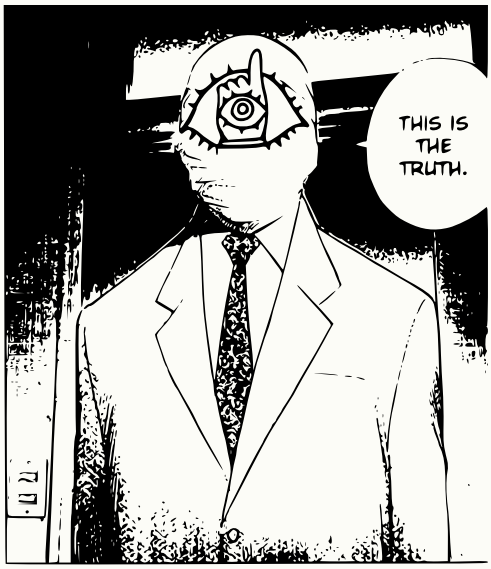
Marco Casu

Automi, Calcolabilità e Complessità

Sapienza
Università di Roma

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Questo documento è distribuito sotto la licenza GNU, è un resoconto degli appunti (eventualmente integrati con libri di testo) tratti dalle lezioni del corso di Automi, Calcolabilità e Complessità per la laurea triennale in Informatica. Se dovessi notare errori, ti prego di segnalarmeli.



INDICE

CAPITOLO

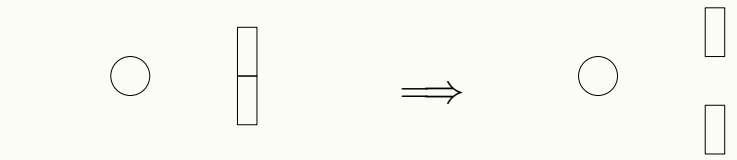
1

AUTOMI

1.1 Linguaggi Regolari

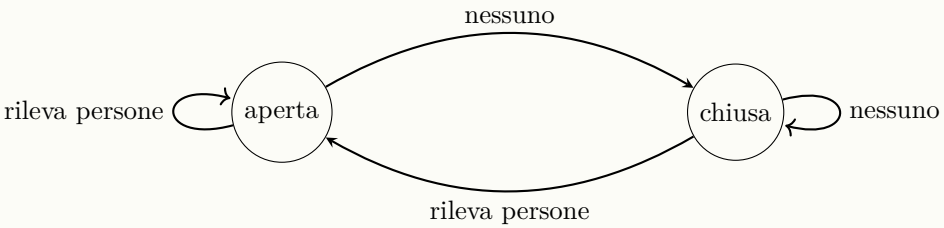
Un *automa a stati finiti* è, seppure limitato nella memoria e nella gestione dell'input, il più semplice modello di computazione. Un automa può interagire con l'input esclusivamente "scorrendolo" in maniera sequenziale.

Esempio : Si vuole modellare una semplice porta con sensore, che si apre quando qualcuno si trova nelle vicinanze.



una persona si trova davanti la porta — la porta si apre

Un automa che modella il problema è il seguente :



Un automa ha alcuni stati speciali, come quello iniziale, indicato con un apposita freccia, e degli stati detti *di accettazione*, ossia stati in cui deve necessariamente terminare la computazione per essere definita valida, vengono rappresentati con un doppio cerchio.

Il modello di calcolo degli automi è riconducibile al concetto di *linguaggio regolare*, che verrà formalizzato in seguito, segue ora una definizione formale di automa.

Definizione (DFA) : : Un DFA (Deterministic Finite Automa) è una 5-tupla, $(Q, \Sigma, \delta, q_0, F)$ di cui

- Q è l'insieme degli stati possibili
- Σ è l'alfabeto che compone le stringhe in input

- δ è una mappa $Q \times \Sigma \to Q$ detta *funzione di transizione.*
- $q_0 \in Q$ è lo stato iniziale.
- $F \subseteq Q$ è l'insieme degli stati di accettazione.

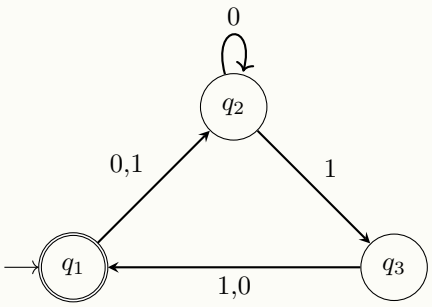


Figura 1.1: semplice automa

Nell'esempio in figura 1.1, si ha che

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $F = \{q_1\}$
- $q_0 = q_1$
- $\delta =$

	0	1
q_1	q_2	q_2
q_2	q_2	q_3
q_3	q_1	q_1

Sia D un DFA, chiamiamo **linguaggio dell'automa**, e denotiamo $L(D)$, l'insieme delle stringhe che date in input a D fanno si che D termini su uno stato di accettazione. Per definire formalmente un linguaggio

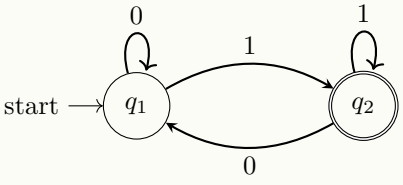


Figura 1.2: il linguaggio di tale automa risulta essere composto dalle stringhe che terminano con 1

di un automa, è necessario introdurre la **funzione di transizione estesa**:

$$\delta^*(q, \epsilon) = \delta(q, \epsilon)$$

$$\delta^*(q, ax) = \delta^*(\delta(q, a), x)$$

dove

$$a \in \Sigma, \quad x \in \Sigma^*, \quad \epsilon = \text{ stringa vuota}$$

Σ^* è l'insieme di tutte le stringhe formate dall'alfabeto Σ. Passiamo ora alla definizione di **configurazione**, essa rappresenta lo stato dell'automa ad un certo punto della computazione, essa è formata da una coppia

$$Q \times \Sigma^*$$

Rappresentante uno stato, ed una stringa di input rimanente da computare.

Un **passo della computazione** in un automa rappresenta una transizione da una configurazione ad un altra, è una relazione binaria $\vdash_D$: $Q \times \Sigma^*$ tale che

$$(p, ax) \vdash_D (q, x) \iff \delta(p, a) = q \quad \text{dove} \quad p, q \in Q, \;\; a \in \Sigma, \;\; x \in \Sigma^*$$

Si può estendere la definizione di passo di computazione, considerando la sua *chiusura transitiva* $\vdash_D^*$. Essa si ottiene aggiungendo a $\vdash_D$ tutte le coppie in $Q \times \Sigma^*$ che rendono la relazione chiusa rispetto la riflessività e rispetto la transitività.

$$(q, aby) \vdash_D (p, by) \;\wedge\; (p, by) \vdash_D (ry) \implies (q, aby) \vdash_D^* (r, y)$$

Ad esempio, nell'automa in figura 1.2, risulta chiaro che

$$\begin{cases} (q_1, 011) \vdash_D (q_1, 11) \\ (q_1, 11) \vdash_D (q_2, 1) \\ (q_2, 1) \vdash_D (q_2, \epsilon) \end{cases} \implies (q_1, 011) \vdash_D^* (q_2, \epsilon)$$

Inoltre

$$\begin{gathered} \delta^*(q_1, 011) = \\ \delta^*(q_1, 11) = \\ \delta^*(q_2, 1) = \\ \delta^*(q_2, \epsilon) = q_2 \end{gathered}$$

Se non specificato diversamente, con ϵ verrà indicata la stringa vuota. Utilizzando le precedenti definizioni, è possibile definire formalmente quali sono gli input accettati da un DFA.

Definizione : : Sia $D = (Q, \Sigma, \delta, q_0, F)$ un DFA, e sia $x \in \Sigma^*$ una stringa, essa è **accettata** da D se

$$\delta^*(q_0, x) \in F$$

Il **linguaggio riconosciuto** da D è

$$L(D) = \{x \in \Sigma^* | \delta^*(q_0, x) \in F\}$$

Definizione (Linguaggi Regolari) : : L'insieme dei linguaggi regolari, denotato REG, contiene tutti i linguaggi, tali che esiste un DFA che li ha come linguaggi riconosciuti.

$$REG = \{L \mid \exists D = (Q, \Sigma, \delta, q_0, F) \texttt{ t.c. } L \in \Sigma^* \wedge L(D) = L\}$$

Uno fra gli scopi di questo corso riguarda il capire come progettare automi, e capire se, ogni linguaggio è regolare, o ce ne sono alcuni che non possono essere riconosciuti da alcun possibile DFA.

1.1.1 Esempi di DFA

Vediamo in questa sezione alcuni semplici esempi di DFA.

Esempio 1) Si vuole progettare un DFA che accetti il seguente linguaggio

$$\{x \in \{0, 1\}^* \mid w_h(x) \geq 3\}$$

Si ricordi come

$$w_h(x) = \text{ occorrenze di 1 in } x$$

Una volta progettato il DFA, è anche importante dimostrarne la correttezza, ossia dare una prova matematico che l'automa in questione accetti il linguaggio.

- Se $x \in L(D)$ allora D accetta x
- Se D accetta x allora $w_h(x) \geq 3$

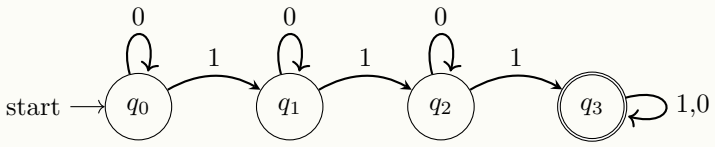


Figura 1.3: Esempio (1) di DFA

In questo, e nei seguenti casi, essendo i DFA estremamente semplici, risulta ovvio che accettino il dato linguaggio, in casi più avanzati, sarà necessario fornire una dimostrazione rigorosa.

Esempio 2) Si vuole progettare un DFA che accetti il seguente linguaggio

$$\{x \in \{0,1\}^* \mid x = 1y \wedge y \in \{0,1\}^*\}$$

Appunto sulla notazione : Se $a \in \Sigma^*$ e $b \in \Sigma^*$, allora con ab si denota la concatenazione di stringhe.

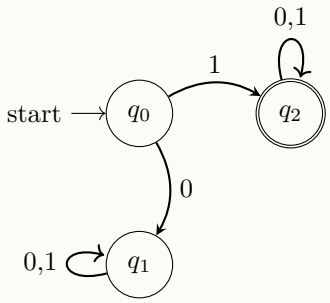


Figura 1.4: Esempio (2) di DFA

Nell'esempio (2), quando dallo stato q_0 il DFA riceve in input 0, la computazione cade su uno stato "buco nero", dalla quale non si può uscire a prescindere dall'input, l'operazione che fa cadere in questo stato è da considerarsi "non definita" in quanto non porterà mai la computazione a terminare su uno stato accettabile, è quindi comodo rimuovere tale stato dal diagramma.

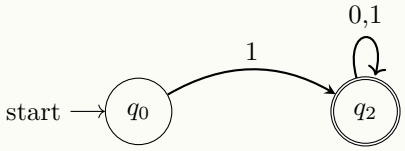


Figura 1.5: Esempio (2.1) di DFA

Anche in questo caso la dimostrazione della correttezza risulta banale.

Esempio 3) Si vuole progettare un DFA che accetti il seguente linguaggio

$$\{x \in \{0,1\}^* \mid x = 0^n 1, \;\; n \in \mathbb{N}\}$$

Con $0^n 1$ si intende una stringa che sia composta esclusivamente da 0, ma con un 1 come ultimo termine, ad esempio :

0000000000000001

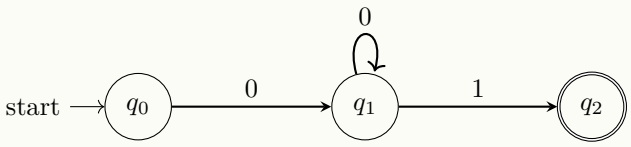


Figura 1.6: Esempio (3) di DFA

1.2 Operazioni sui Linguaggi

Lo studio delle proprietà dei linguaggi regolari può fornire opportune accortezze utili nella progettazione di automi, siccome i linguaggi sono insiemi di stringhe costruiti su un alfabeto Σ, essi godono delle operazioni insiemistiche.

Risulta utile definire formalmente la concatenazione fra stringhe, siano

$$x = a_1, a_2 \ldots, a_n \qquad y = b_1, b_2 \ldots, b_n$$

due stringhe, esse possono essere concatenate

$$xy = a_1, a_2 \ldots, a_n, b_1, b_2 \ldots, b_n$$

L'operazione di concatenazione non è commutativa, può essere definita ricorsivamente in tal modo :

$$x(ya) = (xy)a$$

dove

$$x, y \in \Sigma^* \qquad a \in \Sigma$$

Siano L_1, L_2 due linguaggi regolari in REG (per semplicità, definiti su uno stesso alfabeto Σ), e sia n un numero naturale, sono definite su di essi le seguenti operazioni :

- **unione** : $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$
- **intersezione** : $L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$
- **complemento** : $\neg L_1 = \{x \in \Sigma^* \mid x \notin L_1\}$
- **concatenazione** : $L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
- **potenza** : $L_1^n = \underbrace{L_1 \circ L_1 \circ L_1, \cdots \circ L_1}_{n \text{ volte}}$
- **star** : $L_1^* = \{x_1, x_2 \ldots, x_k \mid k \in \mathbb{Z}^+ \wedge x_i \in L_1\}$
  Si può definire anche diversamente
  $$L_1^* = \bigcup_{k=0}^{\infty} L_1^k$$

Esempio di concatenazione e potenza :

$$\Sigma = \{a, b\} \quad L_1 = \{a, ab, ba\} \quad L_2 = \{ab, b\} \quad L = \{a, ab, ba\}$$

$$L_1 \circ L_2 = \{aab, ab, abab, abb, baab, bab\}$$

$$L^2 = \{aa, aab, aba, abab, abba, baa, baba\}$$

Teorema (Chiusura di REG) : La classe dei linguaggi regolari REG, è chiusa rispetto a tutte le operazioni appena elencate, siano L_1 ed L_2 due linguaggi regolari, allora :

$$L_1 \cup L_2 \in REG \qquad L_1 \circ L_2 \in REG \qquad L_1 \cap L_2 \in REG$$

$$L_1^n \in REG \qquad \neg L_1 \in REG \qquad L_1^* \in REG$$

Dimostrazione (unione ed intersezione) : Siano L_1 ed L_2 due linguaggi regolari, considero due DFA, per semplicità, con lo stesso alfabeto

$$D_1 = (Q_1, \Sigma, \delta, q_1, F_1)$$

$$D_2 = (Q_2, \Sigma, \delta, q_2, F_2)$$

tali che

$$L(D_1) = L_1 \ \wedge \ L(D_2) = L_2$$

Si costruisce un DFA che simula contemporaneamente l'esecuzione di D_1 e D_2, in cui gli stati possibili saranno le possibili combinazioni di coppie di stati. Si definisce $D = (Q, \Sigma, \delta, q_0, F)$ tale che

- $Q = Q_1 \times Q_2 = \{(r_1, r_2) \| \ r_1 \in Q_1 \wedge r_2 \in Q_2\}$
- $\delta((r_1, r_2), a) = (\delta(r_1, a), \delta(r_2, a))$ dove $a \in \Sigma$ e $(r_1, r_2) \in Q$
- $q_0 = (q_1, q_2)$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(r_1, r_2) \| \ r_1 \in F_1 \vee r_2 \in F_2\}$

Nel caso si dovesse dimostrare la proprietà dell'intersezione, si avrebbe che

- $F = F_1 \times F_2$

A questo punto risulta chiaro che

(i) $x \in L_1 \cup L_2 \implies x \in L(D)$

(ii) $x \in L(D) \implies x \in L_1 \cup L_2$ ■

Dimostrazione (complemento) : : Sia L un linguaggio regolare, e D un automa che lo accetta $L(D) = L$. Si vuole dimostrare che esiste un automa che accetti $L^C = \{w \in \Sigma^* \mid w \notin L\}$. Essendo

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

considero

$$D' = (Q, \Sigma, \delta, q_0, F^C)$$

dove $F^C = Q \backslash F$. Supponiamo che $w \in L^C$, allora sicuramente, se data come input a D', la computazione terminerà in uno stato che non è in F, dato che, per definizione, se terminasse in F, sarebbe accettato da D, ma L^C contiene tutte le stringhe che non sono accettate da D, quindi

- $w \in L^C$
- la computazione termina in uno stato $q \in F^C$
- D' accetta w
- L^C è un linguaggio regolare. ■

Per dimostrare la proprietà di concatenazione, è necessario introdurre un nuovo concetto.

1.3 Non Determinismo

Si può generalizzare il modello di DFA, in modo tale che la lettura di un input non scaturisca il passaggio da uno stato ad un'altro, ma da uno stato ad un insieme di stati, tale generalizzazione è detta *Non-Deterministic Finite Automa.*

Definizione (NFA) : : Un NFA è una tupla $N = (Q, \Sigma, \delta, q_0, F)$ tale che

- Q, Σ, q_0, F condividono la definizione con i loro corrispettivi nel DFA
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$

- $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$
- $\mathcal{P}(Q)$ è l'insieme delle parti di Q

Una computazione in un NFA è paragonabile ad una computazione parallela, in cui un input può risultare in diversi *rami* di computazione. Una funzione di transizione di un NFA inoltre accetta la stringa vuota ϵ, se la computazione finisce in uno stato in cui è presente un arco di questo tipo, verrà considerata anche una diramazione verso quell'arco a prescindere dall'input.

Una computazione si può rappresentare graficamente con un albero, se una delle diramazioni possibili termina in uno stato accettabile, allora la stringa in input è accettata.

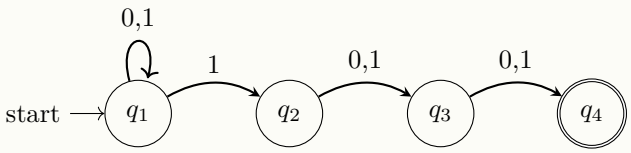


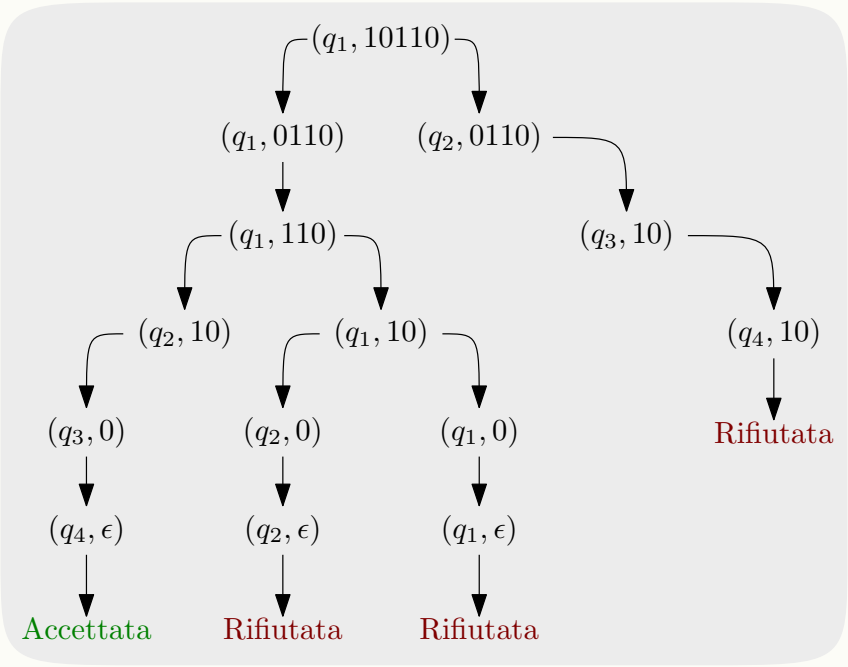
Figura 1.7: Esempio di NFA

Sia N l'NFA mostrato in figura 1.7, si ha

$$L(N) = \{x \in \{0,1\}^* \| \ x \text{ ha 1 come terzultimo valore } \}$$

Si può visualizzare il seguente albero di computazione data come input una stringa w :

$$w = 10110$$



Essendo che la traccia di sinistra accetta w, allora N accetta w.

È necessario estendere il concetto di *configurazione* per gli NFA, essa, rappresentante uno stato della computazione, sarà una coppia

$$(q, x) \in Q \times \Sigma_\epsilon$$

E diremo che

$$(p, ax) \vdash_N (q, x) \iff q \in \delta(p, a)$$

dove

$$p, q \in Q \quad a \in \Sigma_\epsilon \quad x \in \Sigma_\epsilon^*$$

Si considera ora la chiusura transitiva di $\vdash_N$, denotata $\vdash_N^*$, se w è una stringa ed N un NFA, si ha che

$$w \in L(N) \iff \exists q \in F \| \ (q_0, w) \vdash_N^* (q, \epsilon)$$

Consideriamo adesso l'unione di due NFA, risulta particolarmente semplice da definire, siano N_1 e N_2 due NFA, che per semplicità, condividono l'alfabeto

$$N_1 = \{Q_1, \Sigma_\epsilon, \delta_1, q_1, F_1\}$$

$$N_2 = \{Q_2, \Sigma_\epsilon, \delta_2, q_2, F_2\}$$

Definisco un nuovo NFA $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ tale che

- $Q = Q_1 \cup Q_2$
- $F = F_1 \cup F_2$
- Siano $q \in Q$ e $a \in \Sigma_\epsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) \text{ se } q \in Q_1 \\ \delta_2(q,a) \text{ se } q \in Q_2 \\ \{q_1, q_2\} \text{ se } q = q_0 \ \wedge \ a = \epsilon \\ \emptyset \text{ se } q = q_0 \ \wedge \ a \neq \epsilon \end{cases}$$

Si avrà che $L(N) = L(N_1) \cup L(N_2)$

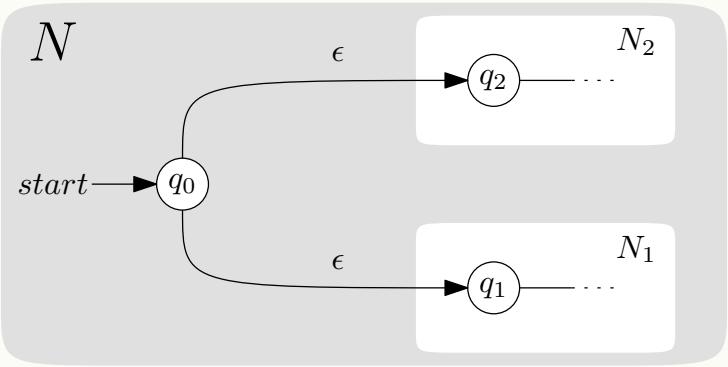


Figura 1.8: Unione di due NFA

Denotiamo $\mathcal{L}(DFA)$ l'insieme dei linguaggi accettati da un qualsiasi DFA, che per definizione è REG, e denotiamo, in maniera analoga $\mathcal{L}(NFA)$.

Teorema : : L'insieme dei linguaggi accettati da un qualsiasi DFA, e quello dei linguaggi accettati da un qualsiasi NFA coincidono

$$\mathcal{L}(DFA) = \mathcal{L}(NFA) = REG$$

Dimostrazione : Il caso $\mathcal{L}(DFA) \subseteq \mathcal{L}(NFA)$ è banale e non verrà dimostrato. Si vuole dimostrare che se L è accettato da un generico NFA, allora esiste un DFA che accetta L, l'idea è quella di "simulare" un NFA tramite un DFA che rappresenti ogni possibile stato di computazione.

Sia $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$ un NFA, e sia $L = L(N)$. Si considera un DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$ tale che

- $Q_D = \mathcal{P}(Q_N)$
- $q_{0_D} = \{q_{0_N}\}$
- $F_D = \{R \in Q_D | R \cap F_N \neq \emptyset\}$, ovvero, D accetta tutti gli insiemi in cui compare almeno un elemento accettato da N.

- Sia $R \in Q_D$ e $a \in \Sigma$, si definisce $\delta_D(R, a) = \bigcup_{r \in R} \delta_N(r, a)$

Questo caso non tiene conto di un NFA in cui sono presenti degli ϵ-archi. Supponiamo che vi siano, sia $R \in Q_D$, si definisce la funzione estesa E definita come segue

$$E(R) = \{q \in Q_N \| \ q \text{ può essere raggiunto da un qualsiasi stato } r \in R \text{ attraverso zero o più } \epsilon\text{-archi } \}$$

Cambia la definizione del DFA utilizzato per la dimostrazione

- $q_{0_D} = E(\{q_{0_N})\}$
- $\delta_D(R, a) = \bigcup_{r \in R} E(\delta_N(r, a))$

È chiaro che D tiene traccia di tutte le possibili computazioni di N, ed accetta L, ossia $L(D) = L(N)$. ■

Esempio : Si consideri l'NFA N definito come segue

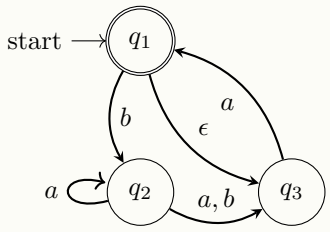


Figura 1.9: $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$

Si costruisce un DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$ con le seguenti specifiche (per comodità, l'elemento $\{q_i, q_j \ldots, q_k\}$ verrà denotato $p_{ij\ldots k}$), mostrato in figura 1.10

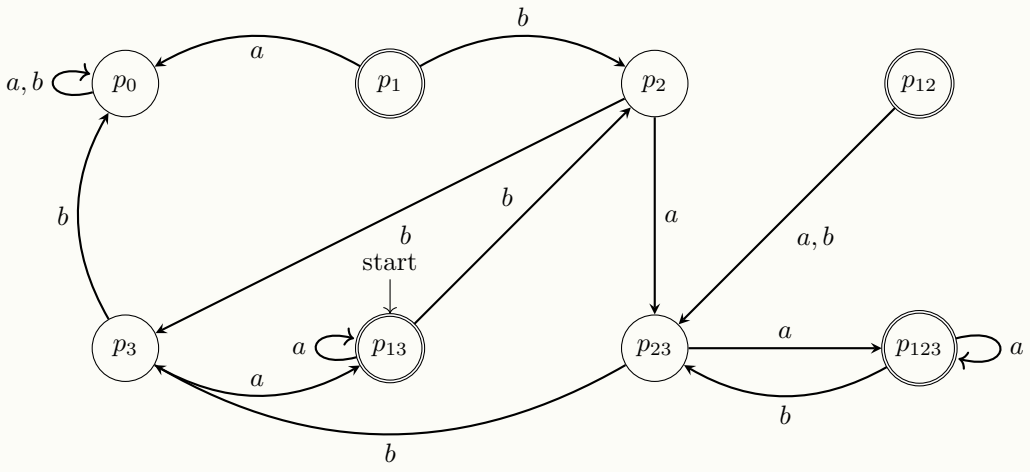


Figura 1.10: grafico di D

- $Q_D = \{p_0, p_1, p_2, p_3, p_{12}, p_{13}, p_{23}, p_{123}\}$
- $E(q_{0_N}) = \{q_1, q_3\} \implies q_{0_D} = p_{13}$
- $F_D = \{p_1, p_{12}, p_{13}, p_{123}\}$
- la funzione δ_D si definisce osservando il grafico di N
  - $\delta_N(q_2, a) = \{q_2, q_3\} \implies \delta_D(p_2, a) = p_{23}$
  - $\delta_N(q_2, b) = \{q_3\} \implies \delta_D(p_2, a) = p_3$

- $\delta_N(q_1, a) = \emptyset \implies \delta_D(p_1, a) = p_0$
- $\delta_N(q_1, b) = \{q_2\} \implies \delta_D(p_1, b) = p_2$
- etc...

L'introduzione degli automi non deterministici è stata necessaria in principio per la dimostrazione della chiusura di *REG* rispetto le operazioni di concatenazione e star.

Teorema : *REG* è chiusa per concatenazione.

Dimostrazione : Siano L_1 ed L_2 due linguaggi regolari, esistono quindi due NFA

$$N_1 = (Q_1, \Sigma_\epsilon, \delta_1, q_0^1, F_1) \qquad N_2 = (Q_2, \Sigma_\epsilon, \delta_2, q_0^2, F_2)$$

tali che $L(N_1) = L_1 \ \wedge \ L(N_2) = L_2$. Si costruisce un NFA $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$, l'idea è quella di concatenare le ramificazioni di N_1 ad N_2.

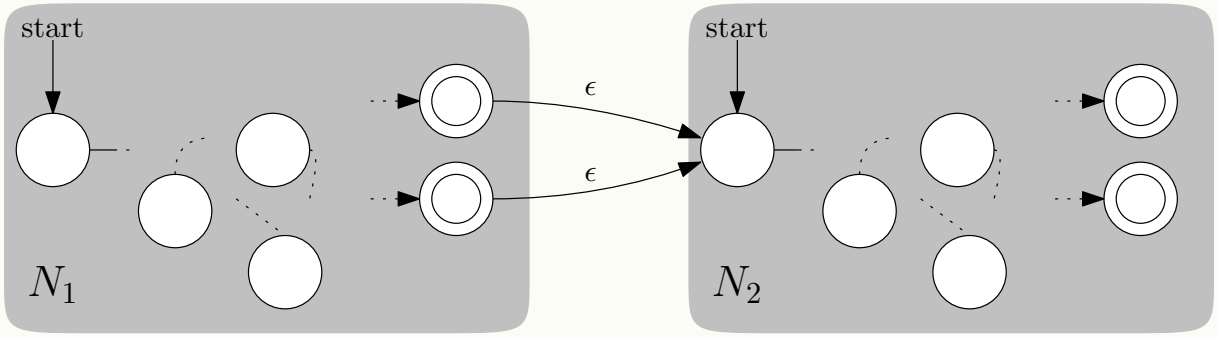


Figura 1.11: schema di N

Un NFA di questo tipo computerà una stringa in L_1, se finirà in uno stato di F_1, andrà nello stato iniziale di N_2, è chiaro che l'automa accetta solamente una concatenazione di stringhe fra L_1 ed L_2.

- $Q = Q_1 \cup Q_2$
- $q_0 = q_0^1$
- $F = F_2$
- per $a \in \Sigma_\epsilon$ e $q \in Q$ si ha $\delta(q, a) = \begin{cases} \delta_1(q, a) \text{ se } q \in Q_1 \wedge q \notin F_1 \\ \delta_1(q, a) \text{ se } q \in F_1 \wedge a \neq \epsilon \\ \delta_1(q, a) \cup \{q_0^2\} \text{ se } q \in F_1 \wedge a = \epsilon \\ \delta_2(q, a) \text{ se } q \in Q_2 \end{cases}$

Si ha quindi che $L(N) = L_1 \circ L_2$. ■

Teorema : *REG* è chiusa per star.

Dimostrazione : Sia $L \in REG$ e sia $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ un NFA tale che $L(N) = L$. Considero un NFA $N^* = (Q^*, \Sigma_\epsilon, \delta^*, q_0^*, F^*)$, identico ad N, con opportune modifiche, lo stato q_0 iniziale di N non è iniziale in N^*, ed ogni stato finale ha una ϵ-arco verso q_0.

- $Q^* = Q \cup \{q_0^*\}$
- q_0^* è un nuovo stato
- $F^* = F \cup \{q_0^*\}$ questo perché in L^* è presente la stringa vuota

- per $a \in \Sigma_\epsilon$ e $q \in Q^*$ si ha $\delta^*(q,a) = \begin{cases} \delta(q,a) \text{ se } q \in Q \wedge q \notin F \\ \delta(q,a) \text{ se } q \in F \wedge a \neq \epsilon \\ \delta(q,a) \cup \{q_0\} \text{ se } q \in F \wedge a = \epsilon \\ \{q_0\} \text{ se } q = q_0^* \wedge a = \epsilon \\ \emptyset \text{ se } q = q_0^* \wedge a \neq \epsilon \end{cases}$

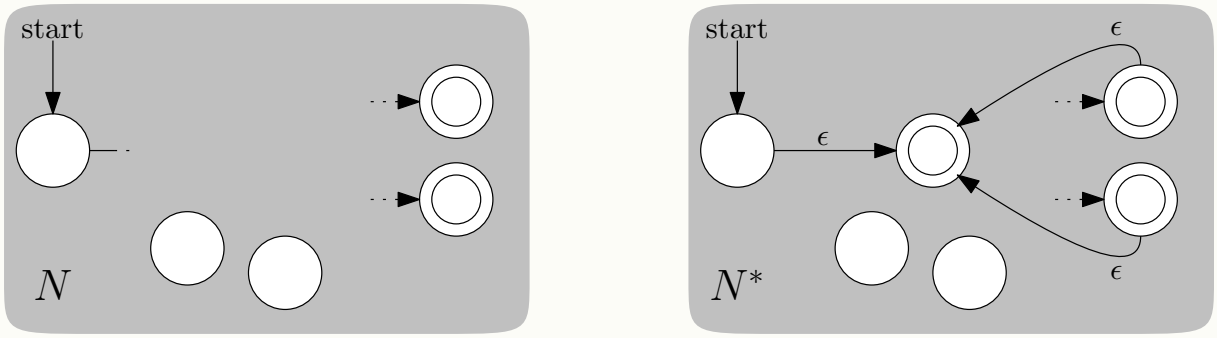


Figura 1.12: schema di N^*

1.4 Espressioni Regolari

Un espressione regolare è simile ad un espressione algebrica ma opera sulle stringhe, dato un alfabeto, un'espressione su tale alfabeto rappresenta un insieme di stringhe, un esempio è

$$(0|1)0^*$$

Dove $(0|1) \equiv \{0\} \cup \{1\} = \{0,1\}$ e $0^* \equiv \{0\}^*$ quindi $(0|1)0^* \equiv \{0,1\} \circ \{0\}^*$.

Definizione (espressione regolare) : : Sia Σ un alfabeto, un espressione regolare r su Σ, denotata $r \in re(\Sigma)$, è definita per induzione

Caso base

- $r = \emptyset \in re(\Sigma)$
- $r = \epsilon \in re(\Sigma)$
- $r = a \in re(\Sigma)$

Caso induttivo

- $r = r_1 \cup r_2$ dove $r_1, r_2 \in re(\Sigma)$
- $r = r_1 \circ r_2$ dove $r_1, r_2 \in re(\Sigma)$
- $r = r_1^*$ dove $r_1 \in re(\Sigma)$

L'insieme delle stringhe definite da $r \in re(\Sigma)$ è il *linguaggio* di r ed è denotato $L(r)$.

Esempio : Sia $\Sigma = \{0,1\}$

- $0^*10^* = \{w \mid w \text{ ha esattamente un } 1\}$
- $\Sigma^*1\Sigma^* = \{w \mid w \text{ ha almeno un } 1\}$
- $\Sigma^*001\Sigma^* = \{w \mid w \text{ ha la sottostringa } 001\}$

Per convenzione si definisce

$$1^*\emptyset = \emptyset \qquad \emptyset^* = \epsilon$$

Teorema Fondamentale : Sia $\mathcal{L}(DFA) = REG$ l'insieme dei linguaggi accettati da un qualsiasi DFA, e sia $\mathcal{L}(re)$ l'insieme dei linguaggi accettati da una qualsiasi espressione regolare, è vero che

$$\mathcal{L}(re) = \mathcal{L}(DFA) = REG$$

Dimostrazione : è necessario dimostrare due direzioni

$\boxed{\mathcal{L}(re) \subseteq \mathcal{L}(DFA)}$: Sia r un espressione regolare, si considera un DFA D_r definito come segue, a seconda dei casi

Caso base

- $r = \emptyset \implies D_r$ non accetta alcuna stringa
- $r = \epsilon \implies D_r$ accetta la stringa vuota
- $r = a \implies D_r$ accetta la $a \in \Sigma$

Caso induttivo

- $r = r_1 \cup r_2$, esistono due automi D_{r_1} e D_{r_2} che accettano rispettivamente $L(r_1)$ e $L(r_2)$, ma allora esiste necessariamente un automa D_r che accetta $L(r_1) \cup L(r_2)$.
- $r = r_1 \circ r_2$, esistono due automi D_{r_1} e D_{r_2} che accettano rispettivamente $L(r_1)$ e $L(r_2)$, ma allora esiste necessariamente un automa D_r che accetta $L(r_1) \circ L(r_2)$.
- $r = r_1^*$, esiste un automa D_{r_1} che accetta $L(r_1^*)$, ma allora esiste necessariamente un automa D_r che accetta $L(r_1)$.

Tali tesi sono vere dato che la classe dei linguaggi regolari è chiusa per le operazioni di star, concatenazione ed unione. □

$\boxed{\mathcal{L}(DFA) \subseteq \mathcal{L}(re)}$: Sia L un linguaggio regolare, e sia N l'NFA tale che $L(N) = L$. Si costruisce un nuovo tipo di NFA che sarà equivalente ad N. Tale automa è detto *GNFA*, dove la G sta per "Generalizzato", tale automa ha una *forma canonica*, ossia, rispetta le seguenti proprietà

- Lo stato iniziale, ha solo archi uscenti
- Vi è un singolo stato finale, ed ha solo archi entranti
- Per ogni coppia di stati (non necessariamente distinti), c'è esattamente un arco.
- Ogni arco è etichettato da un espressione regolare.

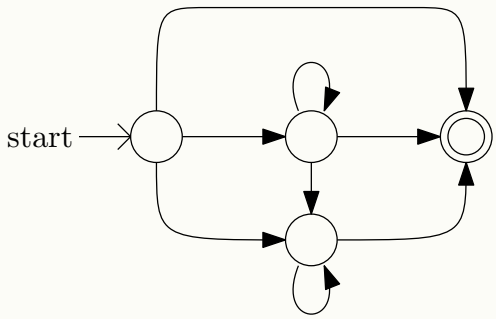


Figura 1.13: forma di un GNFA

Più precisamente, sia G un GNFA definito $G = (Q, \Sigma, \delta, q_{start}, q_{acc})$ dove

$$\delta : Q\backslash\{q_{acc}\} \times Q\backslash\{q_{start}\} \to re(\Sigma)$$

Dato un generico NFA, è possibile trasformarlo in un GNFA aggiungendo al più due stati (iniziale e finale), ed utilizzando gli ϵ-archi per riempire le coppie di stati che non sono collegate.

La funzione $Convert : GNFA \to GNFA$ modifica un GNFA restituendone uno equivalente, ma con uno stato in meno. Tale funzione è definita in tal modo, sia k il numero di archi di G, si esegue $Convert(G)$

- Se $k = 2$, allora esiste un solo arco fra questi etichettato con un espressione regolare r, la funzione restituirà r.
- Se $k > 2$, viene selezionato un qualsiasi nodo in $Q\backslash\{q_{start}, q_{acc}\}$, sia questo q_{rip}, si avrà $Convert(G) = G' = (Q\backslash\{Q_{rip}\}, \Sigma, \delta', q_{start}, q_{acc})$ dove

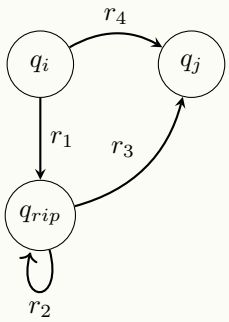
  $$\delta' : Q\backslash\{q_{acc}, q_{rip}\} \times Q\backslash\{q_{start}, q_{rip}\} \to re(\Sigma)$$

  Inoltre ogni etichetta di G' viene aggiornata secondo la seguente procedura, siano $q_i \in Q\backslash\{q_{acc}, q_{rip}\}$ e $q_j \in Q\backslash\{q_{start}, q_{rip}\}$ due stati qualsiasi

  $$\delta'(q_i, q_j) = r_1 r_2^* (r_3 | r_4)$$

  Dove

  - $r_1 = \delta(q_i, q_{rip})$
  - $r_2 = \delta(q_{rip}, q_{rip})$
  - $r_3 = \delta(q_{rip}, q_j)$
  - $r_4 = \delta(q_i, q_j)$



Bisogna ora dimostrare che un generico GNFA G è equivalente a $Convert(G)$. Si dimostra per induzione su k numero di stati.

caso base $k = 2$: In tal caso la procedura $Convert$ restituisce l'espressione regolare r sull'unico arco che descrive ogni stringa accettata da G. $L(r) \equiv L(G)$

passo induttivo : si assume che G è equivalente a $Convert(G)$ per $k - 1$ stati.

- Se G accetta w, allora esiste un ramo di computazione $C = \{q_{start}, q_1 \ldots, q_{accept}\}$, se q_{rip} che è stato rimosso in $G' = Convert(G)$ non appartiene a C, allora la computazione non è alterata e G' accetta w, altrimenti ci sarà una differente sequenza di stati, ma gli stati q_i, q_j adiacenti a q_{rip} sono ora uniti da un arco etichettato da un espressione regolare che comprende le stringhe per andare da q_i a q_j passando per q_{rip}.
- Se G' accetta w, anche G lo accetta dato che per ogni coppia di stati in C si è aggiornata l'etichetta tenendo conto della transazione che porta da uno stato all'altro passando per q_{rip}.

Quindi $Convert$ restituisce un automa equivalente con $k - 1$ stati, quindi l'asserto è vero. ■

1.4.1 Esempi

Esempio 1) Si trasformi $r = (ab|a)^*$ in un NFA.

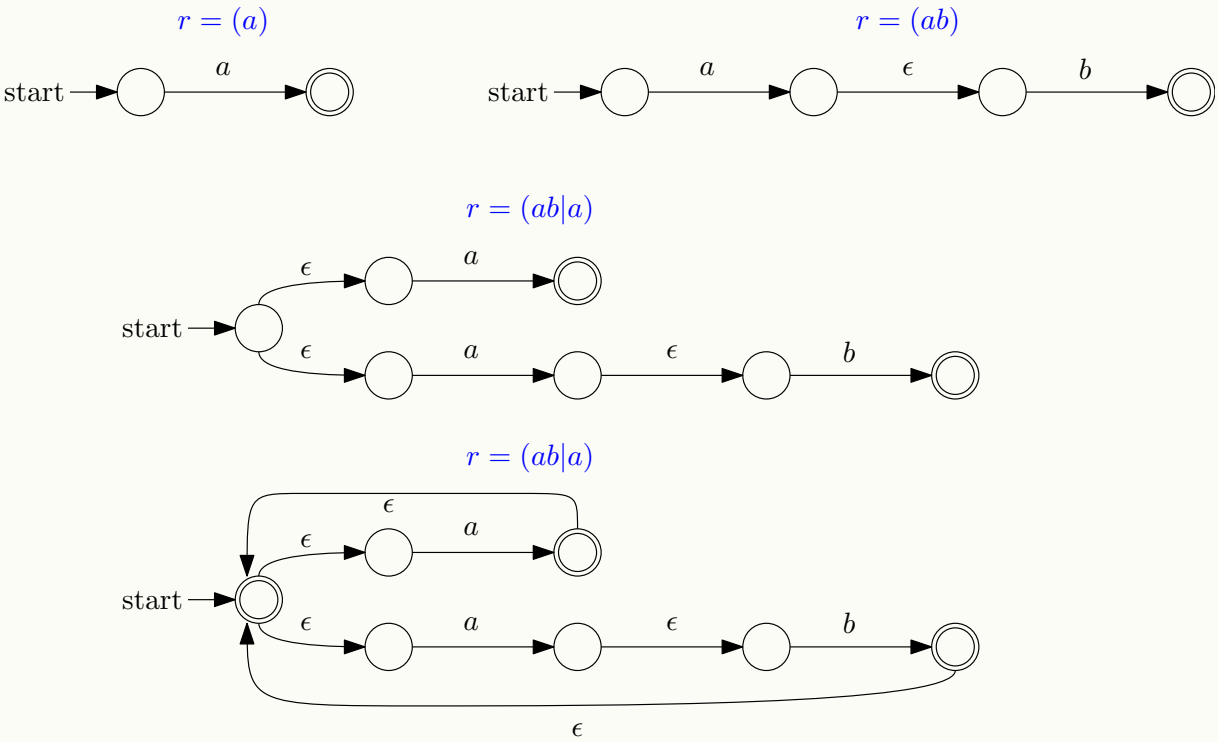
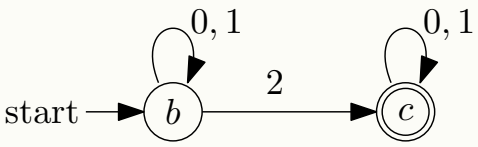
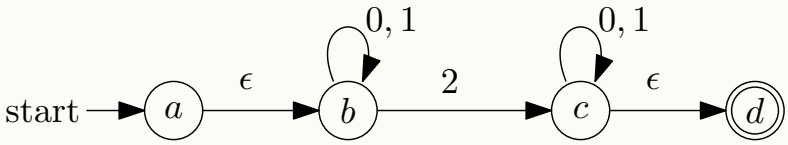


Figura 1.14: Esempio 1

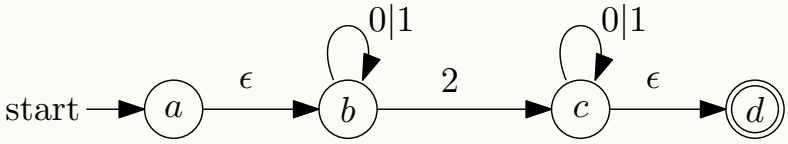
Esempio 2) Dato il seguente automa, si trovi l'espressione regolare associata



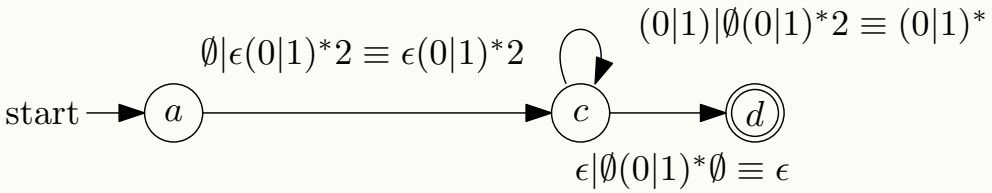
passo 1 : si trasforma in forma canonica (ignorati gli archi etichettati con $\emptyset$)



passo 2 : si trasformano le etichette in espressioni regolari



passo 3 : si rimuove b



passo 4 : si rimuove b

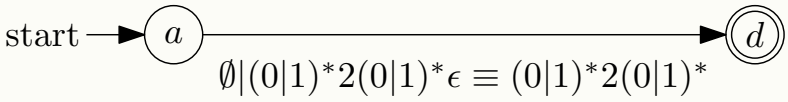


Figura 1.15: Esempio 2

1.5 Linguaggi non regolari

1.5.1 Il Pumping Lemma per i Linguaggi Regolari

A questo punto della lettura, è naturale porsi una domanda : Tutti i linguaggi sono regolari? Esistono linguaggi che non possono essere accettati da alcun DFA? Nel caso solamente un sottoinsieme dei linguaggi fosse regolare, quali proprietà soddisfa? Si consideri il seguente linguaggio

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

Si può provare a disegnare un automa che accetti L, rendendosi ben presto conto che è *impossibile*, L non è regolare, esistono quindi dei linguaggi che non sono regolari. L'automa a stati finiti è un modello semplice, non può "ricordare" quanti caratteri di un certo tipo sono stati letti.

Essendo che solamente alcune stringhe possono essere accettate da un qualsiasi automa, è importante caratterizzare tali stringhe e definirne le proprietà in tal merito.

Appunto sulla notazione : Se w è una stringa, allora $|w|$ è il numero dei suoi caratteri.

Osservzione : Se un DFA con n stati legge una stringa di $k > n$ caratteri, allora ci sarà almeno uno stato che verrà considerato due volte durante la computazione.

Teorema (Pumping Lemma) : Sia L un linguaggio regolare, sia D l'automa tale che $L(D) = L$, si considera una stringa $w \in L(D)$, ed una sua decomposizione in 3 stringhe concatenate $w = xyz$. Esiste un intero $p \leq |w|$, denotato *pumping* tale che

1. $\forall i \geq 0$, $xy^i z \in L(D)$
2. $|y| > 0$
3. $|xy| \leq p$

Dimostrazione : Sia $D = (Q, \Sigma, \delta, q_{start}, F)$ un automa, e sia $p = |Q|$. Sia w una stringa su Σ di $n \geq p$ caratteri definita $w = w_1 w_2 \dots w_n$. Sia $\{r_1, r_2 \dots, r_{n+1}\}$ la sequenza di stati che D computa su input w, ossia

$$\delta(r_i, w_i) = r_{i+1}$$

Tale sequenza è lunga $n+1 \geq p+1$ stati, fra i primi $p+1$ elementi c'è necessariamente uno stato ripetuto, sia r_j la prima occorrenza di tale stato, e sia r_l la seconda.

Siccome la ripetizione avviene fra le prime $p+1$ computazioni, si ha che $l \leq p+1$. Si consideri la seguente scomposizione di w

- $x = w_1, w_2 \dots, w_{j-1}$
- $y = w_j, w_{j+1} \dots, w_{l-1}$
- $z = w_l, w_{l+1} \dots, w_n$

1. $xy^i z \in L(D)$ perché x parte da $r_1 = q_{start}$ e arriva a r_j, y^i parte da r_j e ritorna su r_l, che è lo stesso stato, e z porta da r_l allo stato finale di accettazione.
2. Essendo che $i \neq l$, $|y| > 0$.
3. $l \leq p + 1$ ovvero $l - 1 = |xy| \leq p$.

I tre punti sono dimostrati. ■

Proposizione : Se L è un linguaggio regolare, ed L^* un sottoinsieme di L, allora L^* non è necessariamente regolare.

Alcuni esercizi al seguente link : Esercizi Linguaggi Regolari.

1.6 Grammatiche Acontestuali

Lo scopo delle grammatiche acontestuali è quello di estendere l'automa a stati finiti per ottenere un modello di computazione più potente. Tale automa al quale corrispondono le gramamtiche è detto *PDA*. Le grammatiche hanno applicazioni fondamentali, nei linguaggi di programmazione, precisamente, nel funzionamento dei compilatori.

Una definizione informale può essere la seguente : Una grammatica è composta da un insieme di *regole* su un alfabeto e delle variabili, tali regole sono annotate come segue

$$\begin{cases} A \longrightarrow 0A1 \\ A \longrightarrow B \\ B \longrightarrow \# \end{cases} \qquad \Sigma = \{0, 1, \#\}$$

Ciascuna regola contiene una variabile alla quale viene associata una stringa, composta da variabili e *terminali*, ossia i caratteri dell'alfabeto Σ. Una variabile è considerata iniziale, e per convenzione, è sempre quella presente nella prima regola.

Precisamente, una grammatica può generare stringhe

1. Si scrive la variabile iniziale
2. Si sostituisce applicando una delle regole
3. Si ripete ricorsivamente il procedimento finché la stringa contiene solo terminali.

Esempio : Considerando la grammatica con le regole prima elencate :

1. A si applica $A \longrightarrow 0A1$
2. $0A1$ si applica $A \longrightarrow 0A1$
3. $00A11$ si applica $A \longrightarrow 0A1$
4. $000A111$ si applica $A \longrightarrow B$
5. $000B111$ si applica $B \longrightarrow \#$
6. $000\#111$

Applicando le regole secondo un ordine arbitrario, è possibile generare qualsiasi stringa del tipo $0^n\#1^n$, ad una grammatica quindi corrisponde un linguaggio. Una 'computazione' di una grammatica può essere rappresentata con un albero sintattico. 1.16

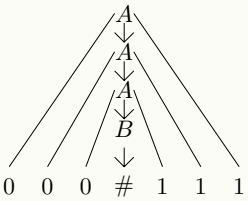


Figura 1.16: Albero Sintattico

Con una grammatica è anche possibile rappresentare una qualsiasi espressione algebrica

$$\begin{cases} E \longrightarrow E + E \\ E \longrightarrow E \times E \\ E \longrightarrow (E) \\ E \longrightarrow 0 \vee 1 \vee 2 \ldots, \vee 9 \end{cases} \qquad \Sigma = \{0, 1, 2, 3 \ldots, 9\}$$

Definizione (Grammatica Acontestuale) : Una **CFG** (Context Free Grammar) è una tupla $G = (V, \Sigma, R, S)$ dove

- V è un insieme di simboli dette variabili
- Σ è un insieme di simboli detti terminali
- $V \cap \Sigma = \emptyset$
- $S \in V$ è la variabile iniziale
- R è un insieme di regole

Le regole R possono essere rappresentate come una funzione $R : V \to (V \cup \Sigma)^*$, associa ad ogni variabile una stringa composta da terminali, variabili, o entrambe.

Sia uAv u8na stringa tale che $A \in V$, $u, v \in \Sigma \cup V$, e sia $w \in \Sigma \cup V$ diremo che uAv **produce** uwv, e denoteremo $uAv \Rightarrow uwv$ se e solo se

$$A \longrightarrow w \in R$$

Il simbolo $\Rightarrow$ rappresenta una relazione su $\Sigma \cup V$. Si può considerare la sua chiusura transitiva $\Rightarrow^*$

$$u \Rightarrow^* v \iff \exists\{u_1, u_2 \ldots, u_k\} \mid u \Rightarrow u_1 \Rightarrow u_2 \cdots \Rightarrow u_k \Rightarrow v$$

Sia $G = (V, \Sigma, R, S)$ una CFG, $L(G)$ è il **linguaggio della grammatica**, definito come segue

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Unione

Si consideri il seguente insieme di grammatiche

$$\{G_i = (V_i, \Sigma_i, R_i, S_i)\} \qquad i \in \{1, 2, 3 \ldots, n\}$$

È possibile considerare *l'unione delle grammatiche*, definita come segue

$$G = (\bigcup_i \{V_i\} \cup \{S\}, \bigcup_i \Sigma_i, R, S)$$

Dove

- S è una nuova variabile
- $R = \{\bigcup_i R_i\} \cup \{S \to S_1 \vee S \to S_2 \cdots \vee S \to S_n\}$

Proposizione : $L(G) = \bigcup_i L(G_i)$

Ambiguità

Una CFG soddisfa la proprietà di ambiguità se, una stessa stringa può essere generata seguendo sequenze di regole differenti. L'ambiguità di una grammatica può risultare problematica nelle applicazioni.

Definizione : Una stringa di una CFG ha una *derivazione a sinistra* se può essere ottenuta sostituendo ad ogni passo di produzione (applicazione delle regole) la variabile che si trova più a sinistra. Una stringa è *derivata ambiguamente* se ha 2 o più derivazioni a sinistra. Una CFG è ambigua se ha almeno una stringa derivata ambiguamente.

1.6.1 Forma Normale

In questa sezione verrà definita una forma canonica per le grammatiche, tale forma è fondamentale, soprattutto nelle applicazioni, se una grammatica è in tale forma, è possibile stabilire un tetto minimo di operazioni da eseguire per ottenere una determinata stringa.

Definizione : Una CFG $G = (V, \Sigma, R, S)$ è in **forma normale Chomsky** (o più comunemente, in forma normale) se ogni sua regola è della forma

$$A \longrightarrow BC$$
$$A \longrightarrow a$$

Dove

- $a \in \Sigma$
- $A, B, C \in V$
- $B \neq S, \quad C \neq S$
- La regola $S \longrightarrow \epsilon$ è permessa.

La variabile iniziale S non può mai essere nel termine destro di una regola.

Teorema : Per ogni CFG, esiste una CFG equivalente (che genera lo stesso linguaggio) in forma normale.

Dimostrazione : La dimostrazione consiste in una procedura, in cui vengono applicate delle trasformazioni alle regole di una generica CFG in modo che soddisfino la forma normale. Sia $G = (V, \Sigma, R, S)$ una CFG, non necessariamente in forma normale

- Si definisce una nuova variabile S_0, essa sarà considerata la nuova variabile iniziale della grammatica G, e verrà aggiunta la regola $S_0 \to S$, ciò garantisce che la variabile iniziale non compare mai come termine destro.
- Data una ϵ-regola, ossia le regole della forma

  $$A \longrightarrow \epsilon \qquad A \in V$$

  Si considera ogni occorrenza di A nel termine destro di una regola, e si definisce una nuova regola identica, dove A è assente. Infine, $A \longrightarrow \epsilon$ viene rimossa dalle regole

  $$\begin{cases} A \longrightarrow \epsilon \\ B \longrightarrow xAyA \end{cases} \implies \text{(diventa)} \begin{cases} B \longrightarrow xAyA \\ B \longrightarrow xyA \text{ (aggiunta)} \\ B \longrightarrow xAy \text{ (aggiunta)} \end{cases} \qquad \begin{matrix} B \in V \\ x, y \in (\Sigma \cup V)^* \end{matrix}$$

- Si considerano poi tutte le regole unitarie, ossia del tipo

  $$A \longrightarrow B$$

  con $B \in V$, una volta rimossa, per ogni altra regola $B \to u$, si aggiunge la regola $A \to u$, con $u \in (\Sigma \cup V)^*$, ripetendo ricorsivamente il procedimento.
- Infine si considerano le regole del tipo

  $$A \longrightarrow u_1 u_2 \ldots, u_k \qquad \begin{matrix} u_i \in (\Sigma \cup V) \\ k \geq 3 \end{matrix}$$

  Tale regola viene rimossa, e vengono spezzate in un set di regole come segue

  $$\begin{cases} A \longrightarrow u_1 A_1 & \text{nuova variabile } A_1 \text{ aggiunta a } V \\ A_1 \longrightarrow u_2 A_2 & \text{nuova variabile } A_2 \text{ aggiunta a } V \\ A_2 \longrightarrow u_3 A_3 & \text{nuova variabile } A_3 \text{ aggiunta a } V \\ \vdots & \\ A_{k-2} \longrightarrow u_{k-1} u_k & \text{nuova variabile } A_{k-2} \text{ aggiunta a } V \end{cases}$$

  Una volta fatto ciò, per ogni regola $A_i \longrightarrow u_j A_j$ in cui u_j è un terminale, si definisce una nuova variabile U_j ed una nuova regola $U_j \longrightarrow u_j$. Ad esempio, se u_1 è un terminale, viene rimossa $A \longrightarrow u_1 A_1$, e vengono definite le nuove regole

  $$\begin{cases} A \longrightarrow U_1 A_1 \\ U_1 \longrightarrow u_1 \end{cases}$$

Una volta eseguite le procedure, la nuova CFG sarà equivalente a quella originale, e sarà in forma normale Chomsky. ■

Esempio

Si consideri la seguente grammatica G

$$\begin{cases} S \longrightarrow ASA \vee aB \\ A \longrightarrow A \vee S \\ B \longrightarrow b \vee \epsilon \end{cases} \quad \text{si può scrivere anche} \quad \begin{cases} S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases}$$

- Passo 1 : variabile iniziale

$$\begin{cases} S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases}$$

- Passo 2 : ϵ-regole

$$\begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S|\epsilon \\ B \longrightarrow b \end{cases}$$

$$\begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S|\epsilon \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB|SA|AS|S \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases}$$

- passo 3 : regole unitarie
  si rimuove $S \longrightarrow S$

$$\begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB|SA|AS|S \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases}$$

  si rimuovono $A \longrightarrow B$ e $A \longrightarrow S$

$$\begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow b \\ A \longrightarrow ASA|aB|SA|AS \\ B \longrightarrow b \end{cases}$$

- passo 4 : convertire le ultime regole nella forma corretta

$$\begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow b \\ A \longrightarrow ASA|aB|SA|AS \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} U \longrightarrow a \\ S_0 \longrightarrow AA_1|UB|SA|AS \\ A_1 \longrightarrow SA \\ S \longrightarrow AA_1|UB|SA|AS \\ A \longrightarrow b \\ A \longrightarrow AA_1|UB|SA|AS \\ B \longrightarrow b \end{cases} \quad \text{variabili aggiunte : } A_1,\ U$$

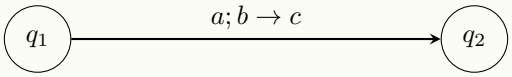
1.7 Push Down Automata

Si considera adesso un nuovo modello di computazione che estende il concetto di automa, vedremo in seguito che, i linguaggi regolari stanno agli NFA, come le grammatiche acontestuali stanno ai PDA. Introdurremo alcune caratteristiche dei PDA in modo informale, per poi darne la definizione.

Innanzitutto un PDA è sempre un automa, non deterministico, in particolare si differenzia dagli NFA/DFA per la presenza di una **pila** (di dimensione potenzialmente infinita) detta anche *stack*, il cui contenuto evolve dinamicamente durante la computazione, tale pila permette all'automa di *memorizzare*. L'accesso ad essa è limitato, esclusivamente LIFO, è possibile scrivere sulla pila una serie di caratteri appartenenti ad un determinato insieme, su di essa è possibile eseguire operazioni di

- pop
- top
- push

Se un NFA ha un solo alfabeto Σ, un PDA avrà due alfabeti, Σ e Γ, denotati *input* e *pila*. Anche la funzione δ sarà differente, infatti gli archi saranno marcati nel seguente modo



Dove b e c sono elementi di $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$, ed a è un elemento di Σ_ϵ. In particolare, la freccia in figura indica che :

> Se l'automa è nello stato q_1, legge in input a, ed in cima alla pila si trova b, allora si sposterà nello stato q_2, verrà rimosso b dalla pila e verrà aggiunto c

Il dominio della funzione di transizione sarà quindi $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$, mentre il codominio sarà l'insieme delle parti di $Q \times \Gamma_\epsilon$. Essendo non deterministico, ad ogni ramo di computazione sarà associata una differente evoluzione della pila.

Definizione (PDA) : Un *Push Down Automata* è una tupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ dove

- Q, Σ, q_0, F sono definiti identicamente agli NFA/DFA
- Γ è un alfabeto finito rappresentante gli elementi che possono essere nello stack
- $\delta : Q \times \Gamma_\epsilon \times \Sigma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$

Considerato un PDA, supponiamo che

$$(q, c) \in \delta(p, a, b) \quad \text{con} \quad \begin{cases} q, p \in Q \\ a \in \Sigma_\epsilon \\ c, b \in \Sigma_\epsilon \end{cases}$$

Allora, a seconda dei valori di a, b e c, lo step di computazione assume il seguente significato

- $a, b, c \neq \epsilon \implies$ il PDA legge a, passa dallo stato p allo stato q e sostituisce b (che si trova nel top dello stack) con c.
- $a, c \neq \epsilon \wedge b = \epsilon \implies$ il PDA legge a, passa dallo stato p allo stato q indipendentemente dal valore del top dello stack, su cui andrà ad inserire c
- $a, b \neq \epsilon \wedge c = \epsilon \implies$ il PDA legge a, passa dallo stato p allo stato q ed eseguo una pop sullo stack rimuovendo b.

Passiamo alla descrizione di come viene *eseguita la computazione* su un PDA, si consideri una stringa w in input, composta dai caratteri

$$w = w_0 w_1 \dots w_n \text{ con } w_i \in \Sigma$$

consideriamo ora gli stati che verranno attraversati nella computazione :

$$r_0, r_1, r_2 \ldots, r_{n+1}$$

e si considerino le stringhe

$$s_1, s_2 \ldots, s_n \text{ con } s_i \in \Gamma^*$$

Lo stato $r_0 = q_0$ sarà lo stato iniziale, ed $s_0 = \epsilon$ rappresenterà la pila vuota, inoltre, $\forall i = 0, 1 \ldots, n$ si avrà che

$$(r_{i+1}, a) \in \delta(r_i, w_{i+1}, b)$$

Dove s_i rappresenterà lo stack nel momento in cui si è nello stato r_i, ed s_{i+1} rappresenterà lo stack nel momento in cui si è nello stato r_{i+1}, si avrà infatti che

- $s_i = bT$
- $s_{i+1} = aT$
- $T \in \Gamma_\epsilon^*$
- In breve, nello stack è stata sostituito il simbolo b con a

La computazione sarà andata a buon fine se e solo se $r_n \in F$, ossia lo stato finale è di accettazione.

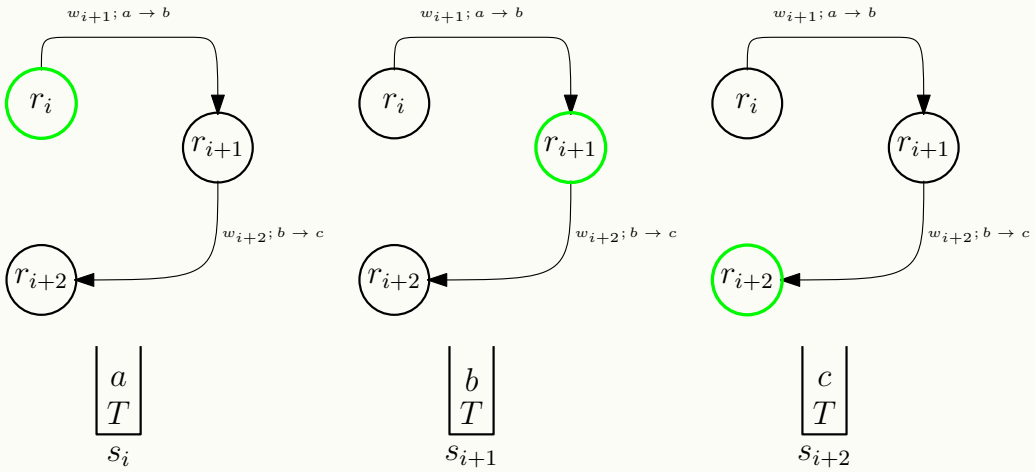


Figura 1.17: Lo stato verde rappresenta la computazione corrente

Gli stati consistenti nella computazione sono in relazione

$$(p, ax, by) \vdash (q, x, cy) \iff (q, c) \in \delta(p, a, b)$$

$$p, q \in Q \quad b, c \in \Gamma_\epsilon \quad a \in \Sigma_\epsilon \quad x \in \Sigma^* \quad y \in \Gamma^*$$

La chiusura transitiva $\vdash^*$ definisce la relazione di transizione estesa, utile per la seguente definizione.

Definizione : Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ un PDA, definiamo **linguaggio di** P, e denotiamo $L(P)$, l'insieme

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, \epsilon) \vdash^* (q, \epsilon, y) \wedge q \in F \wedge y \in \Gamma^*\}$$

Si può definire in manire equivalente assumendo che la computazione termini sempre con lo stack vuoto

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, \epsilon) \vdash^* (q, \epsilon, \epsilon) \wedge q \in F \wedge\}$$

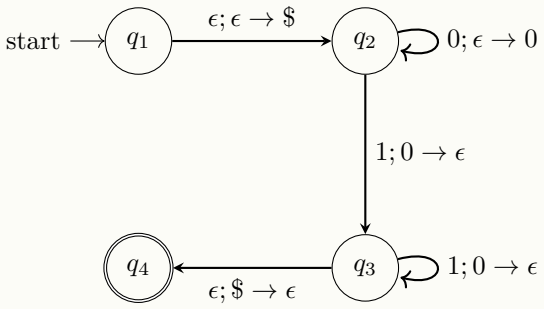
Il fatto che le due definizioni siano equivalenti, implica che ogni PDA può essere trasformato in un PDA equivalente in cui ogni stringa accettata termini la computazione con lo stack vuoto.

1.7.1 Esempi

Esempio 1

Sia $L = \{0^n 1^n | n \geq 0\}$ un linguaggio, si vuole determinare un PDA P tale che $L(P) = L$.

L'idea è quella di utilizzare lo stack, aggiungendo un valore ogni volta che si legge uno zero, e togliendone uno ogni volta che si legge un 1, se al termine la pila sarà vuota, allora la stringa letta apparterrà al linguaggio L. Definiamo $\Gamma = \{0, \$\}$, il simbolo \$ è utile per marcare il fondo dello stack, verrà aggiunto automaticamente all'inizio.

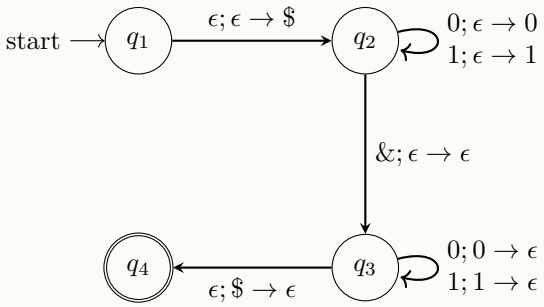


Esempio 2

Data una stringa w, definiamo w^R la stringa w specchiata, ad esempio

$$w = a12hgf \implies w^R = fgh21a$$

Sia $L = \{w\&w^R | w \in \{0,1\}^*\}$, ossia l'insieme di tutte le stringhe binarie palindrome, al cui centro è presente il carattere &.



1.7.2 PDA e Linguaggi Acontestuali

Tale sezione si concentrerà sul seguente teorema.

Teorema : Un linguaggio è acontestuale (generato da una CFG) se e solo se esiste un PDA che lo riconosce. Per rendere più leggibile e chiara la dimostrazione del teorema, essa verrà scomposta in due lemma, che rappresentano le due implicazioni della dimostrazione.

Lemma [$\implies$] : Se un linguaggio è acontestuale, esiste un PDA che lo riconosce.

Dimostrazione [$\implies$] : Sia $G = (V, \Sigma, R, S)$ una generica CFG, il cui linguaggio è $L(G)$. L'idea è quella di costruire un PDA P che sfrutti il non determinismo per accettare una generica stringa $w \in L(G)$ usando tutte le possibili derivazioni (applicazioni delle regole) di G.

Lo stack di P conterrà le variabili ed i terminali di G che verranno appositamente sostituiti con le regole. Possono esserci differenti alberi di computazione, il comportamento del PDA può essere scritto dal seguente algoritmo

◇ si inserisce $ nella pila

◇ while(true){

◇ Se nel top dello stack vi è una variabile, si eseguono varie diramazioni in base a tutte le regole che comprendono tale variabile

◇ Se nel top dello stack c'è un terminale, viene rimosso dallo stack e si confronta con il prossimo carattere in input, se identici la computazione continua, altrimenti rifiuta.

◇ Se nel top dello stack c'è $ (la pila è svuotata), si accetta se e solo se è stata letta tutta la stringa in input.

◇ }

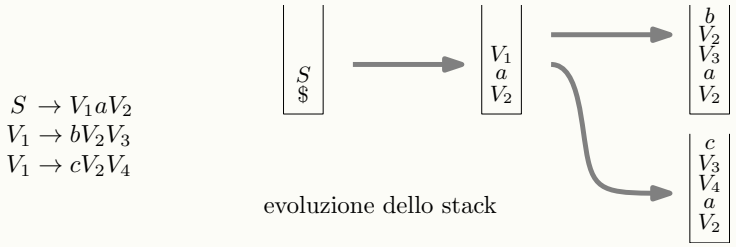
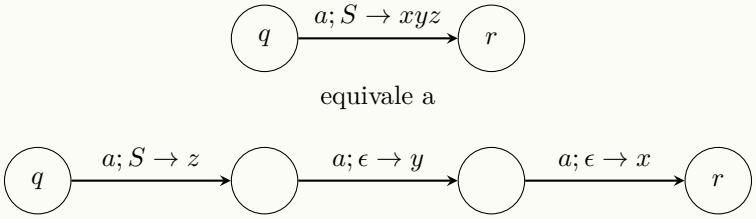


Figura 1.18: Esempio di computazione

Notazione : Per comodotià, verrà utilizzata una notazione ridotta nel diagramma del PDA in questione, in particolare, si introduce l'inserimento di una stringa nello stack, piuttosto che di un solo carattere, ovviamente un push down di questo tipo equivale a più push down, che richiedono stati intermedi che verranno omessi.



Sia $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$ l'automa che deve accettare il linguaggio $L(G)$, l'insieme degli stati sarà

$$Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$$

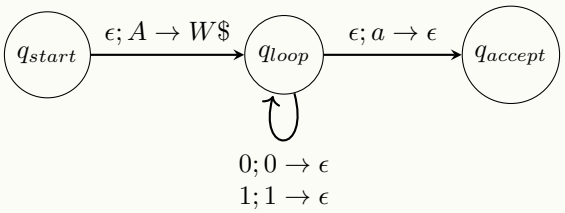
Dove E è l'insieme degli stati intermedi, ossia quelli che verranno omessi nella notazione. Siano

- $a \in \Sigma$ un carattere generico della grammatica e dell'alfabeto del PDA corrispondente
- $A \in V$ una generica variabile della grammatica

Si ricordi che gli elementi del codominio di δ sono insiemi, dato che il PDA è non deterministico. La funzione di transizione del PDA P sarà definita come segue

- $\delta(q_{start}, \epsilon, \epsilon) = \{(q_{loop}, S\$)\}$ si ricordi che S è la variabile iniziale della grammatica e $ è il simbolo per marcare il fondo dello stack
- $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, W) \mid A \to W \in R\}$, ossia $A \longrightarrow W \in R$ è una generica regola della grammatica
- $\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$
- $\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$

In generale (ignorando gli stati in E, in notazione abbreviata), il PDA assumerà una forma canonica del tipo



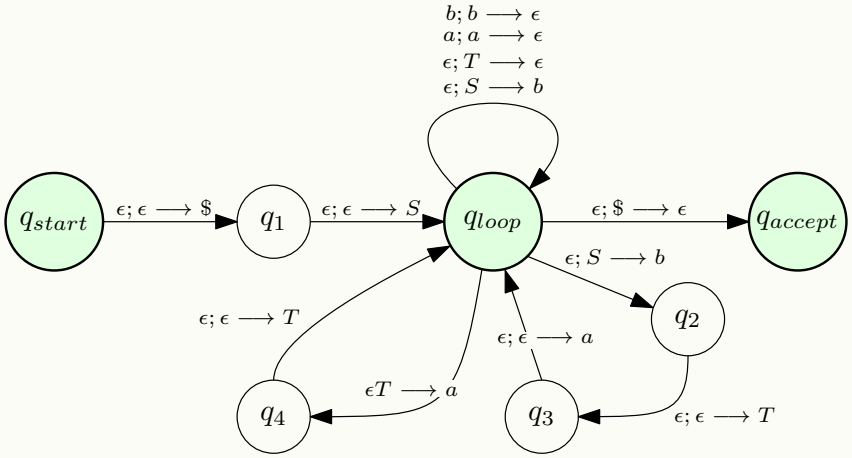
■

Esempio

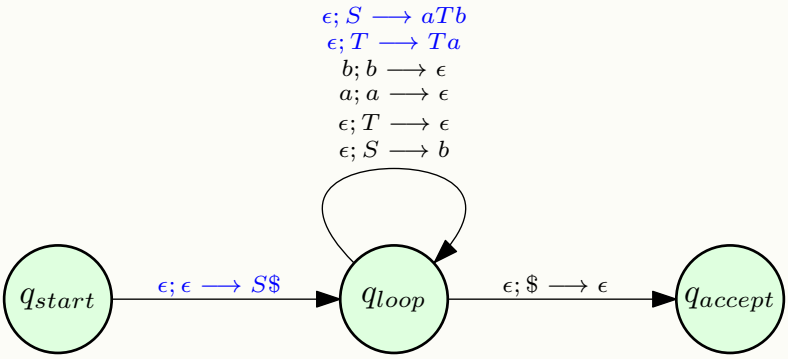
Si consideri la seguente grammatica

$$\begin{cases} S \longrightarrow aTb|b \\ T \longrightarrow Ta|\epsilon \end{cases} \qquad \begin{matrix} \Sigma = \{a, b\} \\ V = \{S, T\} \end{matrix}$$

Il PDA corrispondente avrà pila $\Gamma = \{\$, S, a, T, b\}$, precisamente



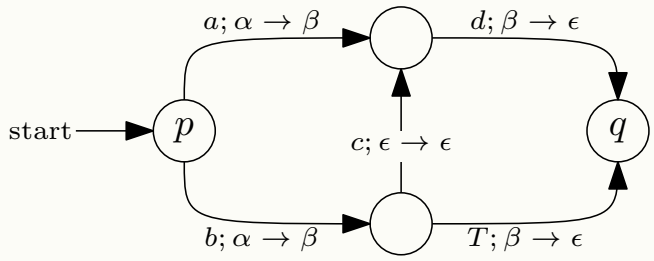
Gli stati $E = \{q_1, 1_2, q_3, q_4\}$ sono quelli di transizione che nella notazione abbreviata verrebbero omessi, sostituendo le etichette con gli archi.



notazione abbreviata

Lemma [$\Longleftarrow$] : Dato un PDA, esiste una CGF le cui stringhe generate sono il linguaggio del PDA.

Dimostrazione [$\Longleftarrow$] : Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ il PDA in questione. L'idea è quella di considerare, per ogni coppia di stati p, q in Q, una variabile denotata A_{pq} che genera tutte le stringhe che permettono al PDA di passare da p a q lasciando la pila vuota, come nell'esempio in figura 1.19.



si avrà

$A_{pq} \longrightarrow ad$
$A_{pq} \longrightarrow bcd$
$A_{pq} \longrightarrow bT$

Figura 1.19: esempio

Forma Canonica : Durante la dimostrazione, si assumerà che P soddisfi certe proprietà, ossia che assuma una *forma canonica*, tale forma non fa perdere di generalità alla dimostrazione in quanto ogni PDA può essere trasformato in un PDA in tale forma lasciando invariato il linguaggio che accetta. Un PDA è nella forma canonica se

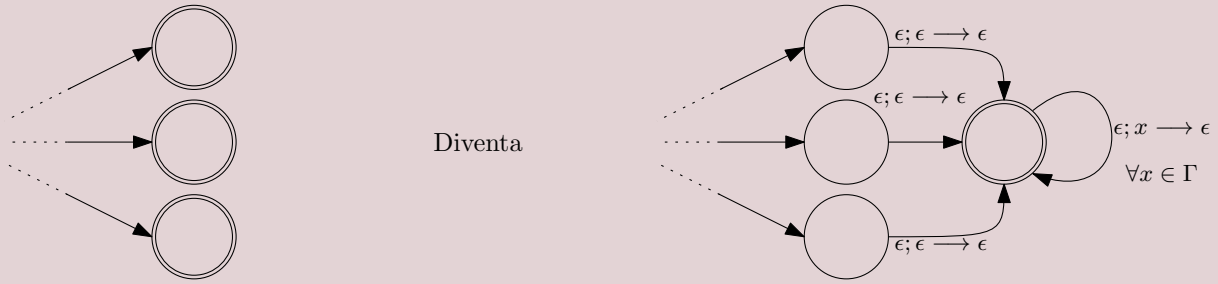
1. Inizia con la pila vuota, ed termina la computazione sempre con la pila vuota
2. Ha un solo stato accettante $|F| = 1$
3. Il PDA ad ogni passo computazionale, non sostituisce mai un elemento della pila con un altro, o ne elimina uno, o ne aggiunge uno, la sostituzione non avviene mai in un solo passo di computazione, formalmente, cambia la definizione della funzione di transizione

$$\delta_1 : Q \times \Gamma_\epsilon \times \Sigma_\epsilon \to \mathcal{P}(Q \times \{\epsilon\})$$

$$\delta_2 : Q \times \{\epsilon\} \times \Sigma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$$

$$\delta = \delta_1 \cup \delta_2$$

È chiaro che ogni PDA può essere portato in tale forma, se ci sono più stati di accettazione, si creerà un nuovo stato alla quale tutti rimandano, rendendolo l'unico stato di accettazione. Quest'ultimo stato inoltre si occuperà di svuotare la pila automaticamente grazie agli ϵ-archi. Inoltre, ogni arco che esegue una sostituzione nello stack sarà diviso in due archi che eseguono in sequenza un pop ed un push, facendo uso di uno stato intermedio.



Bisogna definire le regole dela grammatica, la variabile iniziale S sarà $A_{q_0 q_{accept}}$, quindi per definizione S genererà tutte le stringhe che accetta il PDA. L'insieme delle regole R della grammatica G sarà definito come segue

- Siano $p, q, r, s \in Q$, $u \in \Gamma$, $a, b \in \Sigma_\epsilon$, se

$$(r, u) \in \delta(p.a.\epsilon) \wedge (q, \epsilon) \in \delta(s, b, u)$$

allora ci sarà la regola

$$A_{pq} \longrightarrow aA_{rs}b$$

Figura 1.20: situazione del PDA

È ovvio che che si possa arrivare da p a q in tal modo, il carattere a porta da p ad r (aggiungendo u), il carattere b porta da s a q (rimuovendo u), la variabile A_{rs} deriva[1] i caratteri che portano da r ad s con pila vuota.

- Per ogni tripla $p, q, r \in Q$ si pone la regola $A_{pq} \longrightarrow A_{pr}A_{rq}$.
- Per ogni $p \in Q$, si pone la regola $A_{pp} \longrightarrow \epsilon$

La prova della dimostrazione segue dal fatto che A_{pq} genera x *se e solo se* x porta l'automa P da p a q con pila vuota, ne segue che, se $S = A_{q_0 q_{accept}}$ deriva x, allora $x \in L(P)$. Tale dimostrazione sarà suddivisa in due claim.

Claim 1 : Se A_{pq} deriva x, allora x porta da p a q con pila vuota.

Dimostrazione claim 1 : Verrà dimostrato per induzione sul numero di produzioni di x in G.

- **caso base** : x è generata da una regola di G per $k = 1$ produzione, allora l'unica regola che può generare x è del tipo $A_{pp} \longrightarrow \epsilon$, allora $x = \epsilon$, e la stringa vuota porta l'automa P da p a p con pila vuota.
- **ipotesi induttiva** : Per ogni x generata da una regola di G per $k > 1$ produzioni, tale stringa congiunga i due stati in questione lasciando la pila vuota.
- **passo induttivo** : Supponiamo che A_{pq} produca x in $k+1$ produzioni, allora, data la definizione delle regole di G, la regola in questione può essere di due tipi :
  - La regola in questione è $A_{pq} \longrightarrow aA_{rs}b$, allora x ha la forma ayb, ne consegue che A_{rs} deriva y in al più k passi, quindi, per ipotesi, la stringa y porta l'automa P dallo stato r allo stato s lasciando la pila vuota. Inoltre, per definizione delle regole è vero che

    $$(r, u) \in \delta(p, a, \epsilon) \wedge (q, \epsilon) \in \delta(s, b, u) \quad \text{per qualche } u \in \Gamma$$

    Quindi è chiaro che x porti P da p a q lasciando la pila vuota.
  - La regola in questione è $A_{pq} \longrightarrow A_{pr}A_{rq}$„ allora la stringa x sarà del tipo yz, quindi A_{pr} deriva y, mentre A_{rq} deriva z. Se A_{pq} deriva x in $k+1$ produzioni, allora le derivazioni di y e z avvengono in al più k produzioni, ma allora per ipotesi

    $$\begin{cases} y \text{ porta } P \text{ da } p \text{ a } r \text{ lasciando la pila vuota} \\ z \text{ porta } P \text{ da } r \text{ a } q \text{ lasciando la pila vuota} \end{cases} \implies x = yz \text{ porta } P \text{ da } p \text{ a } q \text{ lasciando la pila vuota}$$

□

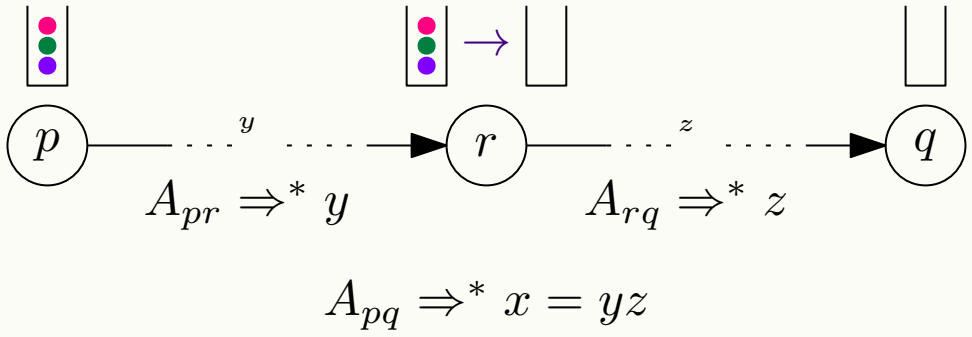
Claim 2 : Se x porta da p a q con pila vuota, allora A_{pq} deriva x.

Dimostrazione claim 2 : Verrà dimostrato per induzione sul numero di passi di computazione dell'automa P.

- **caso base** : L'automa compie 0 passi, la computazione inizia e finisce in p, l'input x è quindi la stringa vuota, siccome per definizione delle regole, G ha la regola $A_{pp} \longrightarrow \epsilon$, allora A_{pp} deriva x.

[1] con *deriva*, si intende l'atto di generare tale stringa attraverso diverse produzioni, ossia la chiusura transitiva della relazione di produzione, precedentemente denotata $\Rightarrow^*$

- **ipotesi induttiva** : Si ipotizza che per ogni computazione di $k > 0$ passi (che lasci vuota la pila) da p a q con stringa in input x, la regola A_{pq} deriva x.
- **passo induttivo** : Supponiamo che x porti da p a q lasciando la pila vuota in $k+1$ passi di computazione, precisamente, ci sono due possibili casi
  - La pila è vuota solo negli stati p e q, negli stati intermedi è stata riempita dal primo passo, e verrà poi svuotata all'ultimo passo. Sia u, tale carattere inserito all'inizio e rimosso al termine, e sia a il simbolo che porta p allo stato successivo r (primo passo di computazione), e b il simbolo che porta dal penultimo stato s a q (ultimo passo di computazione), proprio come mostrato in figura 1.20. Per definizione, la grammatica G contiene la regola $A_{pq} \longrightarrow aA_{rs}b$, quindi x è del tipo ayb. La variabile A_{rs} deriva y, quindi y porta da r a s lasciando la pila vuota con al più $k-1$ passi. Ricapitolando
    - a porta da p ad r aggiungendo u
    - y porta da r ad s senza fare operazioni sulla pila
    - b porta da s a q rimuovendo u
    - $x = ayb$ porta da p a q lasciando la pila vuota
    - A_{pq} deriva x.
  - La pila viene svuotata negli stati intermedi fra p e q, sia r lo stato in cui la pila si svuota, essendo che la computazione da p a q richiede $k+1$ passi, allora le computazioni da p ad r e da r a q richiederanno al più k passi. Sia $x = yz$, y porta da p ad r, e z porta da r a q, per ipotesi A_{pr} deriva y e A_{rq} deriva z, ma allora, essendo che esiste la regola $A_{pq} \longrightarrow A_{pr}A_{rq}$ ne consegue che A_{pq} deriva $x = yz$.



□

I due claim dimostrano il lemma $\Longleftarrow$. ■
Essendo entrambi i lati dimostrati, il teorema sull'equivalenza fra PDA e CFG è dimostrato.

1.7.3 Il Pumping Lemma per le Grammatiche Acontestuali

Il teorema presentato in questa sezione mostra che esistono anche dei linguaggi non acontestuali, ossia, per i quali non esiste alcuna grammatica che li genera. Un esempio di linguaggio acontestuale è il seguente

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

Prima di presentare il teorema, è necessario introdurre il seguente fatto.

Claim : Sia G una CFG in forma normale Chomsky, data la forma delle regole, ogni albero di derivazione di G è binario, da ciò ne deriva che, preso un qualsiasi albero, se il cammino più lungo di tale albero è lungo i, allora allora la stringa generata sarà lunga al più 2^{i-1}.

Dimostrazione claim : Si dimostra per induzione su i

- **caso base** : $i = 1$, allora la derivazione è composta da una sola produzione, la stringa è lunga 1, infatti $2^{i-1} = 2^{1-1} = 2^0 = 1$.
- **ipotesi induttiva** : si assume sia vero per un generico $i > 1$.

- **passo induttivo** : si consideri un cammino lungo $k = i + 1 = (i > 1) + 1 \geq 3$, la prima regola applicata deve essere necessariamente del tipo

$$S \longrightarrow BC$$

i sotto alberi generati da B e C hanno il cammino più lungo (denominato da ora in poi *altezza*) grande al più i, generano quindi stringhe lunghe al più 2^{-1}, quindi S genera stringhe lunghe al più $2 \cdot 2^{i-1} = 2^i = 2^{k-1}$. ■

Teorema (Pumping Lemma per le CFG) : : Sia L un linguaggio acontestuale, allora esiste un numero p tale che, presa una qualsiasi stringa w lunga almeno p, ($|w| \geq p$), esiste una sua suddivisione

$$w = uvxyz$$

tale che

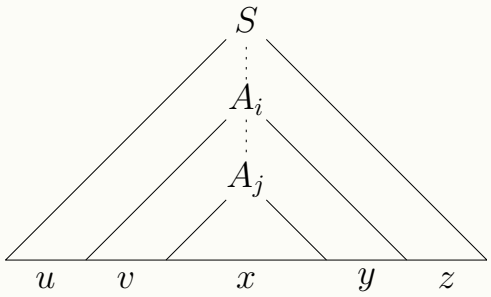
1. $\forall i \geq 0 \quad uv^ixy^iz \in L$
2. $|vy| > 0$
3. $|vxy| \geq p$

Dimostrazione (Pumping Lemma per le CFG) : Sia $G = (V, \Sigma, R, S)$ la CFG associata ad L, ossia $L(G) = L$, si assume che G sia in forma normale Chomsky (senza perdita di generalità), quindi ogni suo albero di derivazione sarà binario. Sia $m = |V|$ il numero di variabili distinte di G, identifichiamo come *pumping* il numero $p = 2^m$. Sia w una generica stringa tale che $|w| \geq p = 2^m$, allora w avrà un albero di derivazione di altezza almeno $m + 1$, ed il numero di nodi nell'albero sarà almeno $m + 2$, di cui 1 nodo è necessariamente un terminale, e gli altri $m + 1$ sono variabili.

Sia $w = uvxyz$, siccome le variabili distinte sono m, esiste una variabile A_i che si ripete nell'albero, in particolare

- L'albero con radice A_i genera vxy
- L'albero con radice $A_j = A_i$ genera x

Mentre u e z sono generate nella derivazione da S ad A_i.



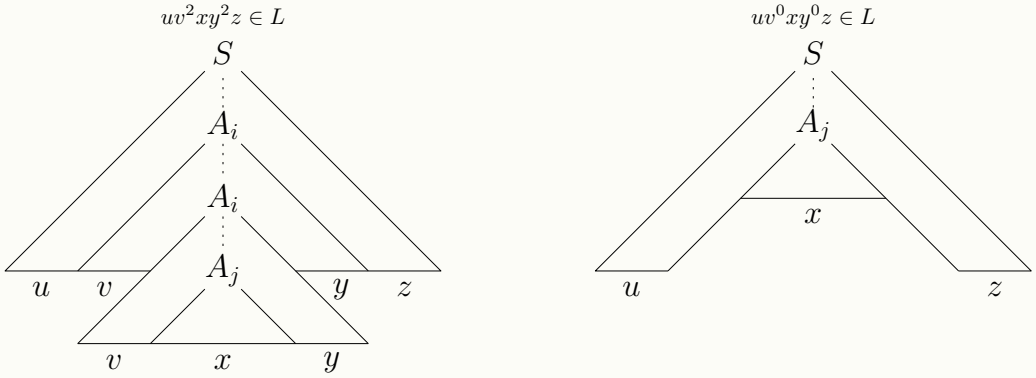
Siccome G è in forma normale, un sotto albero contiene, o un singolo terminale, o due variabili. Quindi, essendo che il sotto albero di A_i non può essere un terminale, avendo A_j, sarà composto da due variabili.

$$A_i \longrightarrow BC \in R$$

Una delle due variabili fra B e C genererà A_j che a sua volta deriverà x. L'altra variabile deriverà vy, quindi $vxy \neq x$, ciò implica che necessariamente $|vy| > 0$ (punto (ii) dimostrato).

Essendo che il cammino più lungo nell'albero ha lunghezza $m+1$, le stringhe generate avranno lunghezza minore o uguale a $2^{(m+1)-1} = 2^m = p$, quindi $|vxy| \geq p$ (punto (iii) dimostrato).

Le variabili A_i e A_j possono essere sostituite nell'albero di computazione.



In generale, è chiaro come $\forall i\ uv^ixy^iz \in L$. (punto (i) dimostrato), la dimostrazione del teorema è completa. ■

1.7.4 Esercizi ed Ultime Proprietà sulle CFG

In questa sezione verrà utilizzato il pumping lemma per dimostrare che alcuni linguaggi non sono acontestuali.

Esercizio 1

Si consideri il linguaggio

$$L = \{0^n1^n2^n \mid n \geq 1\}$$

Tale linguaggio non è acontestuale, non esiste nessuna CFG che lo genera. Per assurdo, si assuma che L sia acontestuale, allora $\exists p$ tale che, per ogni stringa $w \in L$ di lunghezza al più p, valgono le condizioni del pumping lemma.

Si prende in esame la stringa $w = 0^p1^p2^p$, bisogna considerare tutte le scomposizioni di w del tipo

- $w = uvxyz$ con
- $|vy| > 0$
- $|vxy| \leq p$

$$w = \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}\ \underbrace{22\ldots2}_{p \text{ volte}}$$

Se la stringa vxy è lunga al più p conterrà 1 oppure 2 caratteri distinti, dato che, se contenesse 3 caratteri distinti (sia 0 che 1 che 2) allora per costruzione di w sarebbe lunga più di p. Si consideri il punto (1) del pumping lemma, preso $i = 0$ la stringa

$$\hat{w} = uv^0xy^0z$$

Deve essere contenuta in L. Siccome la stringa vxy per definizione non era vuota, la stringa $\hat{w} = uxz$ avrà un numero di elementi minore di w dato che v ed y sono state rimosse ed insieme erano composte da almeno 1 carattere.

La rimozione di vy comporta il cambio del numero di occorrenze di 1 (oppure 2) simboli in $\hat{w}$ rispetto a w, quindi $\hat{w}$ avrà 1 (oppure 2) simboli le cui occorrenze differiscono dal/dai restante/restanti, ma allora non è del tipo $0^n1^n2^n,\ \ n \geq 1$, quindi non è in L, ma allora non è vero che L è acontestuale.

Esercizio 2

Si consideri il linguaggio

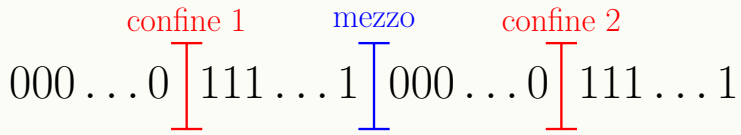
$$L = \{ww \mid w \in \{0,1\}^*\}$$

Tale linguaggio non è acontestuale, non esiste nessuna CFG che lo genera. Per assurdo, si assuma che L sia acontestuale, allora $\exists p$ tale che, per ogni stringa $w \in L$ di lunghezza al più p, valgono le condizioni del pumping lemma.

Si considera la stringa

$$w = 0^p 1^p 0^p 1^p$$

di cardinalità $|w| = 4p$, bisogna considerare ogni possibile modo di scomporre la stringa, come prima cosa, si individuano le seguenti sezioni nella stringa



La stringa può essere scomposta in diversi modi

- *caso 1* : vxy non si trova a cavallo fra i confini e non si trova a cavallo nel mezzo, è quindi confinata fra una delle 4 metà della stringa, è quindi composta di soli 0 oppure di soli 1. Secondo il punto (1) del pumping lemma, preso $i = 2$ si ha la stringa

  $$\hat{w} = uv^2xy^2z$$

  ma allora $\hat{w}$ non appartiene ad L.

- *caso 2* : vxy si trova a cavallo fra uno dei due confini, ma non tocca il mezzo. Si considera $i = 0$:

  $$\hat{w} = uv02xy^0z = uxz$$

  eliminando vy, si crea un "buco" rispetto alla stringa originale, tale buco ha dimensione al più p, essendo che una delle due metà della stringa cambia, il mezzo si sposta, supponiamo che vxy si trovava sul confine destro (la dimostrazione è analoga al sinistro), allora il centro si sposterà verso destra di $\geq p$ posizioni, e la metà destra della stringa terminerà necessariamente con un 1, ma la metà sinistra terminerà necessariamente con uno zero, quindi le due metà non sono uguali e $\hat{w} \notin L$.

- *caso 3* : vxy si trova a cavallo sul mezzo ma non oltrepassa alcun confine. Si considera, preso $i = 0$

  $$\hat{w} = uxz$$

  In tal modo, la seconda e la terza sezione di w verrà modificata, ed una delle due parti avrà meno di p caratteri

  $$w = \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}\ \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}$$

  $$\hat{w} = \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{k \text{ volte}}\ \underbrace{00\ldots0}_{j \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}$$

  $$k \neq p \vee j \neq p$$

  Ma allora $\hat{w} \notin L$.

Durante dimostrazioni di questo tipo può risultare utile l'applicazione di una proprietà delle CFG, ossia la loro **chiusura per concatenazione**, siano $G_1 = (V_1, \Sigma, R_1, S_1)$ e $G_2 = (V_2, \Sigma, R_2, S_2)$ due CFG, la grammatica $G = (V_1 \cup V_2, \Sigma, R, S)$ dove

- S è un nuovo stato
- $R = R_1 \cup R_2 \cup \{S \to S_1S_2\}$

é ancora una CFG.

Differentemente, le CFG non sono chiuse per intersezione, basta dare un singolo contro esempio per dimostrare la tesi, si considerino i seguenti linguaggi

$$L_1 = \{0^n1^n2^i \mid n \geq 0 \wedge i \geq 0\}$$
$$L_2 = \{0^k1^n2^n \mid n \geq 0 \wedge k \geq 0\}$$

Entrambi i linguaggi sono acontestuali, infatti sono generati dalle regole

$$R_1 = \begin{cases} S \longrightarrow S_1S_2 \\ S_1 \longrightarrow 0S_11|\epsilon \\ S_2 \longrightarrow 2S_2|\epsilon \end{cases} \qquad R_2 = \begin{cases} S \longrightarrow S_1S_2 \\ S_1 \longrightarrow 0S_1|\epsilon \\ S_2 \longrightarrow 1S_22|\epsilon \end{cases}$$

L'intersezione dei due linguaggi $L = L_1 \cup L_2 = \{0^n1^n2^n \mid n \geq 0\}$ sappiamo essere un linguaggio acontestuale, quindi le CFG **non sono chiuse per intersezione**.

Esercizio 4

Si consideri il seguente linguaggio

$$L = \{a, b\}^* \setminus \{ww \mid w \in \{a, b\}^*\}$$

Comprende tutte le stringhe che non possono essere scritte come una stessa stringa ripetuta due volte. Nell'esercizio 1.7.4 si è visto che il complementare di L, non è un linguaggio acontestuale, si vuole dimostrare che invece L lo è. Sicuramente, tutte le stringhe di cardinalità dispari saranno in L. Le regole che generano il linguaggio sono le seguenti

$$R = \begin{cases} S \longrightarrow A|B|AB|BA \\ A \longrightarrow a|aAa|aAb|bAa|bAbB \longrightarrow b|aBa|aBb|bBa|bBb \end{cases}$$

- A genera le stringhe dispari con una a al centro
- B genera le stringhe dispari con una b al centro

Dimostrazione : : Si vuole dimostrare che le stringhe di lunghezza pari generate da R sono in L. Si dimostreranno entrambi i versi.

$\boxed{\implies}$: Se $x \in L$, allora $S \Rightarrow^* x$. Se $x \in L$, allora esiste almeno una lettera che si differenzia fra le due sotto stringhe di eguale lunghezza che compongono x, sia $n = |x|$

$$\exists\ i \mid x_i \neq x_{n/2+1}$$

x può essere scritta come unione di due stringhe di lunghezza dispari $x = uv$. Si pongono

- $u = x_1x_2 \dots x_{2i-1}$
- $v = x_{2i}x_{2i+1} \dots x_n$

$S \longrightarrow AB$ e $A \longrightarrow u \wedge B \longrightarrow v \implies S \Rightarrow^* x$.

$\boxed{\impliedby}$: Se $S \Rightarrow^* x$ allora $x \in L$. Siccome $|x| = n = 0 \mod 2$, può essere generata a partire da $S \longrightarrow AB$ oppure $S \longrightarrow BA$, si suppone che sia generata dalla prima (il procedimento è analogo), allora $x = uv$ con $u \neq v$ e

- $A \Rightarrow^* u$
- $B \Rightarrow^* v$

Sia $|u| = l$ e $|v| = n - l$, le lettere centrali di u e v sono

- $u_{\frac{l+1}{2}}$
- $v_{\frac{n-2+1}{2}}$
- dove $u_{\frac{l+1}{2}} \neq v_{\frac{n-2+1}{2}}$

Scrivendo le lettere in funzione di x si ha

$$u_{\frac{l+1}{2}} = x_{\frac{l+1}{2}} \qquad v_{\frac{n-2+1}{2}} = x_{\frac{n+l+1}{2}}$$

dimostrazione non completa

CAPITOLO

2

CALCOLABILITÀ

2.1 Macchina di Turing e Decidibilità

Esistono linguaggi che nessuna CFG può accettare, si vuole estendere il modello di calcolo ad uno più potente. Negli anni 30' del ventesimo secolo fu introdotta la **Macchina di Turing** (alla quale ci riferiremo come TM), estendendo gli automi dandogli una memoria illimitata, tale modello è una semplice astrazione dei calcolatori odierni, e corrisponde alla nozione di algoritmo.

Caratteristiche di una TM

- Una TM possiede un *nastro di lavoro*, rappresenta una memoria (illimitata) sulla quale il modello può scrivere dei caratteri.
- Una *testina di lettura* che identifica la precisa posizione attuale sul nastro, che può spostarsi a destra o a sinistra.
- Vi sono poi degli *stati di accettazione* (la stringa in input è accettata dalla TM) e degli *stati di rifiuto* (la stringa in input è rifiutata dalla TM). Quando la TM passa su uno stato di rifiuto, la computazione è immediatamente terminata.
- È possibile che per alcuni input una TM entri in uno stato di *loop* in cui non termina.

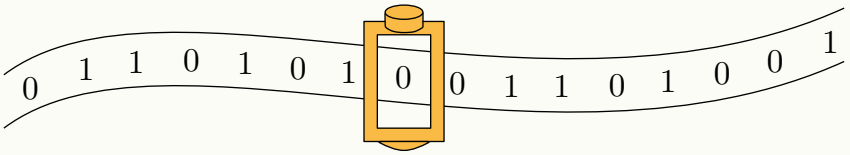
Definizione (Turing Machine) : Una TM $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ è una tupla tale che

- Q è l'insieme degli stati
- Σ è l'alfabeto delle stringhe in input
- Γ è l'insieme dei caratteri che possono essere scritti sul nastro, solitamente $\Sigma \subseteq \Gamma$. Inoltre in Γ vi è sempre un carattere speciale $\sqcup$ (denominato "blank") che rappresenta il carattere vuoto. Inoltre $\sqcup \notin \Sigma$. δ è la funzione di transizione definita

  $$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

  L'insieme $\{L, R\}$ rappresenta i possibili spostamenti della testina a sinistra o a destra
- q_0 è lo stato iniziale
- q_{acc} è lo stato (unico) di accettazione
- q_{rej} è lo stato (unico) di rifiuto

Canonicamente, la configurazione iniziale di una TM prevede l'intera stringa in input contenuta nel nastro, seguita dal carattere ⊔.



Una TM computa la stringa in input seguendo le regole definite dalla δ, come per gli automi, per una TM è defintia la configurazione ad un certo passo nella configurazione, essa determina il contenuto del nastro, la posizione della testina, e lo stato attuale. Una configurazione si denota

$$uqav$$

dove

1. $u, v \in \Gamma^*$
2. $q \in Q$
3. uav è il contenuto del nastro, $a \in \Gamma$ è il carattere su cui si trova la testina.

Data in input una stringa w, la configurazione iniziale sarà $q_0 w$. Si può rappresentare graficamente una configurazione come segue

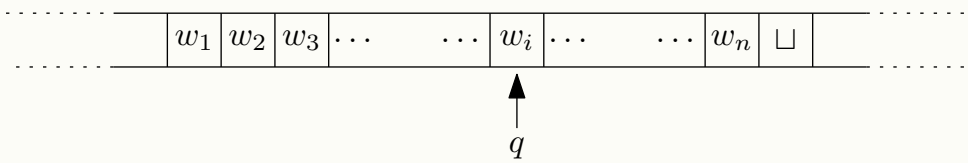


Figura 2.1: la testina è sul carattere w_i e lo stato attuale è q

Per definire il concetto di accettazione, bisogna stabilire la relazione di *produzione* :

$$uaq_ibv \text{ produce } uq_jacv$$
$$\text{se e solo se}$$
$$\delta(q_i, b) = (q_j, c, L)$$

Vuol dire che la TM, nello stato q_i, leggendo con la testina il carattere b si sposta a sinistra. Può essere analogamente definito per lo spostamento a destra.

$$uaq_ibv \text{ produce } uacq_jv$$
$$\text{se e solo se}$$
$$\delta(q_i, b) = (q_j, c, R)$$

Diremo che una TM **accetta** w se e solo se esiste una sequenza di configurazioni

$$C_1 \to C_2 \to C_3 \cdots \to C_k$$

dove

- $C_1 = q_0 w$
- $\forall i \quad C_i$ produce C_j
- lo stato della configurazione C_k è lo stato accettante $C_k = uq_{acc}av$

Definizione (Riconoscibilità) : Un linguaggio L è **turing riconoscibile** se esiste una TM M che *accetta* ogni sua stringa, si dice che L è il linguaggio di M.

Se una TM deve computare una stringa che non è nel suo linguaggio, può

- rifiutare
- andare in loop

Una TM M per cui, data una qualsiasi stringa, non vai mai in loop, viene detta **decisore**.

Definizione (Decidibilità) : Un linguaggio L è **turing decidibile** se esiste una TM M che è un decisore per L, ossia, per ogni sua stringa, non va mai in loop. Si dice che M *decide* L.

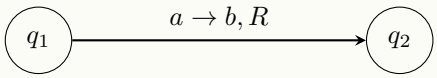
Data una TM M si hanno gli insiemi

- $L(M) = \{w \in \Sigma^* \mid M \text{ accetta } w\}$
- $R(M) = \{w \in \Sigma^* \mid M \text{ rifiuta } w\}$
- Generalmente $L(M) \cup R(M) \subseteq \Sigma^*$
- Se $L(M) \cup R(M) = \Sigma^*$ allora M è un decisore.

Un linguaggio che non ha decisori *non è decidibile*, la definizione di decidibilità stabilisce i limiti della computabilità, esistono infatti dei linguaggi (astrando, dei problemi) che non possono essere decisi (non possono essere risolti), ciò si lega inevitabilmente con i *teoremi di incompletezza* di Gödel, esisteranno sempre delle proposioni per cui è *impossibile* stabilire se sono vere o false.

2.1.1 Esempi di TM

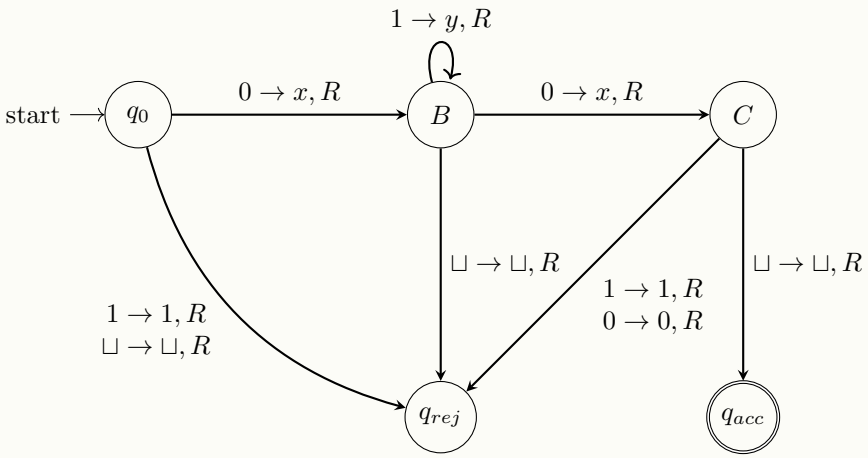
Le TM verranno rappresentato in maniera compatta sottoforma di grafi proprio come per gli automi.



Il grafo rappresentato in figura descrive la seguente situazione : Se la TM si trova nello stato q_1, e la testina si trova sul carattere a, allora si sostituisce il carattere a nel nastro con il carattere b, si sposta la testina a destra (se al posto di R ci fosse stato L si sarebbe andati a sinistra) e la TM si sposta nello stato q_2.

Esempio 1

Si consideri il seguente linguaggio $L = \{01^*0\}$. La TM che decide L è la seguente

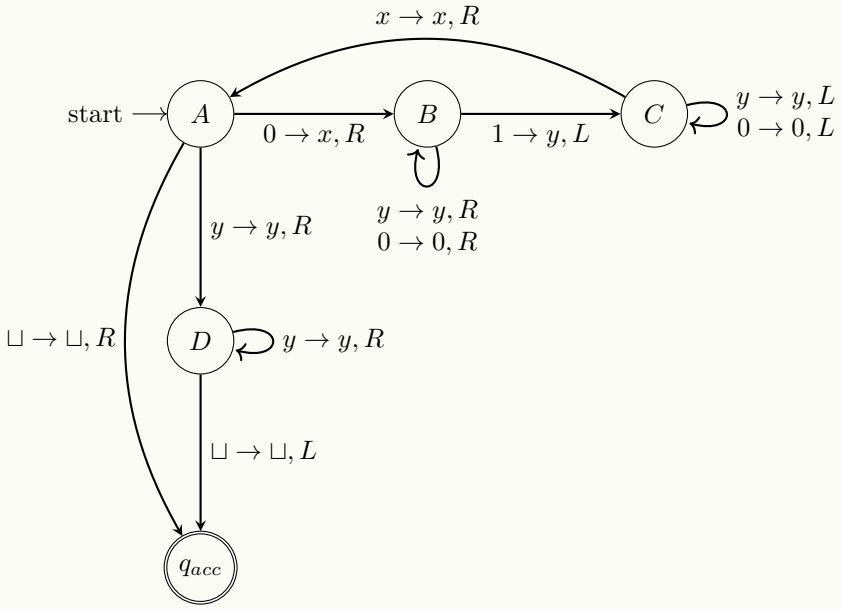


Esempio di computazione su una stringa di L:

$$\bar{0}1110\sqcup \quad x\bar{1}110\sqcup \quad xy\bar{1}10\sqcup$$
$$xyy\bar{1}0\sqcup \quad xyyy\bar{0}\sqcup \quad xyyyx\bar{\sqcup}$$

Esempio 2

Si consideri il seguente linguaggio $L = \{0^n 1^n \mid n \geq 0\}$. La TM che decide L è la seguente (stato di rifiuto omesso)



Nel capitolo precedente si è visto come il linguaggio $L = \{0^n 1^n 2^n \mid n \geq 0\}$ non è acontestuale, è possibile computarlo tramite le TM, de facto, basterà unire 2 TM che si comportino come quella vista nell'esempio 2.1.1, infatti nella prima computazione si occuperà di controllare che la stringa in input abbia i primi $2n$ caratteri del tipo $0^n 1^n$.

$$00\ldots0 \quad 11\ldots1 \quad 22\ldots2 \quad \text{viene trasformata} \quad xx\ldots x \quad yy\ldots y \quad 22\ldots2$$

La seconda TM si occuperà di controllare se gli ultimi $2n$ caratteri sono del tipo $y^n 2^n$.

$$xx\ldots x \quad yy\ldots y \quad 22\ldots2 \quad \text{viene trasformata} \quad xx\ldots x \quad xx\ldots x \quad yy\ldots y$$

Senza perdita di generalità, è possibile astrarre le TM considerano due nuovi modelli equivalenti

- **TM multinastro**
- **TM non deterministica**

Prima di introdurli, si consideri il seguente esempio di astrazione di una TM, ossia di una macchina che, ad ogni passo di computazione, piuttosto che spostarsi necessariamente a destra o a sinistra, può rimanere ferma, la denominiamo STM.

La definiamo M', e la sua funzione di transizione sarà del tipo

$$\delta' : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

Dove S (che sta per 'stay') indica l'azione del restare fermi senza muovere la testina.

Proposizione : Per ogni STM esiste una TM classica equivalente.

Dimostrazione : ,Sia M' la STM, l'idea è quella di considerare una TM M che gestisca le transizioni di stato in cui la testina non si muove, a tal scopo, basta simulare tale azione tramite una sequenza di 2 movimenti che fanno uso di uno stato ausiliario q_s.

$$\delta'(q, a) = (p, b, S)$$
è equivalente alle azioni
$$\begin{cases} \delta(q, a) = (p, q_s, L) \\ \delta(q_s, *) = (p, *, R) \end{cases}$$

dove $*$ rappresenta un qualsiasi elemento di Γ.

2.1.2 TM multi nastro

Introduciamo la macchina di Turing con più nastri, che verrà denotata MTM, sia k il numero di nastri, ogni nastro avrà una testina proprietaria, uno stato sarà quindi rappresentato dalle stringhe scritte su tutti e k i nastri, e la relativa posizione delle testine.

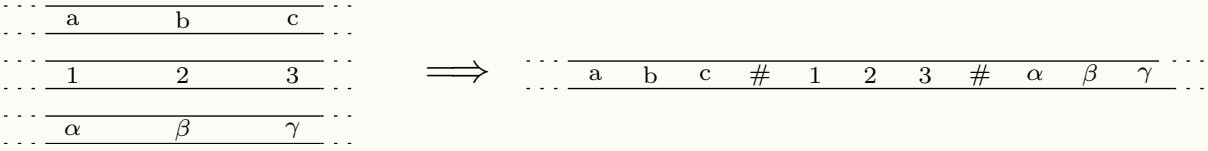
La funzione di transizione prenderà le decisioni valutando le posizioni ed il valore di tutte le testine

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

Teorema (MTM≡TM) : Per ogni MTM esiste una TM equivalente.

Dimostrazione : La dimostrazione, piuttosto che formale, mostrerà la costruzione di una TM che si comporta come la generica MTM.

Sia MM la macchina multi nastro, ed M la macchina classica che deve simularla, si introduce un nuovo carattere $\# \notin \Gamma$ che verrà utilizzato nel nastro di M come separatore per i differenti nastri originari di MM.



Per simulare le k testine, si implementa la possibilità per M di *marcare* i caratteri (in tal caso, con un punto come apice). I caratteri marcati indicano che la testina è posta su di essi. Sia Γ' l'alfabeto dei nastri di MM, e Γ l'alfabeto del nastro di M.

$$\forall a \in \Gamma', \quad \exists \dot{a} \in \Gamma$$

Si descrive ora la computazione di M, data una stringa in input w la configurazione iniziale è la seguente:

$$\#\dot{w}_1 w_2 w_3 \ldots, w_n \#\underbrace{\dot{\sqcup}\#\dot{\sqcup}\ldots\#\dot{\sqcup}}_{k-1 \text{ volte}}$$

Nel *passo di computazione*, si scansiona una prima volta il nastro partendo dal primo simbolo #, leggendo tutti i caratteri marcati, considerando poi la funzione di transizione, si esegue una seconda scansione aggiornando i valori delle testine (le marcature dei caratteri) ed il contenuto dei k caratteri in questione. Se la testina su uno dei nastri deve spostarsi su #, allora M sposta il contenuto dell'intero nastro a destra di 1 posizione.

Corollario : L è turing riconoscibile/decidibile se e solo se esiste una MTM che lo riconosce/decide.

2.1.3 TM non deterministiche

Si arricchisce il modello della macchina di turing introducendo il non determinismo, denotandolo NTM, la funzione di transizione cambia definizione

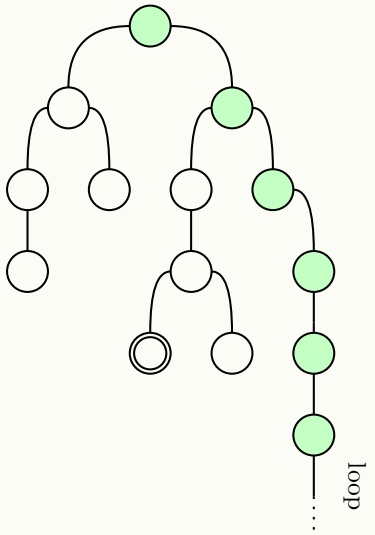
$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Per ogni input si diramano più vie di computazione, una NTM accetta una stringa in input se almeno un ramo di computazione è accettante.

Teorema (NTM≡TM) : Per ogni NTM esiste una TM equivalente.

Dimostrazione : La dimostrazione, piuttosto che formale, mostrerà la costruzione di una TM che si comporta come la generica NTM.

Sia N la NTM in questione, e sia M la TM classica che la vuole simulare. M, per essere equivalente, dovrà esplorare ogni ramo di computazione, precisamente, deve esplorarlo in altezza. L'esplorazione in profondità in presenza di un loop su un ramo di computazione, comporterà il loop su M, anche se l'originaria N aveva uno stato accettante su un differente ramo.



Idea algoritmica : M utilizzerà 3 nastri

1. il primo nastro conterrà la stringa in input che non verrà modificata (il riferimento va mantenuto)
2. il secondo nastro sarà il nastro di lavoro che verrà modificato per esplorare un cammino emulando la NTM.
3. il terzo nastro conterrà l'indirizzo del nodo dell'albero che si sta esplorando.

Nell'i-esimo step, identificato da una tripla

$$(M, w, i)$$

usando l'input del primo nastro si percorre il ramo simulando N, se viene trovato uno stato accettante, M accetta, altrimenti si incremente l'indice i e si ripercorre.

- Se un ramo di N è accettante, M accetta
- Se ogni ramo di N rifiuta, M rifiuta
- Se almeno un ramo di N va in loop, e nessun ramo accetta, M va in loop.

  Se una NTM è un decisore, tutti i cammini hanno lunghezza finita

■

2.1.4 L'Enumeratore

L'enumeratore è un modello (in particolare, una variante di una TM) capace di generare un preciso insieme di stringhe, in un qualsiasi ordine e con eventuali ripetizioni, in particolare, per ogni linguaggio turing-riconoscibile, esiste un enumeratore che lo genera.

Un enumeratore è una TM che "stampa" delle stringhe, eventualmente, infinite.

Teorema : Un linguaggio è turing riconoscibile se e solo se esiste un enumeratore che lo genera.

Dimostrazione : Si dimostrano separatamente le due direzioni.

$\boxed{\Longleftarrow}$: Dato un enumeratore E, definiamo una TM M come segue (sia w l'input)

- Si esegue E, se stampa una stringa, si confronta con w
- Se w è uguale ad una stringa stampata da E, allora M accetta.

$\boxed{\Longrightarrow}$: Sia M una TM, si considera un enumeratore E che deve stampare $L(M)$. Sia Σ l'alfabeto di $L(M)$, identifichiamo come segue

$$\Sigma^* = \{\epsilon, s_1, s_2 \dots\}$$

la lista di tutte le possibile stringhe su tale alfabeto. E sarà definito come segue

- Si ripetono i passi per $i = 1, 2 \ldots$
  - Si esegue M per i passi su ogni input $s_1, s_2 \ldots s_i$
  - Se una qualsiasi computazione su s_j accetta, E stampa s_j

■

2.2 Indecidibilità

Come esistono linguaggi non regolari e linguaggi non acontestuali, risulta naturale porsi la domanda : Esistono dei linguaggi non turing riconosibili? Il modello della TM corrisponde al concetto di algoritmo, un linguaggio non riconoscibile corrisponde ad un problema che *nessun* algoritmo può risolvere.

Una TM può ricevere in input un qualsiasi oggetto matematico, come un polinomio, un DFA, o direttamente un'altra TM, è importante che tale oggetto O, venga **codificato** in binario, denotandolo $< O >$ prima di essere computato dalla TM.