightarrow con $\alpha \in M$ e $x \in \Sigma^*$ (consente creazione di regole nella forma $A \rightarrow \epsilon$.)
 - $\alpha \rightarrow y\beta$ \Leftrightarrow con $\alpha \in M$, $y \in \Sigma^*$ e $\beta \in M$

Le grammatiche di tipo 3 sono identificate da regole di produzione che assumono una delle seguenti forme:

- 1) $A \rightarrow x$ oppure $A \rightarrow yB$
- 2) $A \rightarrow \sigma$ oppure $A \rightarrow \sigma B$ oppure $A \rightarrow \epsilon$
- 3) $A \rightarrow \sigma B$ oppure $A \rightarrow \epsilon$
- 4) $A \rightarrow \sigma B$ oppure $A \rightarrow \sigma$

\rightarrow dove $A, B \in M$, $x, y \in \Sigma^*$, $\sigma \in \Sigma$

Queste quattro categorie formano una gerarchia, dove le grammatiche di tipo 3 sono le più restrittive e quelle di tipo 0 le più generali. Ogni linguaggio generato da una grammatica di un certo tipo può essere generato anche da una grammatica di un tipo superiore (meno restrittivo), ma non necessariamente viceversa.

- Dare una grammatica a/che genera un linguaggio

Dare una grammatica di tipo 2 per il linguaggio $\{a^n c^m b^n \mid n, m \geq 0\}$.

Soluzione: Diamo una possibile grammatica di tipo 2 per il linguaggio $\{a^n c^m b^n \mid n, m \geq 0\}$, ricordando la grammatica per il famoso linguaggio $\{a^n b^n \mid n \geq 0\}$ ed usandola per ottenere quella del linguaggio in questione. Definiamo:

$$G = (T = \{a, b, c\}, V = \{S, C\}, S, P = \{S \rightarrow aSb, S \rightarrow C, C \rightarrow cC, C \rightarrow \varepsilon\}).$$

La grammatica G permette di generare parole della forma $a^n S b^n$, per $n \geq 0$, semplicemente iterando la regola $S \rightarrow aSb$. Per eliminare la variabile S dobbiamo necessariamente passare dalla variabile C con la regola $S \rightarrow C$, ottenendo $a^n C b^n$. A questo punto è possibile inserire un numero arbitrario m , di simboli c con le regole $C \rightarrow cC, C \rightarrow \varepsilon$, ottenendo una parola della forma $a^n c^m b^n$. Si noti che dovendo una derivazione partire da S , l'assioma di G , non è possibile ottenere altre parole che non siano della forma richiesta.

Dobbiamo ora giustificare che la grammatica G che abbiamo appena dato sia di tipo 2. Una grammatica è di tipo 2 se le sue regole di produzione sono della forma $A \rightarrow \beta$ con $A \in V$ e $\beta \in (V \cup T)^*$. Ogni regola in P soddisfa questa condizione e pertanto G è di tipo 2.

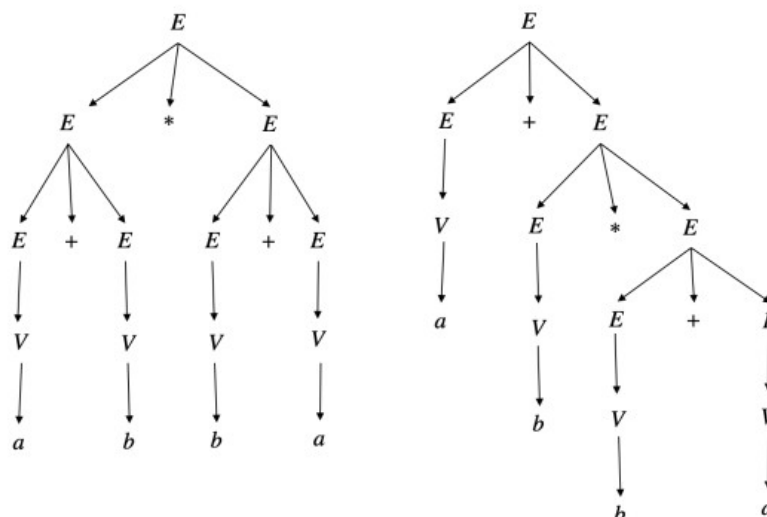
- Disegnare l'albero di derivazione data una grammatica

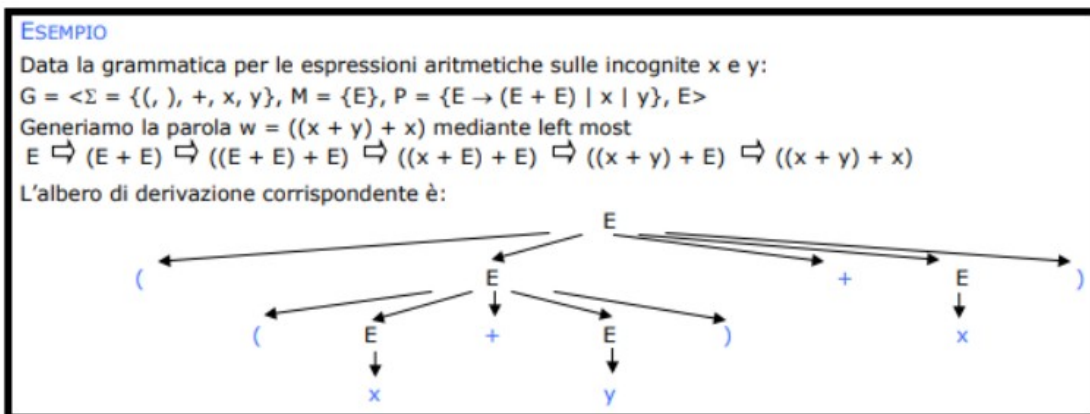
Sia la grammatica

$$G = (T = \{a, b, +, *\}, V = \{E, V\}, E, P = \{E \rightarrow E+E, E \rightarrow E*E, E \rightarrow V, V \rightarrow a, V \rightarrow b\})$$

(a) Disegnare l'albero di derivazione per $a + b * b + a$. È unico?

Soluzione: No, per tale parola esistono almeno due alberi di derivazione in G differenti. Ad esempio i seguenti sono alberi di derivazione per $a + b * b + a$:





Derivazione **left-most**: quando una derivazione viene effettuata derivando sempre a partire dal metasimbolo più a sinistra. Nell'esempio sopra si utilizza.

Data una grammatica G , affermeremo che **due derivazioni sono equivalenti** se hanno associato lo stesso albero di derivazione.

- Stabilire l'ambiguità (parole e grammatica, con definizione)

Una grammatica si dice ambigua se genera almeno una parola ambigua, ovvero esiste più di un albero di derivazione per la stessa parola

Stabilire se la grammatica G è ambigua. Giustificare la risposta.

Soluzione: La grammatica G è ambigua in quanto esiste una parola in $L(G)$ che ammette due alberi di derivazione differenti. Ad esempio, la parola $a + b * b + a$ citata sopra.

- Costruire una Grammatica di tipo X per il linguaggio L_y

Dare una grammatica di tipo 2 per il linguaggio $\{a^n c^m b^n \mid n, m \geq 0\}$.

Soluzione: Diamo una possibile grammatica di tipo 2 per il linguaggio $\{a^n c^m b^n \mid n, m \geq 0\}$, ricordando la grammatica per il famoso linguaggio $\{a^n b^n \mid n \geq 0\}$ ed usandola per ottenere quella del linguaggio in questione. Definiamo:

$$G = (T = \{a, b, c\}, V = \{S, C\}, S, P = \{S \rightarrow aSb, S \rightarrow C, C \rightarrow cC, C \rightarrow \varepsilon\}).$$

La grammatica G permette di generare parole della forma $a^n S b^n$, per $n \geq 0$, semplicemente iterando la regola $S \rightarrow aSb$. Per eliminare la variabile S dobbiamo necessariamente passare dalla variabile C con la regola $S \rightarrow C$, ottenendo $a^n C b^n$. A questo punto è possibile inserire un numero arbitrario m , di simboli c con le regole $C \rightarrow cC, C \rightarrow \varepsilon$, ottenendo una parola della forma $a^n c^m b^n$. Si noti che dovendo una derivazione partire da S , l'assioma di G , non è possibile ottenere altre parole che non siano della forma richiesta.

Dobbiamo ora giustificare che la grammatica G che abbiamo appena dato sia di tipo 2. Una grammatica è di tipo 2 se le sue regole di produzione sono della forma $A \rightarrow \beta$ con $A \in V$ e $\beta \in (V \cup T)^*$. Ogni regola in P soddisfa questa condizione e pertanto G è di tipo 2.

- Costruire una G di tipo 3 per il linguaggio $L(G_1) \cdot L(G_2)$

Per costruire una grammatica G_3 di tipo 3 che genera il linguaggio $L(G_1) \cdot L(G_2)$, dobbiamo combinare le grammatiche G_1 e G_2 in modo tale che le stringhe generate da G_1 siano concatenate con le stringhe generate da G_2 .

Grammatiche iniziali:

- $G_1 = (T_1, V_1, S_1, P_1)$
- $G_2 = (T_2, V_2, S_2, P_2)$

Dove:

- T_1 e T_2 sono i simboli terminali.
- V_1 e V_2 sono le variabili (o simboli non terminali).
- S_1 e S_2 sono i simboli iniziali.
- P_1 e P_2 sono gli insiemi di regole di produzione.

Costruzione di G_3

Definiamo una nuova grammatica $G_3 = (T_3, V_3, S_3, P_3)$ per generare il linguaggio $L(G_1) \cdot L(G_2)$.

1. **Simboli terminali:** Poiché G_3 deve generare stringhe che appartengono a $L(G_1) \cdot L(G_2)$, i simboli terminali saranno l'unione dei terminali di G_1 e G_2 :

$$T_3 = T_1 \cup T_2$$

2. **Simboli non terminali:** I simboli non terminali di G_3 includeranno tutti i non terminali di G_1 e G_2 , e aggiungeremo un nuovo simbolo iniziale S_3 . Inoltre, dobbiamo considerare una copia modificata del simbolo iniziale S_2 di G_2 per evitare conflitti tra le grammatiche:

$$V_3 = V_1 \cup V_2 \cup \{S_3, S'_2\}$$

3. **Simbolo iniziale:** Il nuovo simbolo iniziale sarà S_3 .

4. **Regole di produzione:**

- Iniziamo da S_3 e transizioniamo direttamente al linguaggio di G_1 tramite il suo simbolo iniziale S_1 .
- Aggiungiamo una regola che, al termine della generazione di una stringa da G_1 (ovvero quando si raggiunge un simbolo non terminale che porta a ϵ in G_1), passa a S'_2 , che agisce come nuovo simbolo iniziale per G_2 .
- Infine, aggiungiamo tutte le regole di G_1 e G_2 a P_3 .

Formalmente:

$$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1\} \cup \{A \rightarrow \sigma S'_2 \mid A \rightarrow \sigma \in P_1 \text{ e } S_2 \rightarrow \epsilon \in P_2\}$$

Regole di produzione dettagliate:

1. Transizione dal simbolo iniziale:

$$S_3 \rightarrow S_1$$

2. Regole per combinare G_1 e G_2 :

- Per ogni regola $A \rightarrow \sigma$ in P_1 dove A è un non terminale che produce una stringa terminale (cioè quando la produzione finisce in G_1), aggiungiamo:

$$A \rightarrow \sigma S'_2$$

- Tutte le regole originali di P_1 e P_2 rimangono intatte:

$$P_3 = P_1 \cup \{A \rightarrow \sigma S'_2\} \cup P_2$$

3. Il nuovo simbolo S'_2 corrisponde a S_2 in G_2 , ma senza conflitti con G_1 .

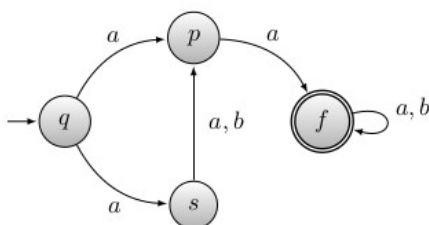
Riassumendo:

La grammatica G_3 genererà stringhe che sono il risultato della concatenazione delle stringhe di $L(G_1)$ e $L(G_2)$. La costruzione di G_3 è tale che ogni stringa in G_3 inizia con una stringa da G_1 e prosegue con una stringa da G_2 .

Automi

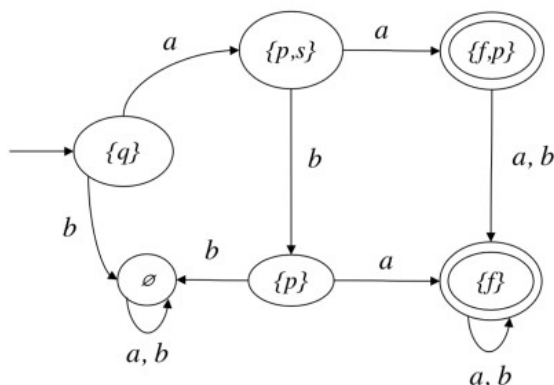
- Da NFA a DFA

Sia A il seguente automa nondeterministico seguente:



Disegnare un automa deterministico equivalente ad A .

Soluzione: Usando la subset construction mostrata nel teorema di equivalenza tra gli automi a stati finiti deterministici e nondeterministici si ottiene l'automato deterministico seguente:



- Stati indistinguibili

- stati **indistinguibili** \approx : lo stato raggiunto nell'automa, partendo da q_1 , leggendo una qualsiasi parola w deve essere obbligatoriamente uguale allo stato raggiunto nell'automa, partendo da q_2 , leggendo la medesima parola w , questo deve accadere per tutte le parole appartenenti a Σ^*

In A sono presenti stati indistinguibili tra loro? Se sì quali?

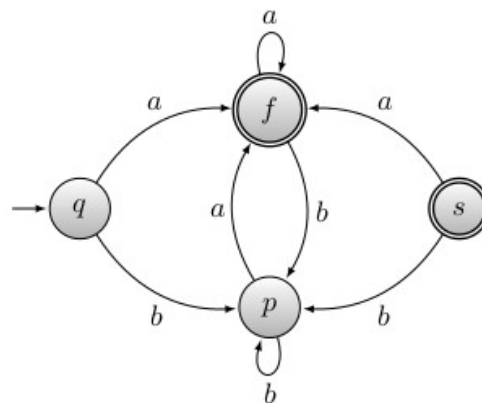
Soluzione: Sì, gli stati indistinguibili sono (p, q) ed (f, s) . Per definizione $p \approx q$ se e solo se per ogni $w \in \{a, b\}^*$ vale $\lambda(\delta^*(q, w)) = \lambda(\delta^*(p, w))$. Dimostrare che ciò è vero in questo caso è facile. Infatti valgono le seguenti uguaglianze:

$$\delta(q, a) = \delta(p, a) \text{ e } \delta(q, b) = \delta(p, b).$$

Qualunque sia la parola w che leggiamo a partire da p o da q , dopo il primo simbolo di w , si raggiunge sempre lo stesso stato e pertanto le risposte dell'automa A non potranno che essere uguali. Similarmente accade per f ed s .

- Fornire una grammatica equivalente all'automa

Sia A il seguente automa:



Fornire una grammatica G equivalente ad A .

Soluzione: Si utilizza la costruzione presente nella dimostrazione del teorema che afferma l'equivalenza tra le grammatiche di tipo 3 e gli automi a stati finiti.

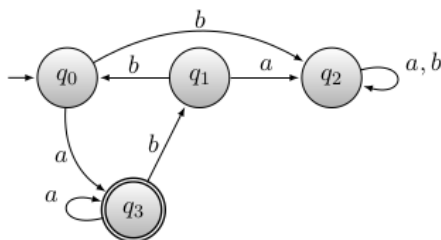
Senza cambiare notazione rispetto a quella che definisce l'automa A , definiamo la grammatica:

$$G = (T = \{a, b\}, V = \{q, p, f, s\}, q \text{ (assioma)}, P),$$

dove le seguenti regole di produzione appartengono a P :

- $q \rightarrow af, q \rightarrow bp,$
- $p \rightarrow bp, p \rightarrow af,$
- $f \rightarrow af, f \rightarrow bp, f \rightarrow \varepsilon,$
- $s \rightarrow af, s \rightarrow bp, s \rightarrow \varepsilon.$

- Ricavare espressione regolare per L(A) (evidenziando metodo)



Ricavare un'espressione regolare per $L(A)$:

Soluzione: Usando la tecnica data nella dimostrazione del Teorema di Kleene, che afferma l'equivalenza tra le espressioni regolari e gli automi a stati finiti, si può impostare il seguente sistema di equazioni di linguaggi dove $L(A)$ è l'incognita X_0 :

$$\begin{cases} X_0 = aX_3 + bX_2 \\ X_1 = aX_2 + bX_0 \\ X_2 = aX_2 + bX_2 \\ X_3 = aX_3 + bX_1 + \varepsilon \end{cases}$$

Ognuna di queste equazioni deve essere risolta sfruttando l'equazione tipo $X = AX + B$, la cui soluzione è $X = A^*B$. Partiamo dall'equazione $X_2 = aX_2 + bX_2$, allora la sua soluzione è $X_2 = \emptyset$. Inserendo tale valore nell'equazione due si ottiene $X_1 = a\emptyset + bX_0$. Ora sostituisco il valore trovato di X_1 nell'equazione quattro, si ha $X_3 = aX_3 + bbX_0 + \varepsilon$, da cui $X_3 = a^*(bbX_0 + \varepsilon)$. Ora possiamo sostituire il valore di X_3 nell'equazione uno:

$$X_0 = a^+(bbX_0 + \varepsilon) + b\emptyset = a^+bbX_0 + a^+.$$

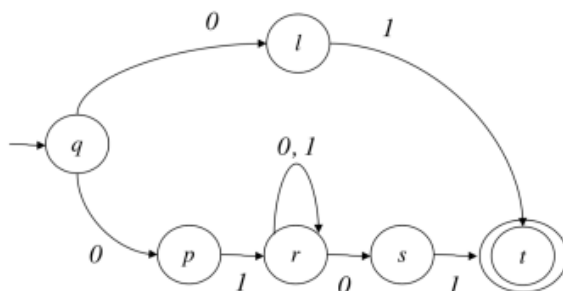
Applicando ancora una volta l'equazione tipo si ottiene:

$$X_0 = (a^+bb)^*a^+ = L(A).$$

- Disegnare automa che riconosce una determinata parola

Disegnare un automa a stati finiti che riconosce le parole su $\{0,1\}$ con prefisso e suffisso 01. Non è richiesto che l'automata sia deterministico.

Soluzione: La scelta di un automa nondeterministico semplifica l'esercizio. Prima di dare il diagramma degli stati dell'automata richiesto, si noti che oltre alle parole denotate dall'espressione regolare $01\{0,1\}^*01$, l'automata deve riconoscere anche la semplice parola 01. Infatti, per definizione di prefisso e suffisso anche 01 ha prefisso e suffisso 01. L'automata richiesto è il seguente:



- Definizione formale di NFA e DFA

Definizione Formale di un NFA

Un **Automa a Stati Finiti Non Deterministico** (NFA) è una quintupla $(Q, \Sigma, \delta, q_0, F)$ dove:

1. Q è un insieme finito di stati.
2. Σ è un insieme finito chiamato alfabeto.
3. $\delta : Q \times \Sigma \rightarrow 2^Q$ è la funzione di transizione. $\delta(q, a)$ restituisce un insieme di stati, cioè per ogni stato $q \in Q$ e simbolo $a \in \Sigma$, $\delta(q, a)$ è un sottoinsieme di Q .
4. $q_0 \in Q$ è lo stato iniziale.
5. $F \subseteq Q$ è l'insieme degli stati finali (o accettanti).

Definizione Formale di un DFA

Un **Automa a Stati Finiti Deterministico** (DFA) è una quintupla $(Q, \Sigma, \delta, q_0, F)$ dove:

1. Q è un insieme finito di stati.
2. Σ è un insieme finito chiamato alfabeto.
3. $\delta : Q \times \Sigma \rightarrow Q$ è la funzione di transizione. $\delta(q, a)$ restituisce un singolo stato, cioè per ogni stato $q \in Q$ e simbolo $a \in \Sigma$, $\delta(q, a)$ è un elemento di Q .
4. $q_0 \in Q$ è lo stato iniziale.
5. $F \subseteq Q$ è l'insieme degli stati finali (o accettanti).

Differenze Chiave tra NFA e DFA

- **Determinismo:**
 - In un DFA, per ogni stato q e simbolo a dell'alfabeto Σ , c'è esattamente una transizione definita, ovvero $\delta(q, a)$ restituisce un singolo stato.
 - In un NFA, per ogni stato q e simbolo a dell'alfabeto Σ , ci può essere zero, una o più transizioni definite, ovvero $\delta(q, a)$ restituisce un insieme di stati.
- **Funzione di Transizione:**
 - La funzione di transizione δ di un DFA mappa ogni coppia (stato, simbolo) a un singolo stato.
 - La funzione di transizione δ di un NFA mappa ogni coppia (stato, simbolo) a un insieme di stati.

- Definire formalmente \approx (indistinguibilità) e l'automa A_{\approx} equivalente ad A , corollario

stati **indistinguibili** \approx : lo stato raggiunto nell'automa, partendo da q_1 , leggendo una qualsiasi parola w deve essere obbligatoriamente uguale allo stato raggiunto nell'automa, partendo da q_2 , leggendo la medesima parola w , questo deve accadere per tutte le parole appartenenti a Σ^*

Dato un automa deterministico a stati finiti $A = (\Sigma, Q, q_0, \delta, F)$, possiamo costruire un nuovo automa A_{\approx} in funzione della relazione di **indistinguibilità** tra stati, denotata con \approx .

Relazione di Indistinguibilità \approx

Due stati $p, q \in Q$ si dicono **indistinguibili** (ovvero $p \approx q$) se, per ogni stringa $w \in \Sigma^*$, la macchina si comporta in modo indistinguibile nei due stati rispetto all'appartenenza a F .

Formalmente:

$$p \approx q \iff \forall w \in \Sigma^*, \delta^*(p, w) \in F \iff \delta^*(q, w) \in F$$

Qui, $\delta^*(p, w)$ indica lo stato raggiunto partendo da p e leggendo la stringa w .

Costruzione dell'Automa Quotiente A_{\approx}

L'automa A_{\approx} è costruito come segue:

- **Alfabeto:** Σ , lo stesso di A .
- **Stati:** Gli stati dell'automa A_{\approx} sono le classi di equivalenza di Q rispetto alla relazione \approx , denotate con $[q]_{\approx}$ per ogni stato $q \in Q$.
- **Stato iniziale:** La classe di equivalenza dell'automa A_{\approx} è $[q_0]_{\approx}$.
- **Funzione di transizione:** Definiamo una funzione di transizione indotta sulle classi di equivalenza:

$$\delta_{\approx}([q]_{\approx}, a) = [\delta(q, a)]_{\approx} \quad \text{per ogni } a \in \Sigma$$

- **Stati finali:** L'insieme degli stati finali di A_{\approx} è $F_{\approx} = \{[q]_{\approx} \mid q \in F\}$.

In altre parole, A_{\approx} è l'automa dove ogni stato rappresenta una classe di stati indistinguibili di A .

Il corollario principale che coinvolge A_{\approx} è il **Corollario di Minimizzazione degli Automi Finite Deterministici**:

Corollario: Minimalità di A_{\approx}

L'automa A_{\approx} costruito a partire da A rispetto alla relazione di indistinguibilità \approx è l'automa **minimale** equivalente ad A . Cioè:

- A_{\approx} riconosce lo stesso linguaggio di A .
- Non esiste un DFA con un numero inferiore di stati che riconosce lo stesso linguaggio di A_{\approx} .

In altre parole, A_{\approx} è il DFA più piccolo (con il minor numero di stati) che è equivalente a A dal punto di vista del linguaggio riconosciuto. Questo corollario è la base della procedura di minimizzazione degli automi deterministici.

Riconoscitore a Pila

- Definizione formale (prestare attenzione a δ)

Un **riconoscitore a pila** (o **pushdown automaton**, PDA) è un modello computazionale utilizzato per riconoscere linguaggi che vanno oltre quelli che possono essere riconosciuti da automi a stati finiti, in particolare i linguaggi liberi dal contesto.

Definizione Formale di un Riconoscitore a Pila

Un riconoscitore a pila è una sestupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, dove:

1. Q è un insieme finito di stati.
2. Σ è un alfabeto finito di simboli di input.
3. Γ è un alfabeto finito di simboli della pila.
4. δ è la funzione di transizione, definita come:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q \times \Gamma^*)$$

La funzione di transizione δ mappa una tripla (stato corrente, simbolo di input, simbolo di pila) a un insieme di coppie (nuovo stato, nuova stringa di simboli di pila).

5. $q_0 \in Q$ è lo stato iniziale.
6. $F \subseteq Q$ è l'insieme degli stati finali (o accettanti).

Interpretazione della Funzione di Transizione

La funzione di transizione δ può essere interpretata come segue:

- $\delta(q, a, X)$ fornisce un insieme di coppie (p, γ) , dove p è un nuovo stato e γ è una stringa di simboli della pila che sostituisce il simbolo X in cima alla pila.
- a può essere un simbolo dell'alfabeto di input Σ o ϵ (che rappresenta una transizione epsilon, cioè una transizione che può essere eseguita senza leggere un simbolo di input).
- X può essere un simbolo dell'alfabeto della pila Γ o ϵ (che rappresenta una transizione che non dipende dal simbolo in cima alla pila).

- Dare il grafo di computazione di A per la parola X

Consideriamo un PDA molto semplice che riconosce esattamente la stringa "Cico".

Supponiamo che l'alfabeto di input sia $\Sigma = \{C, i, c, o\}$.

Definizione del PDA


Un PDA che riconosce la stringa "Cico" può essere definito come segue:

- **Stati:** $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- **Alfabeto di input:** $\Sigma = \{C, i, c, o\}$
- **Alfabeto della pila:** $\Gamma = \{\perp\}$
- **Stato iniziale:** q_0
- **Stati finali:** $F = \{q_4\}$
- **Funzione di transizione:**
 - $\delta(q_0, C, \epsilon) = \{(q_1, \epsilon)\}$
 - $\delta(q_1, i, \epsilon) = \{(q_2, \epsilon)\}$
 - $\delta(q_2, c, \epsilon) = \{(q_3, \epsilon)\}$
 - $\delta(q_3, o, \epsilon) = \{(q_4, \epsilon)\}$

Questo PDA accetta esattamente la stringa "Cico" passando attraverso una sequenza di stati, uno per ogni simbolo della stringa.

Ecco una rappresentazione del grafo:

SCSS

 Copia codice

```
(q_0) -- C --> (q_1) -- i --> (q_2) -- c --> (q_3) -- o --> (q_4)
```

Descrizione dei Passaggi

- Il PDA inizia nello stato q_0 .
- Legge "C", transita allo stato q_1 .
- Legge "i", transita allo stato q_2 .
- Legge "c", transita allo stato q_3 .
- Legge "o", transita allo stato q_4 , che è uno stato finale.
- Poiché q_4 è uno stato finale e tutti i simboli della stringa sono stati letti, la stringa "Cico" viene accettata dal PDA.

- Ricavare un riconoscitore a pila per il linguaggio L

Per costruire un riconoscitore a pila (PDA) per il linguaggio $L = \{ww^R \mid w \in \{0, 1\}^+\}$, dobbiamo progettare un automa che accetti stringhe costituite da una stringa seguita dalla sua inversione.

Descrizione Informale

1. **Prima metà:** Il PDA legge la prima metà della stringa w e la spinge sulla pila.
2. **Seconda metà:** Il PDA legge la seconda metà della stringa w^R e la confronta con i simboli estratti dalla pila.
3. **Accettazione:** La stringa è accettata se tutti i simboli nella seconda metà corrispondono ai simboli della prima metà.

Definizione Formale del PDA

Un PDA per questo linguaggio può essere definito come segue:

- **Stati:** $Q = \{q_0, q_1, q_2, q_f\}$
- **Alfabeto di input:** $\Sigma = \{0, 1\}$
- **Alfabeto della pila:** $\Gamma = \{0, 1, \perp\}$
- **Stato iniziale:** q_0
- **Stati finali:** $F = \{q_f\}$
- **Simbolo iniziale della pila:** \perp
- **Funzione di transizione δ :**

Funzione di Transizione

La funzione di transizione δ è definita come segue:

1. **Stato iniziale q_0 :**
 - $\delta(q_0, 0, \epsilon) = \{(q_0, 0)\}$
 - $\delta(q_0, 1, \epsilon) = \{(q_0, 1)\}$
 - $\delta(q_0, \epsilon, \epsilon) = \{(q_1, \epsilon)\}$ (transizione epsilon per passare allo stato successivo)
2. **Stato q_1 :**
 - $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$
 - $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$
 - $\delta(q_1, \epsilon, \perp) = \{(q_f, \perp)\}$ (transizione epsilon per accettare se la pila è vuota)

Operazione del PDA

1. **Lettura e Push:** In q_0 , il PDA legge i simboli della prima metà della stringa e li spinge sulla pila.
2. **Transizione Epsilon:** Quando raggiunge la metà della stringa, il PDA utilizza una transizione epsilon per passare allo stato q_1 .
3. **Lettura e Pop:** In q_1 , il PDA legge i simboli della seconda metà della stringa e li confronta con i simboli in cima alla pila. Ogni simbolo letto deve corrispondere al simbolo in cima alla pila, che viene quindi rimosso.
4. **Accettazione:** Se la pila è vuota (contiene solo il simbolo \perp) e tutti i simboli della seconda metà corrispondono, il PDA passa allo stato q_f e accetta la stringa.

TEOREMI

- Teorema di inclusione degli R_k (+ giustificare perché valgono le inclusioni)

Teorema di Inclusione degli R_k Formale

Un enunciato formale del teorema può essere il seguente:

Teorema: Siano R_k e R_{k+1} due classi di complessità temporale o spaziale tali che le risorse $f_k(n)$ per R_k e $f_{k+1}(n)$ per R_{k+1} soddisfano $f_{k+1}(n) \geq f_k(n)$ per ogni n . Allora:

$$R_k \subseteq R_{k+1}$$

Dimostrazione:

1. Per ogni problema L appartenente a R_k , esiste una macchina di Turing M_k che decide L usando $f_k(n)$ risorse.
2. Poiché $f_{k+1}(n) \geq f_k(n)$, la macchina M_k può essere eseguita con $f_{k+1}(n)$ risorse senza superare i limiti imposti da R_{k+1} .
3. Quindi, ogni problema risolvibile con risorse $f_k(n)$ è anche risolvibile con risorse $f_{k+1}(n)$.
4. Di conseguenza, $R_k \subseteq R_{k+1}$.

Questo teorema ci dice che man mano che aumentiamo le risorse disponibili (che siano tempo, spazio, o altre misure di complessità), possiamo risolvere almeno tanti problemi quanti ne potevamo risolvere con risorse più limitate.

- Pumping Lemma

Enunciato del Pumping Lemma per i Linguaggi Liberi dal Contesto

Sia L un linguaggio libero dal contesto. Allora esiste un numero $p \geq 1$ (detto costante di pumping) tale che ogni stringa z in L con $|z| \geq p$ può essere suddivisa in cinque parti $uvwxy$ in modo che:

1. $z = uvwxy$
2. $|vwx| \leq p$
3. $|vx| \geq 1$
4. Per ogni $i \geq 0$, la stringa $uv^iwx^iy \in L$