Informatica Teorica

A cura di :

Carmelo D'Oro
Ettore Nicosia
Sandro La Bruzzo

Docente : ***Prof. Antonio Restivo***

Informatica Teorica

Lezione 1 3/10/2002:

Algoritmo: Quando ci si trova davanti a problemi di qualsiasi tipo,si tenta di risolverli cercando di esprimere una successione finita di istruzioni, interpretabili ed eseguibili da un esecutore,che partendo da delle informazioni iniziali permettano di arrivare ad un risultato finale.
Tesi di Church Turing: Tutto ciò che è calcolabile e realizzabile in un algoritmo, è realizzabile in ogni formalismo.
Il linguaggio Pascal è una formalizzazione dell'algoritmo, quindi tutto ciò che è calcolabile in Pascal è calcolabile in ogni linguaggio.
La **teoria della Calcolabilità** è fatta da tanti modelli che cercano di formalizzare il concetto di Algoritmo, tra i quali:
- Macchina di Turing
- Macchina a Registri
Macchina di Turing: Per Turing la nozione di Algoritmo non ha alcun legame con la natura dell'oggetto che si sta trattando, quello che è importante è che l'algoritmo funzioni sempre indipendentemente dalla semantica dell'oggetto.
L'approccio di Turing è che quando si fanno calcoli si manipolano simboli, quindi nel modello di Turing il concetto di Algoritmo non è legato alla natura dell'oggetto.
Una macchina di Turing è costituita da:

- Nastro(supporto fisico) potenzialmente infinito o meglio "lungo quanto lo voglio"
- Alfabeto finito(che si indica con Σ)
- Insieme Finito di stati(che si indica con Q)

La Parte fisica della macchina è costituita da un occhio centrale, detto corpo della macchina, che punta un nastro suddiviso in caselle o celle elementari.In base al simbolo che compare sul nastro e allo stato *q* in cui si trova, la macchina compierà delle azioni.
Una macchina di Turing, svolge

- Azioni elementari (spostamento a destra o a sinistra)
- Scrivere, ossia modificare un simbolo con un altro. ATTENZIONE:scrivere può essere anche cancellare
- Cambiamento di stato

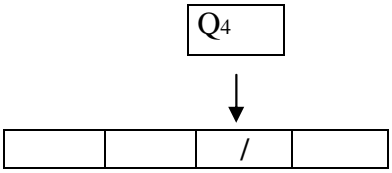
In ultimo dobbiamo definire la tabella delle istruzioni dove sono scritte tutte le azioni che la macchina di Turing deve svolgere quando il suo occhio punta ad un particolare simbolo nel nastro.
Definiamo come esempio una semplice Macchina di Turing a sei stati:
L'alfabeto e gli stati devono essere finiti e li definiamo così come segue:

Σ= {Blank(spazio vuoto),/,*}
Q={q_1,q_2,q_3, q_4,q_5,q_6}

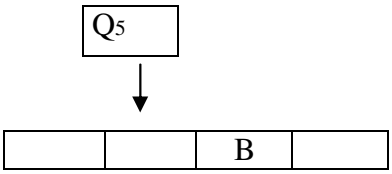
Gli spostamenti possibili saranno :
Spostamento a Sinistra =L Spostamento a destra =R
Definiamo la tabella delle istruzioni:

	B	/	*
Q_1			R q_2
Q_2		R	R q_3
Q_3	L q_4	R	
Q_4		B L q_5	B q_6
Q_5		L	/ q_6
Q_6			

La Macchina si trova in uno stato q_i , l'occhio legge il valore che compare sul nastro, in corrispondenza di queste due informazioni (stato-simbolo), compaiono nella tabella gli eventuali passi che la macchina deve svolgere.Ad esempio,nel caso in cui la macchina sia in uno stato q_4 e legga il simbolo /:



Si guarda la tabella in corrispondenza dello stato q4 e del simbolo / si trovano le operazioni che la macchina di Turing dovrà svolgere: B L q_5 (Scrivi il carattere Blank aggiorna lo stato a q_5 e spostati a sinistra) ottenendo così:

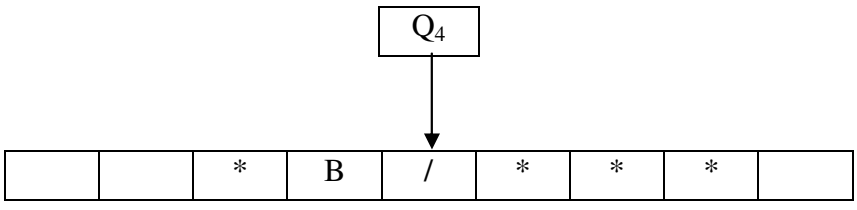


Lezione 2 4/10/2002

Configurazione istantanea di una Macchina di Turing : è esattamente la fotografia dello stato in cui si trova la macchina in un preciso istante ed è quindi formato da 3 elementi essenziali :

1. **Stato della macchina**
2. **Contenuto del nastro**
3. **Posizione della testa**

Un esempio di configurazione potrebbe essere questa :



La configurazione istantanea sintetica è quella che riguarda il contenuto del nastro dal primo simbolo diverso da B all'ultimo simbolo diverso da B.
Esistono 2 modi di descrivere una configurazione istantanea, ad esempio la configurazione precedente si può scrivere come

Q_4
↓
*B/***

oppure: BQ_4/***

Quest'ultimo tipo di descrizione è detta *Descrizione Istantanea*, dove abbiamo una stringa formata da $\mathbf{\Sigma \cup Q}$ con l'unica restrizione che **Q** compare una volta soltanto. Da quanto detto finora possiamo benissimo dire che, data una configurazione istantanea ne segue un'altra che è data dall'applicazione della tabella delle istruzioni. A questo punto possiamo dare una definizione più generica :

Definizione : Siano α e β due configurazioni con $\alpha \rightarrow \beta$ (significa che β segue α), si definisce **Calcolo di una Macchina di Turing** una successione finita $\alpha_1, \alpha_2, \dots, \alpha_n$ di configurazioni istantanee tale che $\alpha_i \rightarrow \alpha_{i+1}$ (i = 1,2,…, n-1) e dove α_n è detta **configurazione di arresto**, ossia non esiste nessuna b tale che $\alpha_n \rightarrow b$.

Possiamo definire α_1 come **Input** e α_n come **Output**.
La configurazione di arresto è dunque quella configurazione in cui macchina si arresta e non fa più niente. Questo concetto è molto importante poiché proprio per definizione di algoritmo, un calcolo deve arrivare ad un punto di fine.
Per fare un calcolo devo individuare una configurazione istantanea, dopodiché la macchina inizia a lavorare, passando attraverso gli stati, fino a trovare un risultato, cioè una stringa α_n, oppure non si trova nessun risultato e non si arriva ad una configurazione di arresto.

Esempio di Calcolo :
Diamo come Input una configurazione istantanea iniziale :

1. $\alpha_1 = q_1 * / / / * / /$, applicando la tabella otteniamo :
2. $\alpha_2 = * q_2 / / / * / /$
3. $\alpha_3 = * / q_2 / / * / /$
4. $\alpha_4 = * / / q_2 / * / /$
5. $\alpha_5 = * / / / q_2 * / /$
6. $\alpha_6 = * / / / * q_3 / /$
7. $\alpha_7 = * / / / * / q_3 /$
8. $\alpha_8 = * / / / * / / q_3 B$
9. $\alpha_9 = * / / / * / q_4 /$
10. $\alpha_{10} = * / / / * q_5 /$
11. $\alpha_{11} = * / / / q_5 * /$
12. $\alpha_{12} = * / / / q_6 / /$

Quindi alla fine il punto d'arresto sarà α_{12} , e come *Output* avremo la stringa * / / / / /
Se consideriamo la stringa del tipo

$$a = * / / / \dots /$$

può essere inteso come un numero non negativo n e quindi possiamo definire una funzione del tipo :

$$F(n_1, n_2, \dots, n_k) \rightarrow f(\underbrace{* / / . . /}_{n_1 \text{ volte}}, \underbrace{* / / . . /}_{n_2 \text{ volte}}, \dots, \underbrace{* / / . . /}_{n_k \text{ volte}})$$

Quindi se vogliamo possiamo interpretare l'input : * / / / * / / = (3,2)
e quindi l'output è : * / / / / / = (5)
Possiamo dunque intendere che questa macchina di Turing svolge la somma tra numeri interi non negativi.

Definizione :
Data la configurazione istantanea : $\alpha_1 = q_1$ $\underbrace{* / / . . /}_{t_1}$, $\underbrace{* / / . . /}_{t_2}$, … , $\underbrace{* / / . . /}_{t_k}$

Supponiamo che esista un calcolo α_1 … α_n tale che α_n = * / / … / , diciamo che $F(x_1 \ldots . x_k)$ è **Turing-calcolabile** se esiste una Macchina di Turing che lo calcola.
Tesi di Church-Turing : Tutto ciò che è calcolabile si può calcolare mediante la Macchina di Turing (si chiama tesi e non teorema perché non si può dimostrare).

Esempio di una Macchina di Turing che copia :
Dato un alfabeto Σ = {b,c,*,B} e data una stringa in input del tipo : bbcbcbcc, restituisce bbcbcbcc * bbcbcbcc.
Una volta definito il problema implementiamo la Macchina e quali azione deve svolgere. Una possibile soluzione sarebbe quella di scrivere l'asterisco dopo l'ultimo carattere, far ritornare l'occhio indietro, prendere un carattere alla volta e farlo scrivere dopo l'ultimo carattere scritto.
Sorge spontaneo un problema : Come fa la Macchina a ricordare qual è l'ultimo carattere copiato ?
Da qui la necessità di aggiungere all'alfabeto due nuovi simboli da cui :

$$\Sigma = \{b,c,y,z,*,B\}$$

Tali simboli sostituiranno quelli dati dicendo quali sono i caratteri copiati.
Diamo di seguito la tabella delle istruzioni :

Descrizione Informale	**Q**	**b**	**c**	**y**	**z**	**B**	*****
Piazza *	$\mathbf{q_1}$	R	R			* Lq_2	
Torna Indietro	$\mathbf{q_2}$	L	L	L	L	Rq_3	L
Trova b, c	$\mathbf{q_3}$	yRq_b	zRq_c	R	R		Lq_4
Copia b	$\mathbf{q_b}$	R	R	R	R	bLq_2	R
Copia c	$\mathbf{q_c}$	R	R	R	R	cLq_2	R
Cancella i segni	$\mathbf{q_4}$			bL	cL		

Inseriamo adesso una stringa in input e vediamo come questa macchina lavora :

1. $\alpha_1 = q_1\ b\ c\ c\ b\ c$
2. $\alpha_2 = b\ q_1\ c\ c\ b\ c$
3. $\alpha_3 = b\ c\ q_1\ c\ b\ c$
4. $\alpha_4 = b\ c\ c\ q_1\ b\ c$
5. $\alpha_5 = b\ c\ c\ b\ q_1\ c$
6. $\alpha_6 = b\ c\ c\ b\ c\ q_1$
7. $\alpha_7 = b\ c\ c\ b\ c\ q_1\ *$
8. $\alpha_8 = b\ c\ c\ b\ q_2\ c\ *$
9. $\alpha_9 = b\ c\ c\ q_2\ b\ c\ *$
10. $\alpha_{10} = b\ c\ q_2\ c\ b\ c\ *$
11. $\alpha_{11} = q_2\ b\ c\ c\ b\ c\ *$
12. $\alpha_{12} = q_2\ B\ b\ c\ c\ b\ c\ *$
13. $\alpha_{13} = q_3\ b\ c\ c\ b\ c\ *$
14. $\alpha_{14} = y\ q_b\ c\ c\ b\ c\ *$
15. $\alpha_{15} = y\ c\ q_b\ c\ b\ c\ *$
16. $\alpha_{16} = y\ c\ c\ q_b\ b\ c\ *$
17. $\alpha_{17} = y\ c\ c\ b\ q_b\ c\ *$
18. $\alpha_{18} = y\ c\ c\ b\ c\ q_b\ *$
19. $\alpha_{19} = y\ c\ c\ b\ c\ *\ q_b\ B$
20. $\alpha_{20} = y\ c\ c\ b\ c\ q_2\ *\ b$

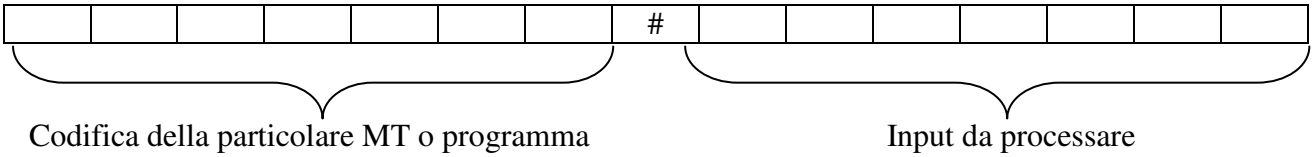
……..

Alla fine l'output sarà : b b c b c * b b c b c

Lezione 3 7/10/2002

Definizione : Una Macchina Universale di Turing è una Macchina che accetta e simula qualsiasi altra Macchina di Turing.

In generale si credeva che per ogni tipo di calcolo occorresse una macchina diversa di Turing, con questo teorema si dimostra invece che esistono macchine di Turing Universali.
Come è possibile tale processo?
Data una M.U.T., diamo in input la descrizione di una particolare Macchina di Turing seguita dall'input che si vuole processare.



Quindi la MUT leggerà l'input da processare ed eseguirà l'opportuna istruzione contenuta nel programma.
Ovviamente la MUT avrà un numero di stati fissati. Se vogliamo calcolare altre funzioni, non sarà la MUT a cambiare ma il programma.
Questo ci evidenzia l'idea di intercambiabilità esistente tra hardware e software.

Esistenza di problemi che non hanno soluzione algoritmica :
Vi sono alcuni problemi per cui non esiste un algoritmo e quindi una Macchina di Turing che li risolve. Per dimostrare tale esistenza si fa riferimento all'*Algoritmo Diagonale di Cantor*.
Supponiamo un insieme di tutti i programmi p_i (ad esempio realizzati in Pascal) ordinati in qualche maniera, ad esempio in ordine crescente: $p_1, p_2, \dots, p_n$ e più in particolare consideriamo la classe di funzioni $f\ N \rightarrow N$ tale che ogni programma p_i mi calcola una particolare f_i.
$p_1 \rightarrow f_1(1), f_1(2), \dots, f_1(n)$
$p_2 \rightarrow f_2(1), f_2(2), \dots, f_2(n)$
…
…
$p_n \rightarrow f_n(1), f_n(2), \dots, f_n(n)$

Ora costruiamoci una funzione che non sta nel precedente elenco, nel seguente modo :

$$f(k) = f_k(k) + 1$$

Questa funzione non sta nell'elenco perché, supponiamo che $F = f_j$, se vogliamo calcolare F(j) otteniamo che :

$$F(j) = f_j(j) + 1 \neq f_j(j)$$

Problema della fermata :
Supponiamo di avere una Macchina di Turing, esiste un algoritmo che ci permette di stabilire se, dopo un certo numero di passi tale macchina si ferma? O meglio, arriveremo ad avere una configurazione d'arresto?
Supponiamo che fino ad un determinato numero di passi non si sia fermata; in che momento posso dire con esattezza che la Macchina non si ferma?
Supponiamo di avere una Macchina di Turing così costituita :
INPUT : Descrizione della MT + configurazione iniziale
OUTPUT : Si = La Macchina di ferma; No = La Macchina non si ferma
Se le possibili configurazioni istantanee sono indefinite, infatti non c'è una Macchina capace di arrivare ad una configurazione d'arresto, allora il problema non ha soluzione. Per dimostrare che un problema non ha soluzione algoritmica, basta paragonarlo al problema della **Fermata di una Macchina di Turing**.
N.B. : Esistono casi particolari del problema della fermata che hanno soluzione algoritmica, come ad esempio la Macchina che calcola la somma.

Teorema della fermata :
Il *teorema della fermata* dice che non esiste alcun algoritmo che ci permette di dire a priori se una Macchina si ferma dopo un certo numero di passi.
Quindi quando mi trovo una Macchina di Turing, dando un input conosco istante per istante cosa fa, però se provo ad esprimere un comportamento generale, non posso dire nulla.
Esempio :
Definiamo un numero x_0 (input) e costruiamo una successione di numeri con la seguente regola che ci permette, dato un numero, di conoscere il suo successivo :

$$X_{i+1} = \begin{cases} x_i / 2, & \text{se } x_i \text{ è pari} \\ 3x_i + 1, & \text{se } x_i \text{ è dispari} \end{cases}$$

Se $x_n = 1$ mi fermo.
Se supponiamo $x_0 = 7$ otteniamo :

7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

Esiste un numero per cui tale calcolo non si ferma? La risposta non è stata trovata.
Questo tipo di problemi che non hanno soluzione algoritmica sono detti *problemi indecidibili*, mentre quando ci troviamo ad avere problemi con una soluzione algoritmica questi sono detti *problemi decidibili*.

Problema dell'equivalenza fra Macchine di Turing :
Esiste una Macchina di Turing che, date in input due macchine di Turing, mi dice se queste sono equivalenti?
La risposta è no, questo è un problema **indecidibile**.

Ricorsione :
Consideriamo l'insieme degli interi non negativi Z_+ , le funzioni *ricorsive* sono funzioni costituite da :

- Funzioni di base
  $S(x) = x + 1$ (funzione successore)
  $F(x) = c$ (funzione costante)
  $P_i (x_1,x_2,\ldots.x_n) = x_i$ (funzione proiezione)
- Un meccanismo che si chiama **composizione** con cui si costruiscono nuove funzioni a partire da quelle di base.
  Supposto che ho definito la funzione $g(x_1, x_2, \ldots ,x_k)$ che ha k argomenti, mi creo k funzioni di nome h con n argomenti :
  $h_1(x_1, x_2, \ldots ,x_n)$
  $h_2(x_1, x_2, \ldots ,x_n)$
  …
  $h_k(x_1, x_2, \ldots ,x_n)$
  Mediante la **Funzione Composta** $f(x_1, x_2, \ldots ,x_n)$ si ottiene che :
  $f(x_1, x_2, \ldots ,x_n) = g(h_1(x_1, x_2, \ldots ,x_n), h_2(x_1, x_2, \ldots ,x_n), \ldots , h_k(x_1, x_2, \ldots ,x_n))$

  Esempio : *Meccanismo* $\rightarrow$ x + y scrivo +(x,y)
  x * y scrivo *(x,y)
  $f(x,y,z) = +(x,*(y,z))$
  $g(x_1, x_2) = +(x_1, x_2)$
  $h_1(x_1, x_2, x_3) = P_1(x_1, x_2, x_3)$
  $h_2(x_1, x_2, x_3) = *(x_2, x_3)$
  Possiamo definire quindi $f(x_1, x_2, x_3) = g(h_1(x_1, x_2, x_3), h_2(x_1, x_2, x_3)) = +(x_1,*(x_2,x_3))$
- **Esempio di ricorsione :** supponiamo di avere la funzione successore $S(x) = x + 1$ e la funzione proiezione $p_1(x_1, x_2) = x_1$; definiamo la funzione somma $+(x_1, x_2)$ nel seguente modo :

  $$\begin{cases} +(x_1, 0) = p(x_1, 0) = x_1 \\ +(x_1, S(x_2)) = +(x_1, x_2) + 1 = S(+(x_1, x_2)) \end{cases}$$

  Tale funzione va calcolando se stessa per valori di x_2 sempre più piccoli, fino a che non si verifica la condizione $x_2 = 0$, i cui il risultato è noto, dopodiché procedendo all'inverso, va calcolando i successori fino a trovare il risultato riguardante i parametri iniziali. Altresì possiamo dire che la funzione procede trovando il valore di x_1 per poi procedere a ritroso calcolando via via il successore del risultato della funzione precedente $S(x_1)$, $S(S(x_1))$, etc…
  Utilizzando una funzione più familiare tale esempio può essere scritto nella forma :

  $$\begin{cases} f(x_1, 0) = x_1 \\ f(x_1, succ(x_2)) = succ(f(x_1, x_2)) \end{cases}$$

Definizione :
La famiglia delle funzioni ricorsive è data da quelle funzioni che si ottengono dalle funzioni base applicando un numero finito di volte tutti i meccanismi, ossia *composizione* e *ricorsione*.
All'inizio si pensò che la classe delle funzioni calcolabili coincidesse con quella delle funzioni ricorsive, questa tesi però, negli anni '30 fu corretta da Ackermann, il quale costruì una funzione detta appunto **di Ackermann**, che è calcolabile, ma che non si può costruire mediante il meccanismo delle funzioni ricorsive. Quindi ciò che abbiamo definito come *funzioni ricorsive*,

vennero dette **funzioni ricorsive primitive**, mentre, per definire le funzioni ricorsive uguali a quelle calcolabili, fu introdotto il meccanismo della *minimizzazione*.

Minimizzazione :
Sia data una funzione $f(x_1, x_2, \dots, x_n)$; questa può essere espressa nella forma :

$$f(x_1, x_2, \dots, x_n) = (\mu z)[g(x_1, x_2, \dots, x_n, z) = 0] \text{ ,con } z \in Z_+$$

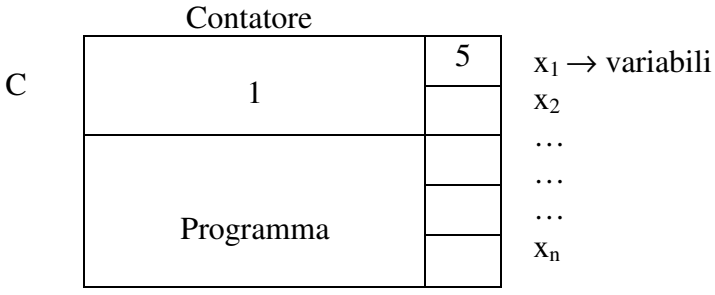
dove il valore di f è il più piccolo valore di z per cui $g(x_1, x_2, \dots, x_n, z) = 0$.

Esempio di Minimizzazione **:**
f(5) = 0… se g(5,0) = 0 altrimenti
f(5) = 1… se g(5,1) = 0 altrimenti
f(5) = 2… se g(5,2) = 0 altrimenti ecc…
… f(5) = non definita se $g(5,z) \neq 0 \ \forall z \in Z_+$
Questo meccanismo *definitorio* introduce una novità : anche se la funzione ha tutti gli argomenti definiti, può dare delle funzioni non definite (delle funzioni sono **non definite** se la Macchina non si ferma, cioè non ha una condizione d'uscita).
Per calcolare la funzione di Ackermann non devo avere funzioni non definite.

Teorema : Le Funzioni Turing-calcolabili sono tutte le funzioni ricorsive.
Le funzioni ricorsive sono quelle che si ottengono dalle funzioni base usando :
- Composizione
- Ricorsione
- Minimizzazione

Lezione 5 11/10/2002

Macchina a Registri
La Macchina a Registri è una Macchina caratterizzata da certi registri che sono locazioni di memoria ciascuno dei quali ha dei nomi ed indirizzi, e possono contenere valori interi non negativi.



In questo caso abbiamo n registri. Per modificare il contenuto dei registri abbiamo bisogno di definire delle *istruzioni* che nel loro insieme sono un **linguaggio di programmazione**. Con questa tecnica è possibile definire diversi linguaggi di programmazione, a seconda del tipo di linguaggio possiamo distinguere diverse Macchine a Registri, tra i vari linguaggi si distinguono il **GOTO** e il **WHILE**.

Linguaggio GOTO:
In generale, in un linguaggio di programmazione, si definiscono due aspetti principali :
- Aspetto sintattico
- Aspetto semantico

Iniziamo ad analizzare la sintassi del linguaggio :

- Simboli di variabili : x, y, z, …
- Numerali : stringhe con alfabeto {0,1,2,3,4,5,6,7,8,9}
- Istruzioni :
  - Assegnazioni di costanti, ad esempio "x:=14" modifica il contenuto del registro x
  - Assegnazioni di variabili, ad esempio "x:=y" copia il contenuto di y in x
  - Incr(x) incrementa di un'unità il valore di x
  - Decr(x) decrementa di un'unità il valore di x
- Istruzioni di controllo :
  - GOTO j : salto incondizionato con j etichetta
  - If x = 0 GOTO j : questa istruzione fa un salto incondizionato con j etichetta se il contenuto del registro x è uguale a 0.

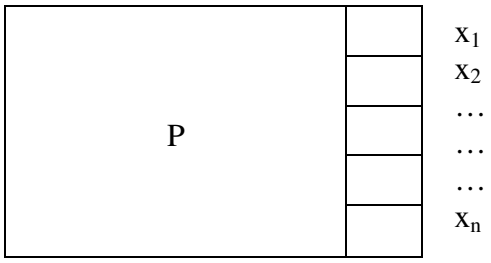
Possiamo fare un'altra aggiunta a tale macchina a Registri specificando che un registro c contiene il numero delle istruzioni da eseguire.

Esempio di programma :

1. If y = 0 GOTO 5;
2. Incr(x);
3. Decr(y);
4. Goto 1 ;
5. z :=x ;

Mettendo all'interno dei Registri x e y un intero, alla fine troveremo nel registro z la loro somma.
Dalla definizione di tale linguaggio, possiamo benissimo dire che un programma è una successione di istruzioni numerate separate da un punto e virgola.
In generale per calcolare la somma di k elementi non sono sufficienti k registri.

Ad esempio consideriamo la seguente Macchina :



Posso dire che P calcola la funzione $f(x_1, x_2, \dots, x_k)$ rispetto alle variabili $x_1, x_2, \dots, x_k, x_{k+1}$ con $n \geq k+1$.
Se si inizializzano le variabili $x_1, x_2, \dots, x_k$ con i valori $t_1, t_2, \dots, t_k$, l'esecuzione del programma si arresto dopo un numero definito di passi, e dopo l'arresto nel registro x_{k+1} troveremo il valore $f(x_1, x_2, \dots, x_k)$.

Funzioni GOTO-calcolabili :
Una funzione $f(x_1, x_2, \dots, x_k)$ è **GOTO-calcolabile** se esiste un linguaggio GOTO che la calcola.
Esempio : $f(x,y) = |x - y|$ è GOTO-calcolabile.

Teorema : Una funzione è GOTO-calcolabile ⇔ è Turing-calcolabile ⇔ è ricorsiva
La dimostrazione di tale teorema è costruttiva, si parte dalla funzione GOTO e si ricava la Macchina di Turing e viceversa.

Modifica al linguaggio e loro conseguenze :
Consideriamo il linguaggio appena presentato ed apportiamo delle modifiche per vedere se i linguaggi che si ottengono sono ugualmente potenti o equivalenti.

Definizione :
Due linguaggi si dicono **equivalenti** se è possibile realizzare le istruzioni di uno con l'altro e viceversa.

- Modifica 1 : eliminiamo l'istruzione Decr (linguaggio GOTO 1)
- Modifica 2 : sostituiamo If x = 0 GOTO j con If x=y GOTO j (linguaggio GOTO 2)
- Modifica 3 : modifica 1 + modifica 2 (linguaggio GOTO 3)

Esaminiamo ognuno dei linguaggi ottenuti :

Linguaggio GOTO 1 :
è facile vedere che tale linguaggio è meno potente del GOTO, perché non esiste alcuna istruzione che emuli Decr(). Non è in alcun modo realizzabile.

Linguaggio GOTO 2 :
Tale linguaggio risulta essere equivalente al linguaggio GOTO, perché la condizione if x = y è realizzabile in linguaggio GOTO nel modo seguente :

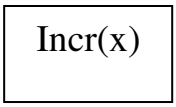
1. If x=0 GOTO 8;
2. If y=0 GOTO 6;
3. Decr(x);
4. Decr(y);
5. GOTO 1;
6. r:=x;
7. GOTO 9;
8. r:=y;
9. If r=0 GOTO 12;
10. scrivi x <> y;
11. GOTO 13 ;
12. scrivi x=y ;
13. ; /*fine istruzioni*/

Linguaggio GOTO 3 :
Risulta essere equivalente al linguaggio GOTO perché è possibile realizzare la funzione Decr() in tale linguaggio, vediamo come:

1. dec:=0; /* *dec è una variabile* */
2. if x=dec GOTO 8;
3. aux:=0;
4. Incr(aux);
5. If aux=x GOTO 8 ;
6. Incr(dec) ;
7. GOTO 4 ;
8. x :=dec ;
9. ;

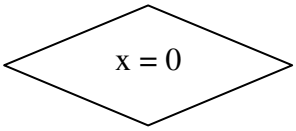
Diagrammi di flusso :
I *diagrammi di flusso* rappresentano un modo grafico per descrivere le azioni di un programma. Il metodo è di usare delle figure predefinite per ogni tipo di istruzione :

- Le istruzioni si rappresentano con un rettangolo :

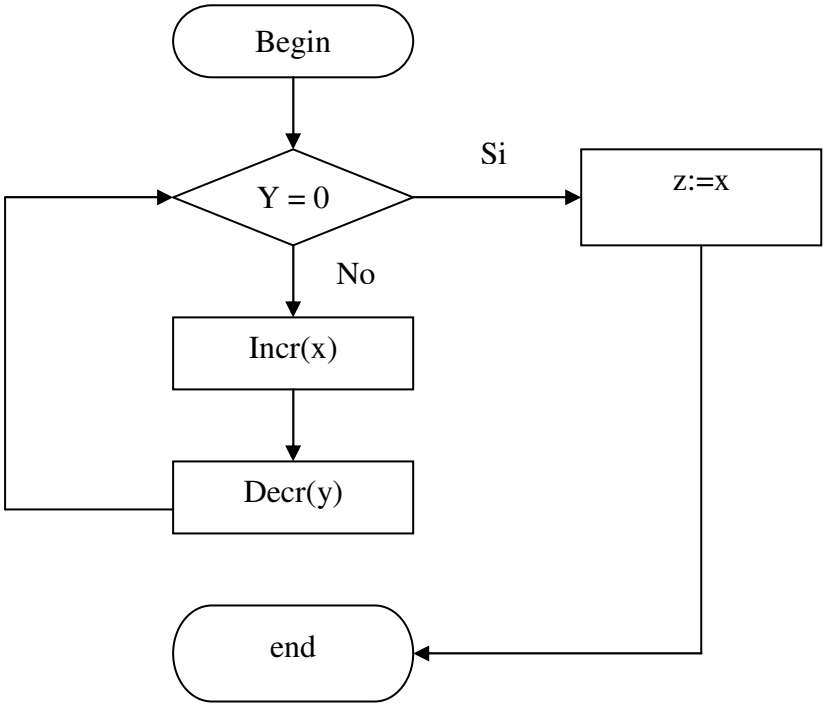


- L'inizio e la fine del programma con delle ellissi :

- Le condizioni con dei rombi :



Nell'esempio precedente il programma di somma realizzato con i diagrammi di flusso sarà :



Esercizi :

- F(x,y) = x – y
    1. If y = 0 GOTO 6
    2. Decr(x);
    3. Decr(y);
    4. If x = 0 GOTO 6
    5. GOTO 1
    6. z:=x;
- F(x,y) = |x – y|

1. If y = 0 GOTO 6
2. Decr(x);
3. Decr(y);
4. If x = 0 GOTO 7
5. GOTO 1
6. z:=x;
7. z:=y;

- F(x,y) = x * y
  1. If x = 0 GOTO 11
  2. If y = 0 GOTO 11
  3. Otherx:=x;
  4. Decr(y);
  5. If y=0 GOTO 13
  6. Aux:=otherx;
  7. Incr(x);
  8. Decr(aux) ;
  9. If aux = 0 GOTO 4
  10. GOTO 7
  11. r :=0 ;
  12. GOTO 14
  13. r:=x;
  14. <Istruzione successiva>

Lezione 6 14/10/2002

Linguaggi strutturati :

Si definiscono *linguaggi strutturati* quei linguaggi che ammettono la programmazione strutturata, ossia quel tipo di programmazione articolata in procedure, che permette di scindere problemi in sottoproblemi secondo il metodo top-down.

***Linguaggio WHILE* :**

Il Linguaggio WHILE mantiene le istruzioni semplici, ma sono modificate le istruzioni di controllo, inoltre le istruzioni non sono etichettate. Analizziamo la sintassi di tale linguaggio :

- Simboli di variabili : x, y, z, …
- Numerali : stringhe con alfabeto {0,1,2,3,4,5,6,7,8,9}
- Istruzioni di *Livello 0* :
  - Assegnazione di costanti, ad esempio x:=14 (modifica il contenuto del registro x).
  - Assegnazione di variabili, ad esempio x:=y (copia il contenuto di y in x).
  - Incr(x), incrementa di un'unità il valore del registro x.
  - Decr(x), decrementa di un'unità il valore del registro x.
- Controllo
  - If x = 0 then [blocco di istruzioni]
  - if x = 0 then [blocco di istruzioni 1] else [blocco di istruzioni 2]
  - WHILE x > 0 do [blocco di istruzioni 1], fino a quando x > 0 ,ripete il seguente blocco di istruzioni.

Classificazione dei programmi WHILE :

In questa classificazione elencheremo programmi di livello 0, Livello i.

Programmi di Livello 0 :

E' una successione di istruzioni di livello 0, separate da un punto e virgola.

Prima di definirci i programmi di livello i-esimo, definiamo le istruzioni di livello i+1 come segue :

- If x = 0 then [P], dove P è un programma di livello i
- If x = 0 then [P] else [Q], in questo caso i è il massimo tra i livelli P e Q
- While x = 0 do P

Programmi di Livello i :
E' una successione di istruzioni di livello massimo i, separate dal punto e virgola.

Esempio **:**
Il seguente programma ha un livello 2 :

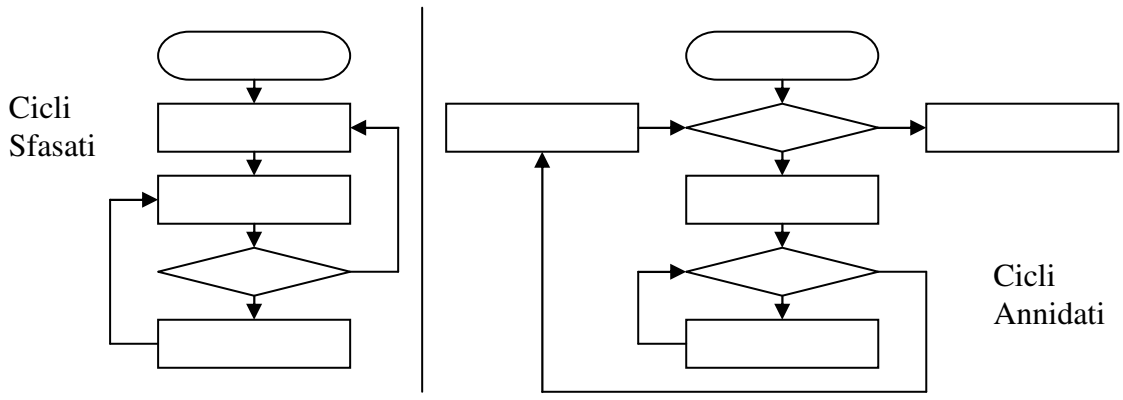
```
x:= 5;                    → livello 0
While x > 0 do            → livello 2
[
 x:= 0;                   → livello 0
 if x= 0 then             → livello 1
 [                                          } Livello 1 } Livello 2
  y:= x;                  → livello 0
  x:= 4;                  → livello 0
 ];
];
y:=14                     → livello 0
```

Esempio
Se vogliamo un programma della somma questo sarà:

```
While y>0 do
        [
                Incr(x);
                Decr(y);
        ];
z:=x;
```

Tale programma è di livello 1.

Da questo esempio risulta evidente che un programma in linguaggio WHILE ha un equivalente in linguaggio GOTO.
Il viceversa non è così ovvio,infatti, se analizziamo un programma il linguaggio GOTO, troveremo dei cicli sfasati, mentre in un programma con linguaggio WHILE troveremo dei cicli annidati dentro altri seguendo una certa gerarchia.



Teorema :
Ogni programma in linguaggio GOTO può essere tradotto in WHILE.
Definiamo con F (GOTO) le funzioni calcolabili in linguaggio GOTO e F (WHILE) le funzioni calcolabili in linguaggio WHILE.
A questo punto posso definire le funzioni F (WHILE) come le F (GOTO) e in base al teorema appena enunciato : F (WHILE) = F (GOTO) = F (TURING) = F (RICORSIVA).

Teorema :
Ogni programma può essere realizzato mediante i seguenti costrutti :
- Inizio / Terminazione
- Condizionale
- Cicli While

In accordo a questo teorema appena enunciato ne diamo un altro :

Teorema :
Ogni funzione ricorsiva si può scrivere come minimalizzazione applicata una sola volta.

$$F(x) = (\mu z)[g(x,z) = 0]$$

La conseguenza di questo teorema è che ogni programma in linguaggio di programmazione WHILE può essere scritto con un solo While.
Se apportiamo delle modifiche al linguaggio WHILE otteniamo altri tipi di linguaggi, un esempio di questi è il **Do-Times**.

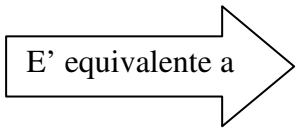
Linguaggio Do-Times :
Questo linguaggio si ottiene dal linguaggio WHILE sostituendo il costrutto :

While x > 0 [blocco di istruzioni]

Con il costrutto :

Do x Times [blocco di istruzioni]

Questo tipo di istruzione controlla il valore di x e ripete il blocco di istruzioni x volte. Così facendo abbiamo reso il linguaggio Do-Times discendente del linguaggio WHILE, e ovviamente ogni programma scritto in linguaggio **Do-Times** ha un suo corrispondente in linguaggio WHILE.

Esempio :
Do x Times [P] 

y:= x; / While y > 0 do [P; Decr(y)]

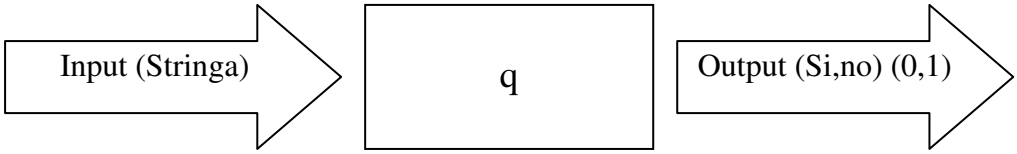
Tuttavia non sempre posso fare il contrario, perché F(Do-Times) è un sottoinsieme di F(WHILE)

Teorema:
F(Do-Times) = Funzioni Ricorsive Primitive

Lezione 7 17/10/2002

Problemi di Decisione :

Una Macchina di Turing che risolve un *problema di decisione*, è una Macchina così costituita :



L'Output può essere dato in diversi modi, in questo caso abbiamo scelto un output del tipo (0,1); quello che interessa realmente è lo stato in cui la Macchina arriva alla configurazione di arresto (o allo stato q).

Se Q è l'insieme degli stati, possiamo creare una partizione di tale insieme nel seguente modo :

$$Q = P \cup F;\ P \cap F = \varnothing$$

Dove F è *l'insieme degli stati di accettazione* in cui l'output della Macchina corrisponderà a 1 (o si).

Il Riconoscimento dei Linguaggi :

Una volta definita la Macchina di Turing che ci risolve i problemi di decisione, possiamo affrontare il problema del riconoscimento dei linguaggi.

Dato un alfabeto Σ definiamo con Σ^* l'insieme delle stringhe(o parole) sull'alfabeto Σ.

N.B. Σ^* è un insieme infinito.

Una Macchina di Turing definisce, dunque , una funzione:

$$f_t: \Sigma^* \rightarrow \{0,1\}$$

f_t risulta essere una funzione parziale perché il suo dominio è un sottoinsieme non proprio di Σ^*.

Esempio:

Sia data una Macchina di Turing che risolve problemi di decisione, si definisce:

- q_0 lo stato iniziale con $q_0 \in Q$ (insieme degli stati).
- Per convenzione l'occhio della macchina è messo sul primo simbolo a sinistra del nastro.

Se diamo in input una stringa w e la macchina non si ferma, allora diremo che:

$f_t(w)$ non è definita.

Se invece si ferma, allora avremo:

$f_t(w)=1$	se $f_t(w)$ sta in F
$f_t(w)=0$	se $f_t(w)$ non sta in F

Alla funzione $f_t:\Sigma \rightarrow \{0,1\}$ posso associare un insieme L(sottoinsieme non proprio di Σ^* così definito:

$$L=\{\ w \text{ in } \Sigma^* / f_t(w) = 1\}$$

Ad una macchina di Turing posso far calcolare f_t e quindi definire L per ogni sottoinsieme di Σ^*

Esempio:
Sia Σ l'insieme di tutte le lettere dell'alfabeto, Σ^* sarà l'insieme di tutte le parole(o stringhe) possibili che si possono creare su tale alfabeto, possiamo costruire una macchina di Turing che data una stringa in input, verifica se appartiene alla lingua italiana.

Una Macchina di Turing di questo tipo che *riconosce* un linguaggio L(o *accetta* un linguaggio L) e una macchina che mi verifica se data una stringa in input essa è corretta(appartiene a L) o meno.

Sistemi di Riscrittura:
Un insieme di riscrittura è così composto:
- Assiomi A
- Regole di riscrittura(o di produzione)

Per capire il funzionamento di tali sistemi è opportuno iniziare con un esempio, formalizzando i ragionamenti di tipo logico.

Esempio:
Si considerano le frasi:
- Uomo è mortale
- Socrate è uomo
- Socrate è mortale

La prima frase è un assioma, mentre la seconda è una regola di riscrittura, ossia sostituisci *Socrate* con *Uomo*, nella terza frase possiamo vedere l'applicazione della regola all'assioma(Teorema).
In generale un sistema di riscrittura definito da

- A = insieme degli assiomi
- R = insieme finito di regole di riscrittura
- (A,R) = Sistema di riscrittura

Posso dunque definire un insieme di stringhe, e quindi un sistema di riscrittura mi definisce un linguaggio L (A,R) sottoinsieme non proprio di Σ^* che chiamiamo *Linguaggio generato dal sistema di riscrittura (A,R).*

Teorema:
L(sottoinsieme non proprio di Σ^*) è riconosciuto da una macchina di Turing ⇔ L è generato dal sistema di riscrittura (a,R).

In genere metodi di riscrittura hanno delle regole unidirezionali, ossia, dato un simbolo *u* posso sostituire con *v*, ma non posso fare il viceversa. Esistono però sistemi di riscrittura simmetrici o bidirezionali, in cui data una lettera *u* posso sostituirla con *v* e viceversa.

Esempio:
Il calcolo algebrico è un sistema di riscrittura bidirezionale, infatti:

$$a^2-b^2= (a + b) (a - b) \text{ e viceversa}$$

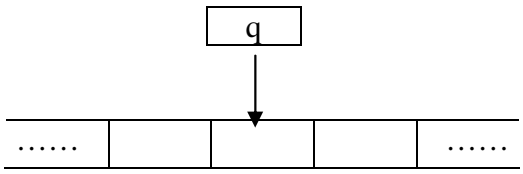
Esempio:
Sia dato l'alfabeto Σ={a,b,c} e siano date le seguenti regole di riscrittura:
1. b = acc;
2. ca= acccc;
3. aa=ε;

4. bb=ε;
5. cccc=ε;

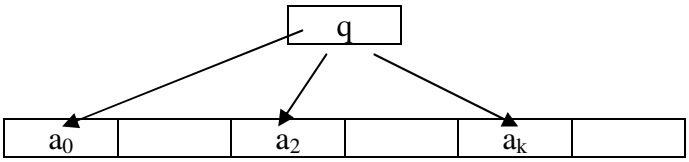
ove ε è la parola vuota.
Trovare un algoritmo che date due stringhe mi dice se queste sono uguali o meno.
Svolgimento:
La strategia è quella di eliminare la lettera b dalla stringa data, in maniera tale da ricondurla ad una stringa contenente soltanto le lettere a e c.
A questo punto è facilmente facile vedere che, applicando le altre regole di riscrittura, ogni stringa può essere ricondotta ad una delle seguenti forme: {ε,a,c,ac,ca,cc,acc} (sottoinsieme di Σ^*).
Proviamo a confrontare due stringhe:
S1= a b c b a a b a
S2= b c a a a c b
Applichiamo le regole di riscrittura:
S1= a b c b a a b c (regola 1) → a a c c c a c c a a a c c a
S2= b c a a a c b (regola 1) → a c c c a a a c a c c
S1= a a c c c a c c a a a c c a (regola 3) → c c c a c c a c c a
S2= a c c c a a a c a c c (regola 3) → a c c c a c a c c
S1= c c c a c c a c c a (regola 2) → c c a c c c c c a c c c c c a c c c c c
S2= c c a c c c c c a c c c c c a c c c c(regola 5) → c c a c a c a
S1= c c a c c c c c a c c c c c a c c c c c(regola 2) →c a c c c c c c c c a c c c c c c c c c a c c c c c c c c c
S2= c c a c a c a(regola 2) → c a c c c c a c c c c a c c c c
S1= c a c c c c c c c a c c c c c c c a c c c c c c c c(regola 5) → c a a a
S2= c a c c c c a c c c c a c c c c (regola 5) →c a a a
S1= c a a a (Regola 3) → ca (forma nota)
S2= c a a a (Regola 3) → ca (forma nota)
Quindi le due stringhe sono uguali.

Lezione 8 21/10/2002

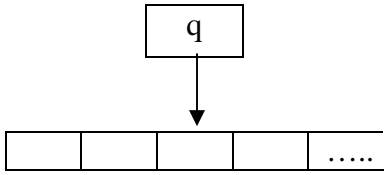
Nella definizione di Macchina di Turing si è definito un nastro praticamente *infinito* sia a destra ce a sinistra.



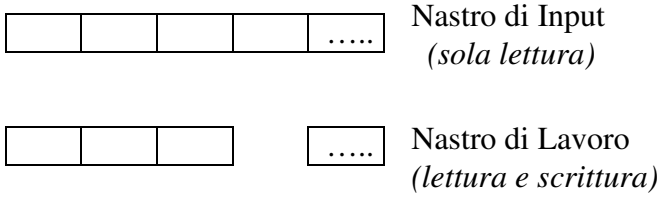
Alcuni scettici sulla macchina di Turing originaria, hanno deciso di apportare delle modifiche a tale macchina ,costruendo così nuovi modelli:



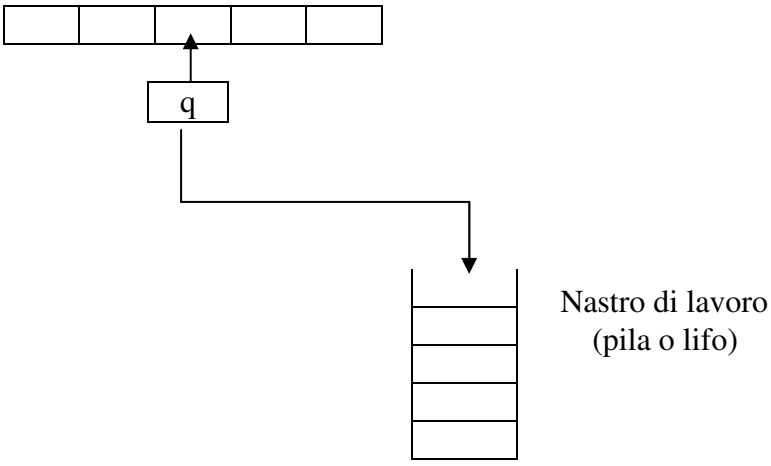
Questo particolare modello è la raffigurazione di una macchina di Turing con più teste.
Possiamo apportare alcune limitazioni alla macchina, infatti, potremmo supporre che il nastro sia infinito solo a destra, o meglio, sappiamo dove inizia, ma non dove finisce.



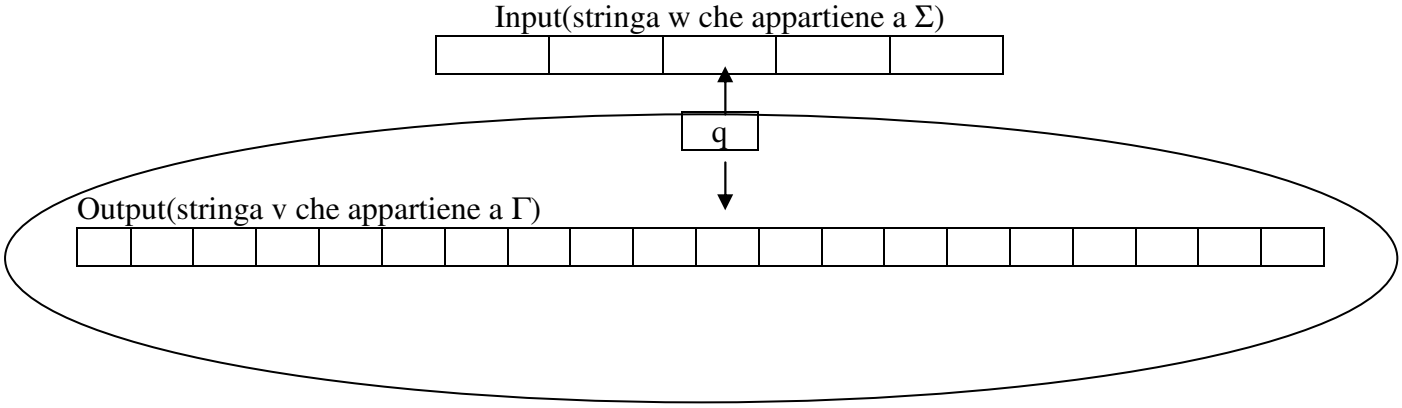
Un'altra limitazione che possiamo apportare è, mettere un doppio nastro con le seguenti caratteristiche:



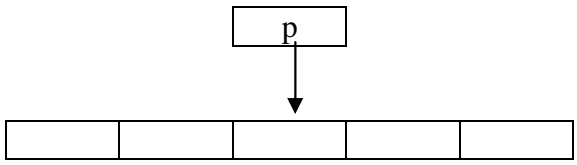
Dove il *Nastro di Input* è di sola lettura e di lunghezza *fissata*, mentre il *nastro di lavoro* è di lettura e scrittura ed ha una lunghezza fissata in funzione del nastro di input.
La domanda che sorge è: Applicando queste modifiche, la potenza di calcolo della macchina di Turing viene limitata?
Le risorse sono legate al tipo di modello che stiamo trattando e in particolare alla lunghezza del nastro di lavoro.
Macchine di Turing come quella vista sopra vengono chiamate LBA(Linear Bounded Automata), e sono classificate come riconoscitori di linguaggi.
Supponiamo adesso di avere un nastro di lavoro dove posso cancellare solo l'ultima lettera che ho scritto, o l'ultima diversa dallo spazio vuoto, e posso leggere solo l'ultima lettera scritta, e posso scrivere solo sull'ultima parola.
Apportando queste modifiche non abbiamo fatto altro che creare una *pila "lifo"* ma vediamo come ci appare questa macchina:



Questo tipo di macchina è detta PDA (push-down-Automata)
Continuando nella discesa astrattiva delle macchine, apportiamo altre modifiche, supponiamo di avere una macchina con due nastri del tipo:



La modifica è quella di considerare la lunghezza della stringa di output indipendente da quella di input w. Questa indipendenza mi permette di inglobare la parte cerchiata in una macchina del tipo:



Le possibili configurazione della parte cerchiata sono (q,v) e quindi p=(q,v). questo tipo di macchina si chiama 2FSA (Finite-State-Automata).
Concludendo l'ultimo tipo, che chiameremo FSA, sarà una macchina dello stesso tipo , con la differenza che la testa si può muovere solo a destra.
Ci troviamo quindi con macchine che hanno un livello di astrazione più basso, e quindi alcuni problemi che si erano considerati in decidibili, ora possono diventare decidibili.
Definiamo quindi alcuni problemi:

Problemi di Decisione:

- **Membership Problem (MP)**
  Supponiamo di avere in input un automa A ed una stringa w, e definiamo L(A) il linguaggio riconosciuto da A ossia l'insieme delle stringhe che portano A ad uno stato di accettazione.
  Esiste una macchina che ci da un output di questo tipo ?

  OUTPUT { w appartiene ad L(A)
  OUTPUT { w non appartiene ad L(A)

- **Emptyness Problem (EP)**:
  Supponiamo di avere in input un Automa .
  Esiste una macchina che ci da un output del tipo:

  OUTPUT { $L(A)=\emptyset$
  OUTPUT { $L(A)\neq\emptyset$

  o meglio esiste almeno una stringa w che è riconosciuta dall'automa A?

- **Fitness Problem (FP)**:
  Si suppone di avere in input un automa A.
  Esiste una macchina che ci da un output di questo tipo?

  OUTPUT { L(A) Finito
  OUTPUT { L(A) Infinito

- **Equivalence Problem (EQP)**:
  Dati in input due automi A_1 e A_2.
  Esiste una macchina che ci da un output di questo tipo?

  OUTPUT { $L(A_1) = L(A_2)$
  OUTPUT { $L(A_1) \neq L(A_2)$

O meglio esiste una macchina di Turing che ci dice se i linguaggi definiti da due automi sono uguali?

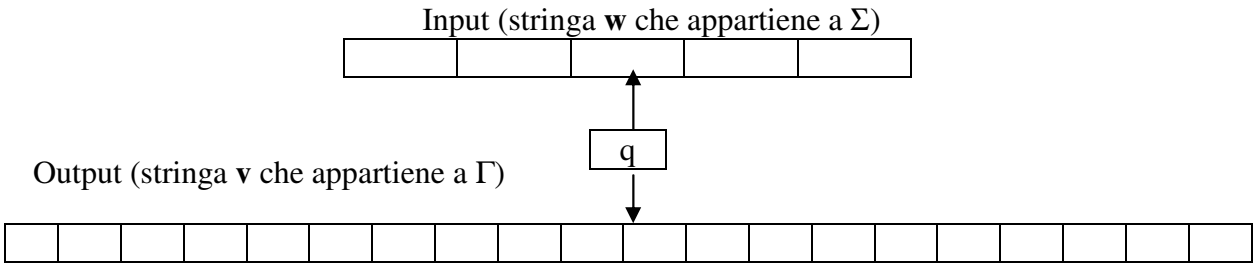
Teorema:
Per le macchine di Turing i problemi MP, EP, FP; EQP sono indecidibili.

Dopo aver fatto una panoramica delle possibili macchine che possiamo costruire facendo opportune modifiche, rilasciamo una tabella che ci rifornisce lo schema visivo di come si pongono gli automi costruiti in precedenza rispetto ai problemi di decisione:

	MP	**EP**	**FP**	**EQP**
MT	**I**	**I**	**I**	**I**
LBA	**D**	**I**	**I**	**I**
PDA	**D**	**D**	**D**	**I**
2FSA	**D**	**D**	**D**	**D**
FSA	**D**	**D**	**D**	**D**

Sulle righe abbiamo posto i vari tipi di automi, mentre sulle colonne i vari problemi di decisione, ovviamente D sta per *problema decidibile*, mentre I sta per *problema indecidibile*.

Teorema :
Per gli automi LBA, MP è decidibile.
Dimostrazione : Se diamo una stringa **w** in ingresso, fissiamo una quantità di nastro di lavoro; infatti nella costruzione degli LBA abbiamo visto che questo è lungo in proporzione alla stringa in ingresso, e quindi al nastro in input, il problema della fermata è dunque decidibile :



Data una stringa in ingresso **w**, la lunghezza della stringa di output **v** sarà :

$$v = k * |w| = \text{lunghezza del nastro}$$

Una possibile configurazione sarà data da : (q,i,j,v)
Dove i evidenzia la posizione della testa nel nastro di input, j la posizione nel nastro di lavoro, e v la lunghezza della stringa. Quindi :

$$q \text{ appartiene a } Q$$
$$|Q| = N \text{ (cardinalità di Q)}$$
$$i = 1,2,3,\ldots,|w|$$
$$j = 1,2,3,\ldots,k*|w|$$
$$\text{Card}(\Gamma) = M$$

Deduciamo che le possibili stringhe dell'alfabeto Γ di lunghezza $v = k * |w|$, sono in numero di $M^{k * |w|}$, e quindi il numero delle configurazioni sarà :

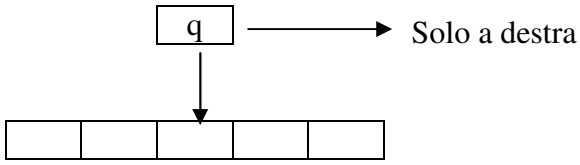
$$N * |w| * k * |w| * M^{k * |w|} = \text{massimo numero di passi}$$

Se la Macchina non si ferma entro tale intervallo (o numero di passi), la stringa non è riconosciuta e l'automa non si ferma mai.
Infatti, come sappiamo, abbiamo un numero di configurazioni fissato, questo significa che se non trovo la stringa al primo passaggio, non la troverò mai, perché non faccio altro che passare sempre sulle stesse configurazioni, e quindi sulle stesse stringhe.

Lezione 9 24/10/2002

Definiamo meglio gli automi a stati finiti, o meglio conosciuti come **FSA**, e vediamo come questo tipo di Macchine lavorano.

Automi a stati finiti (FSA) :
In precedenza avevamo definito l' FSA come un automa formato da un nastro di sola lettura ed una testina che si può spostare solo a destra :



Volendo immaginare come funziona un FSA possiamo pensare ad un computer con una tastiera: quando digitiamo le lettere lo facciamo in maniera sequenziale, e quando avremo finito, sul monitor avremo in output una certa stringa, che determina un certo stato del computer.

Ebbene, allo stesso modo in un FSA abbiamo una certa stringa, e quando finisco la lettura la Macchina sarà in un certo stato. I componenti necessari in un FSA sono :

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

Definiamo i simboli come :

- Σ = Alfabeto;
- Q = Insieme degli stati;
- q_0 = Stato iniziale della Macchina;
- F = Insieme degli stati di accettazione;
- δ = Funzione di transazione.

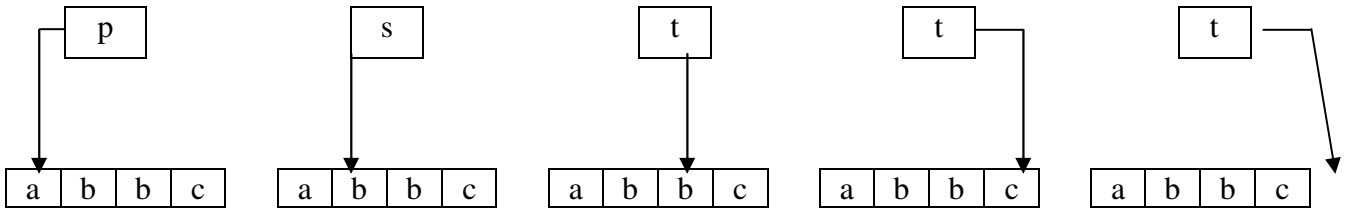
Definiamo meglio δ come segue : $\delta : Q \times \Sigma \rightarrow Q$.
Ossia δ è una funzione definita nel prodotto cartesiano formato dall'insieme degli stati e dell'alfabeto, con valori in Q.
Quindi la scrittura $\delta(q, a) = p$, significa che la Macchina che si trova in uno stato q, legge la lettera a del nastro, e passa allo stato p. Per meglio comprendere, riportiamo un esempio di FSA.

Esempio :
Sia dato un automa **A** di tipo FSA con i seguenti parametri :

- $\Sigma = \{a,b,c\}$;
- $Q = \{p,q,s,t\}$;
- $q_0 = p$;
- $F = \{s\}$;
- δ <dato dalla seguente tabella> :

δ	a	b	c
p	s	q	p
q	s	t	t
s	s	t	p
t	t	t	t

Se diamo in input la stringa **a b b c** :



Sappiamo che, dato un automa A, posso definire :

- L(A) è l'insieme delle stringhe accettate (riconosciute) dall'automa A;

L(A) è il linguaggio riconosciuto dall'automa **A**.

A questo punto definiamo 2 problemi :

- Problema dell'analisi : dato un automa A, possiamo definire un linguaggio L(A) riconosciuto dall'automa?
- Problema della sintesi : dato un linguaggio L(A), possiamo definire un automa A che lo riconosce?

Definizione : siano date delle stringhe u,v,w, contenute in Σ^*, posso definire **concatenazione di due stringhe,** come segue :

$$\text{se } u = a\ b\ b\ a\ b,\ v = b\ a\ b \Rightarrow uv = a\ b\ b\ a\ b\ b\ a\ b$$

Tale operazione è associativa ma non commutativa; infatti, nella maggior parte dei casi, $uv \neq vu$.
A questo punto diamo una descrizione più formale di Σ^* in cui :

$$\Sigma^* = \left\{ \begin{array}{l} \text{Vale l'associativa} \\ \text{non vale la commutativa} \\ \text{Esiste l'elemento neutro, denotato con } \varepsilon, \text{ che è la parola vuota } (v\varepsilon = \varepsilon v = v) \end{array} \right\} \text{ Monoide}$$

Se **u** è una stringa, allora la concatenazione **uu** è uguale a $\mathbf{u^2}$, quindi si ha :

$$\underbrace{uuuuu\ldots u}_{\text{k volte}} = u^k$$

Da notare che se $u = w^p$ e $v = w^q$, allora $uv = w^{p+q} = vu$

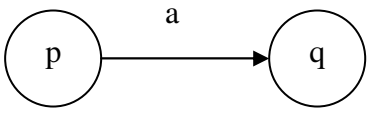
Esempio :
w = a b; p = 2; q = 3; u = a b a b; v = a b a b a b; uv = a b a b a b a b a b = vu;

Un'altra simbologia usata è

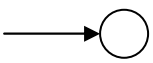
- $|v|$ = {lunghezza della stringa v};
- $|v|_a$ = {numero di volte che compare a nella stringa}.

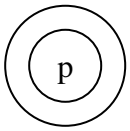
Nell'esempio precedente si ha dunque che $|uv| = |u| + |v|$.

La rappresentazione di un FSA con il grafo degli stati :
Un FSA si può rappresentare con un grafo i cui vertici rappresentano gli stati (elementi di Q), mentre gli archi rappresentano le transizioni :



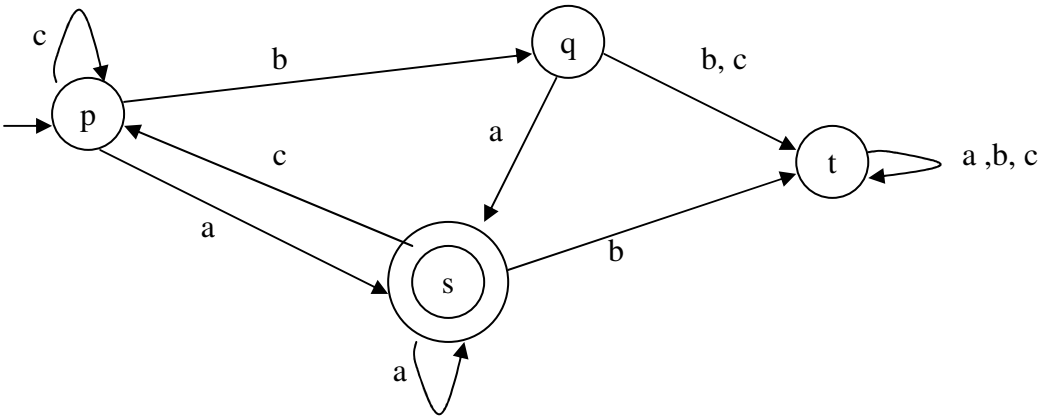
con $a \in \Sigma$, questo pezzo di grafo appena descritto si può leggere come $\delta(p,a) = q$.
E' utile scrivere una simbologia più completa avendo quindi :

 stato iniziale;



stato di accettazione

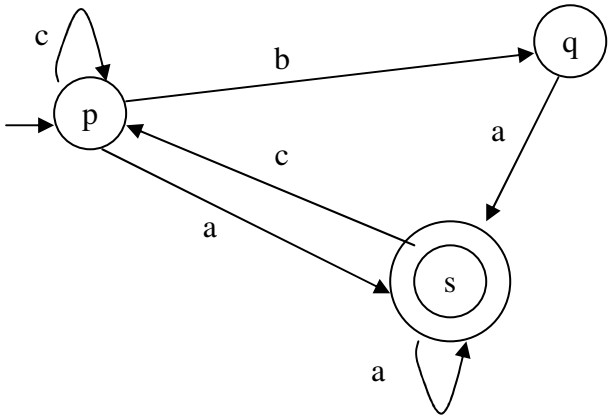
Esempio :
Consideriamo l'automa precedente e costruiamolo con la rappresentazione a grafo :



N.B. : Da ogni nodo escono tanti archi quanti sono gli elementi di Σ e tutti con etichette distinte, anche se, nella nostra notazione, per non disegnare sempre archi diversi abbiamo disegnato con diverse lettere uno stesso arco. Questa proprietà si chiama **determinismo**, ossia ad ogni simbolo corrisponde uno ed un solo arco. Infine, ad ogni stringa mi corrisponde un cammino, e quindi le stringhe che appartengono ad L(A), sono quelle che dopo il cammino vanno a finire nello stato di accettazione (S nel caso precedente. Se consideriamo la stringa c b a a a, questa appartiene ad L(A))
Sul grafo appena descritto possiamo fare alcune considerazioni :

- Nessuna a è seguita da b;
- Ogni b è seguita da a;
- L'ultimo simbolo è sempre a, quindi le stringhe accettate sono quelle che terminano con a.

Da ciò possiamo ridurre il grafo precedente ottenendo così :



Lezione 10 25/10/2002

Abbiamo definito gli elementi necessari per definire un automa, dove :

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

Altresì abbiamo definito la costruzione di un automa attraverso il grafo degli stati, dove la funzione di transizione è così definita :

$$\delta : Q \times \Sigma \rightarrow Q$$

e dove :

$$\delta(q,a) = \text{un certo stato}$$

Esistono alcuni automi in cui la funzione di transizione non è definita per alcune coppie di valori.

Automi incompleti :
Sono automi per cui la funzione di transizione δ, per alcune coppie non è specificata.

Il grafo si può costruire comunque, e dove la coppia non è definita non metto l'arco.
Bisogna dire che è conveniente lavorare con grafi incompleti perché tale handicap non influisce sull'output. Se voglio completare un automa incompleto, basta aggiungere uno stato t, e fare puntare tutti gli archi non definiti a tale stato. Così facendo otteniamo quello che è denominato *stato di pozzo*, e in più otteniamo un nuovo insieme di stati :

$$Q^1 = Q \cup \{t\}$$

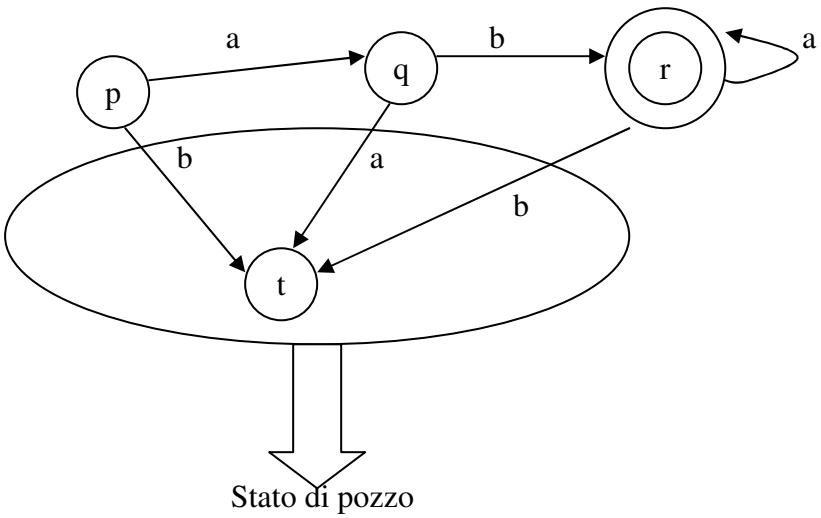
Esempio :
Supponiamo di avere un automa A dove :

$$\Sigma = \{a, b\}; \qquad Q = \{p, q, r\}$$

e dove la funzione di transizione :

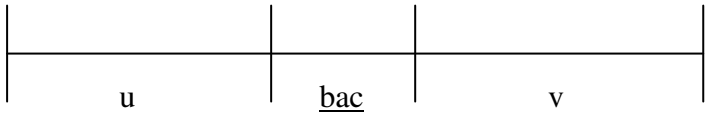
δ	a	b
p	q	
q		r
r	r	

Costruiamo di seguito l'automa incompleto :



Definizioni di alcuni linguaggi :
Definiamo alcuni linguaggi semplici :

- L_1 : Tutte le parole sull'alfabeto $\Sigma = \{a, b, c\}$ che contengono il blocco fattore "bac".
  Quindi la stringa sarà : w = u <u>bac</u> v



  cioè una prima parte u, poi il fattore <u>bac</u>, e una seconda parte v.

- L_2 : Tutte le stringhe, o parole, su $\Sigma = \{a, b, c\}$ che contengono il carattere **a** un numero di volte multiplo di 3.

- L_3 : Tutte le stringhe su $\Sigma = \{0, 1\}$ che corrispondono alla rappresentazione binaria di un multiplo di 3.

- L_4 : Tutte le stringhe su $\Sigma = \{0, 1, 2\}$ che corrispondono alla rappresentazione in base 3 di numeri pari.

- L_5 : Tutte le stringhe su $\Sigma = \{0, 1\}$ che corrispondono alla rappresentazione binaria di numeri primi.

- L_6 : Tutte le stringhe su $\Sigma = \{0, 1, +, =\}$ che hanno un formato di questo tipo :

$$1011 + 11000 = 11$$

- L_7 : Come L_6, in più la somma deve essere corretta.

- L_8 : Tutte le stringhe su $\Sigma = \{$triple su $\{0, 1\}\}$, ossia :

$$\Sigma = \left\{ \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline A & B & C & D & E & F & G & H \end{array} \right\} \{A, B, C, D, E, F, G, H\}$$

che identificano una somma corretta, ad esempio :

$$\begin{array}{ccccccc} 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ B & E & G & F & B & G & F \end{array}$$

Quindi la stringa **BEGFBGF** appartiene L_8.

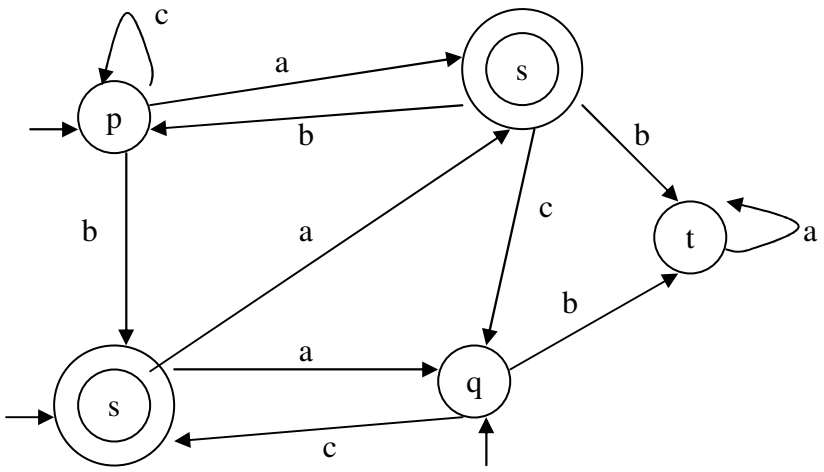
- L_9 : Linguaggio delle parentesi dove $\Sigma = \{(,)\} = \{a, b\}$
  Dove se ho la stringa : **(( ) ( )) ( ) = aababbab** appartiene ad L_9;

mentre la stringa : **( )) (( ) = abbaab** non appartiene ad L_9.

- L_{10} : Linguaggio delle parole palindrome, con $\Sigma = \{a, b, c\}$, ad esempio la stringa :
  - **abccbbccba** appartiene a L_{10}, mentre la stringa :
  - **abccb** non appartiene a L_{10}.

Naturalmente tutti questi linguaggi non è detto che siano realizzabili da automi a stati finiti, e altresì tutti questi sono problemi di sintesi.

Se vogliamo definire un insieme di stringhe qualsiasi basta prendere un grafo arbitrario dove ad esempio abbiamo tre vertici iniziali e due finali :



Il Grafico **G** rappresenta un *automa non deterministico*.

Definizione :
Un automa non deterministico è una quintupla così composta :

$$A = (\Sigma, Q, I, F, \tau)$$

dove :
- Σ = Alfabeto;
- Q = Insieme degli stati;
- I = (sottoinsieme non proprio di Q) insieme degli stati iniziali;
- F = (sottoinsieme non proprio di Q) insieme degli stati di accettazione;
- τ = funzione di transizione così definita :

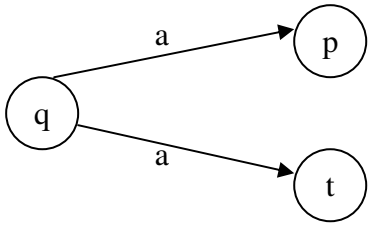
$$\tau : Q \times \Sigma \rightarrow P(Q)$$

dove P(Q) è l'insieme delle parti (possibili sottoinsiemi) di Q.
Per semplificare possiamo altresì definire τ come segue :

$$\tau = \{ (q, a, p) \}$$

ossia un insieme di terne ordinate in cui il primo elemento identifica il nodo di partenza, il secondo è l'etichetta dell'arco uscente dal nodo di partenza e il terzo identifica il nodo di arrivo.
A questo punto possiamo fare delle considerazioni :

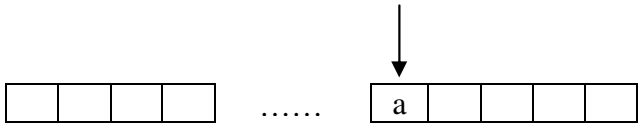
1. Come si evince dalla definizione due elementi del tipo : **(q, a, p) e (q, a, t)**; appartengono entrambi a τ e definiscono questa situazione :



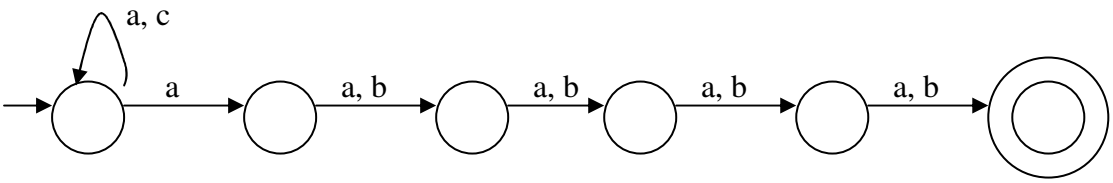
ossia un nodo **q** può avere più archi uscenti, con la stessa etichetta, che definiscono stati diversi.

2. Questo tipo di formalizzazione è applicabile anche ad un qualsiasi **DFA**, a patto che si verifichino le seguenti condizioni :
   - Contiene un solo elemento (il nostro q_0 o il nostro nodo iniziale)
   - Due o più elementi del tipo (q, a, ?) non appartengono contemporaneamente a τ.

Esempio :
Definiamo un nuovo linguaggio L che è l'insieme delle stringhe definite sull'alfabeto $\Sigma = \{a, b\}$ tale che la quintultima lettera sia **a** :



Quando io inizio a leggere non so quanto è lunga una stringa, ma proviamo a costruire un NFA che mi riconosce tale linguaggio :



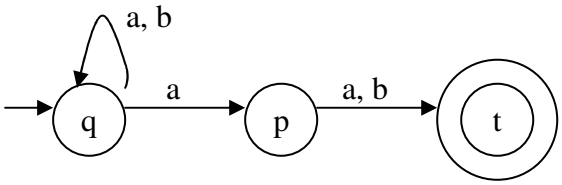
Questo automa legge la stringa e se la quintultima lettera è una **a** esce dal nodo iniziale e arriva allo stato di accettazione.

A questo punto definiamo con :
- **L(NFA)** : Tutti i linguaggi che posso riconoscere con gli **NFA**;
- **L(DFA)** : Tutti i linguaggi che posso riconoscere con i **DFA**.

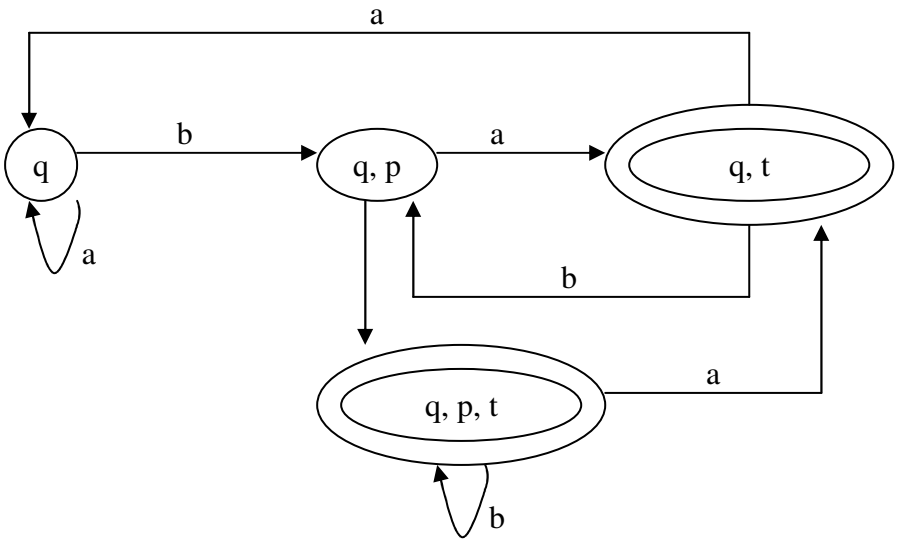
Lezione 11 31/10/2002

E' sempre possibile ricavare da un **NFA** un **DFA**, e il viceversa attraverso un procedimento che viene chiamata *Subset-construction*, vediamo tale procedimento che è costruttivo e poi diamo la definizione formale di tale algoritmo:

Esempio :
Dato un **NFA** del tipo :



ci ricaviamo il seguente **DFA**:



Vediamo adesso passo per passo come abbiamo fatto tale costruzione :

- Partiamo dallo stato iniziale del nostro **NFA q** e vediamo che leggendo **a** questo resta nello stesso stato, così aggiungiamo una freccia di ritorno e andiamo avanti.

- Sempre nello stato **q** se leggiamo **b** andiamo a finire sia nello stato **q** che **p**, così raggruppiamo questi ultimi e creiamo lo stato (q, p) e facciamo puntare la lettera **b** uscente da **q** a quest'ultimo.

- Iteriamo tale procedimento per lo stato (q, p) dove trattiamo **q** e **p** come due stati a se stanti. Infatti leggendo **a** dallo stato **q** andiamo a finire in **q**, mentre se leggiamo **a** dallo stato **q** andiamo a finire in **t**. Raggruppiamo gli stati ottenuti e creiamo lo stato (q, t), che sarà di accettazione perché **t** lo è, a cui facciamo puntare la lettera **a** uscente a (q, p).

- Leggendo la lettera **b** da **q** ci portiamo sia allo stato **q** che allo stato **p**, mentre leggendo **b** da **p** ci portiamo allo stato **t**. Raggruppiamo sempre creando lo stato (q, p, t) a cui facciamo puntare l'arco **b** uscente dallo stato (q, p).

- Passiamo allo stato (q, t), dove leggendo **a** da **q** ci portiamo allo stesso stato **q**, mentre leggendo **a** da **t** vediamo che l'arco non è segnato. In questo caso l'arco **a** uscente da (q, t) punterà solamente allo stato **q** del nostro **DFA**.

- Leggendo **b** dallo stato **q** ci portiamo sia in **q** che in **p**, mentre leggendo **b** dallo stato **t** vediamo che l'arco non è segnato. In questo caso come precedentemente l'arco **b** uscente da (q, t) punterà semplicemente allo stato (q, p).

- Non ci resta che controllare solo (q, p, t), dove leggendo **a** dallo stato **q** ci portiamo allo stato **q** stesso, leggendo **a** da **p** andiamo allo stato **y**, e da **t** l'arco non è segnato. In questo caso (q, p, t) punterà semplicemente a (q, t).

- Al passo successivo leggendo **b** da **q** ci portiamo sia in **q** che in **p**, leggendo **b** da **p** ci portiamo in **t**, mentre in **t** l'arco **b** non è segnato. Quindi l'arco **b** uscente dallo stato (q, p, t) punterà a se stesso.

Dopo avere fatto la costruzione di tale algoritmo è utile dare anche una descrizione più formale anche attraverso il teorema che segue :

Teorema :
L(DFA) = L(NFA)
Dimostrazione:
La dimostrazione di tale teorema è costruttiva e si identifica con il nome di **Subset Construction**.
Sia dato un **NFA** :

$$A = (\Sigma, Q, I, F, \tau)$$

con I, F sottoinsiemi non propri di Q e τ sottoinsieme non proprio di Q x Σ x Q.
Esiste un **DFA** equivalente così fatto :

$$A_D = (\Sigma, Q_D, q_0, F_D, \delta)$$

ove :
Q_D = { $[q_{i1}, q_{i2},\dots , q_{ij}]$ / $\{q_{i1}, q_{i2},\dots , q_{ij}\}$ appartiene a $\vartheta(Q)$ (=insieme delle parti di Q) }.

Ossia l'insieme dei *superstati* i cui elementi formano sottoinsiemi di Q

$q_0 = [q_{i1}, q_{i2},\dots , q_{ij}]$ tale che $\{q_{i1}, q_{i2},\dots , q_{ij}\}$ = I appartenente a $\vartheta(Q)$

F_D = { $[q_{i1}, q_{i2},\dots , q_{ij}]$ / $\{q_{i1}, q_{i2},\dots , q_{ij}\}$ appartiene all'insieme $\vartheta(Q)$ e $\{q_{i1}, q_{i2},\dots , q_{ij}\} \cap F \neq \varnothing$ }

ossia l'insieme dei *superstati* i cui elementi formano sottoinsiemi di **Q** che contengono almeno uno stato di accettazione.
Per definire la funzione di transizione utilizziamo il formalismo di τ :

$$\tau : Q \times \Sigma \rightarrow \vartheta(Q)$$

dove :

$$\delta([q_{i1}, q_{i2},\dots , q_{ij}], a) = [q_{z1}, q_{z2},\dots , q_{zj}]$$

e dove :

$$\{q_{z1}, q_{z2},\dots , q_{zj}\} = \tau(q_{z1}, a) \cup \tau(q_{z2}, a) \cup \dots \cup \tau(q_{zt}, a)$$

Per dimostrare che i due automi riconoscono lo stesso linguaggio bisogna dimostrare che ad ogni computazione del **NFA A** ne corrisponde una del **DFA** A_D, cioè che :

$$\delta^*(q_0, w) = [q_{i1}, q_{i2}, \dots, q_{ij}] \text{ appartenente a } F_D \Leftrightarrow \bigcup_{q \text{ in } I} \tau^*(q, w) \cap F \neq \varnothing$$

Ma questo è vero per costruzione come visto prima.

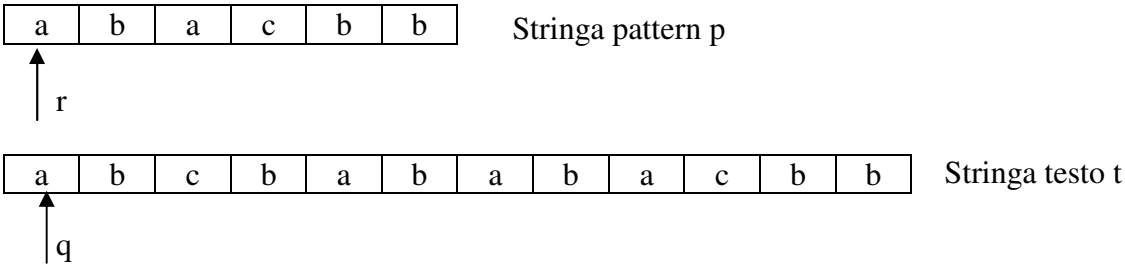
Lezione 12 07/11/2002

Algoritmo di String-Matching :

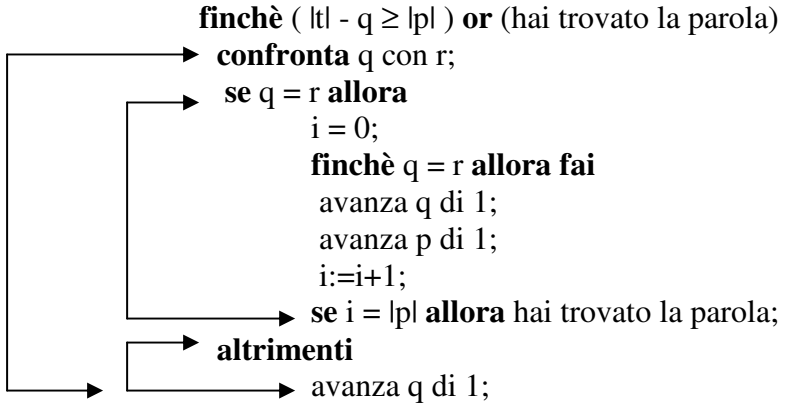
L'algoritmo di String-Matching, consiste nel verificare se una stringa compare in un testo. Supponiamo di avere una stringa, che chiamo *pattern*, e un testo così composti :

$$p = p_1 * p_2 * p_3 * \dots * p_k \qquad e \qquad t = t_1 * t_2 * t_3 * \dots * t_n$$

e graficamente rappresentate in questo modo :



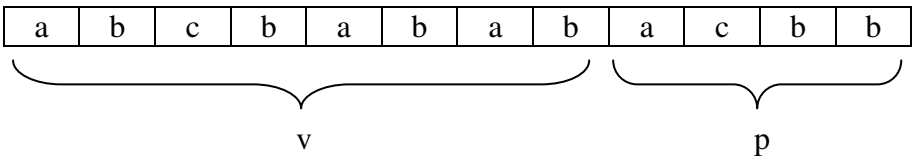
Il procedimento è molto intuitivo, ma per rendere la spiegazione più facile scriviamolo con uno pseudolinguaggio che ricorda un programma :



Innanzitutto, il confronto avviene fino a quando, o trovo la parola, o **q** punta a un elemento del testo che si trova in una posizione che è minore della lunghezza del pattern. Successivamente confronto i due caratteri, se sono uguali faccio controllare i successivi, altrimenti avanzo semplicemente **q** di una posizione. Ovviamente se **i** è uguale alla lunghezza del pattern significa che mi sono confrontato tutti i caratteri, e che quindi sono uguali.
A questo punto non mi resta che fermarmi perché la mia macchina ha riconosciuto la parola.

Preprocessing :
La caratteristica essenziale di questo algoritmo è di fare una preelaborazione o sul pattern o sul testo. Avvio la preprocessing sul pattern quando cercare una parola che compare su diversi testi, mentre il viceversa quando devo trovare il numero di occorrenze di una parola su di un testo.

Preprocessing sul Pattern :
Come detto prima, quando uso questa strategia voglio sapere se una parola si trova su uno o più testi. Il caso diventa più semplice quando devo dire se durante una ricerca trovo o non trovo la parola cercata, e appena la trovo mi fermo.
Questo tipo di strategia mi definisce un insieme di stringhe che contiene tutte le stringhe che hanno come suffisso **p**, quindi :

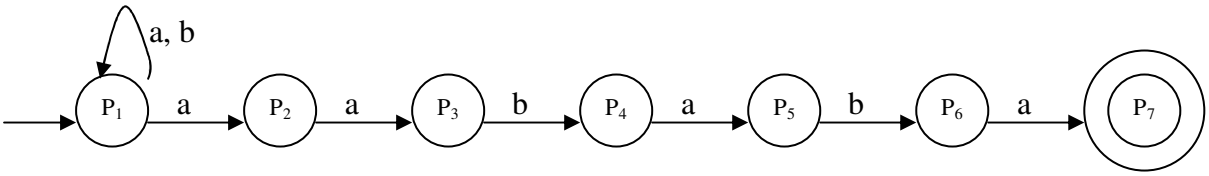


Stringa w = vp

e ovviamente :

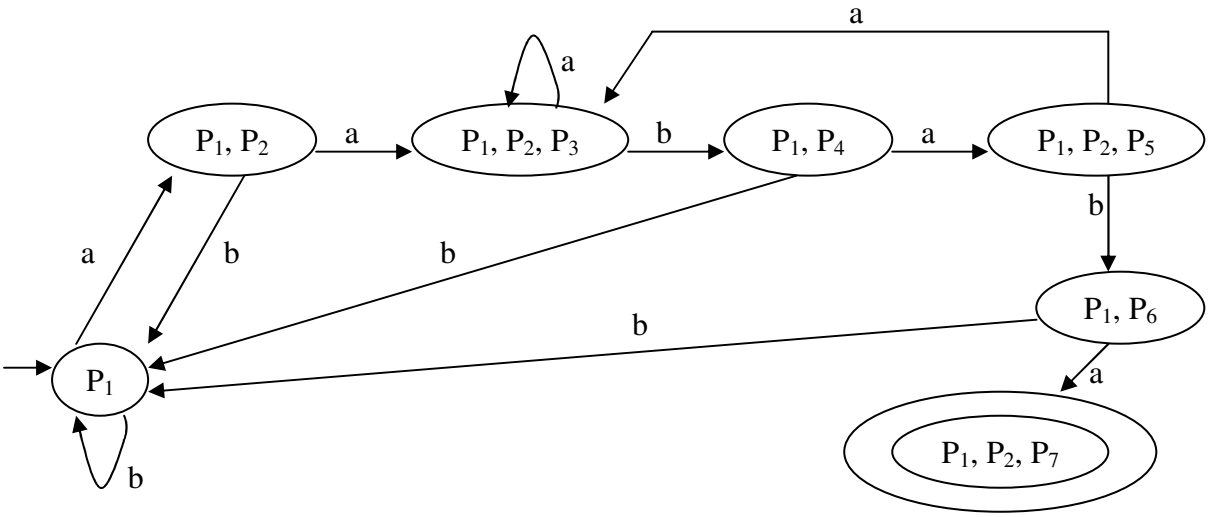
$$L_p = \{vp \,/\, v \text{ appartiene a } \Sigma^*\}$$

Esempio :
Supponiamo di avere un alfabeto $\Sigma = \{a, b\}$ e una parola P = a a b a b a, e costruiamo un **NFA** che conosce tutte le stringhe che finiscono con p :



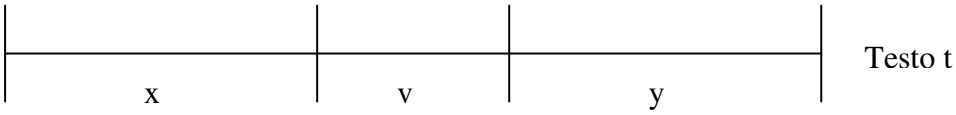
Teorema :
Quanto determinizzo problemi di questa forma il numero di stati non aumenta.

Ma costruiamoci il nostro **DFA** :



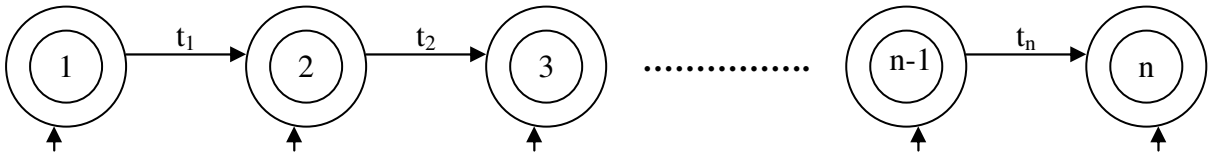
Una volta arrivato allo stato di accettazione, non marco più gli altri archi perché sono inutili in quanto mi farebbero ritornare indietro.
Il tempo totale dell'algoritmo è uguale al tempo per costruire l'automa più la lunghezza **n** della stringa, osservando che il tempo per costruire l'automa è proporzionale a **n**. Ricordando l'algoritmo precedente, notiamo che qui è come se avessimo memoria; infatti l'automa non ritorna più indietro a controllare la lettera successiva dopo che aveva trovato due lettere uguali e aveva iniziato a fare il confronto tra il pattern e il testo.

Preprocessing sul testo :
Ricordiamo che usiamo questa strategia quando vogliamo cercare il numero di occorrenze di una parola su di un testo. Iniziamo considerando F(t) insieme di fattori o "blocchi" di **t**.
Data una stringa **v**, quindi, questa è un blocco o un fattore di **t** se t = xvy, ossia :



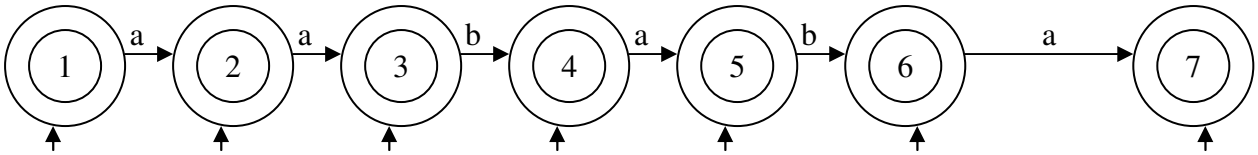
quindi F(t) = l'insieme di **v** appartenenti a **t**.

Costruiamo il nostro **NFA** ricordando che : $t = t = t_1 * t_2 * t_3 * \ldots * t_n$

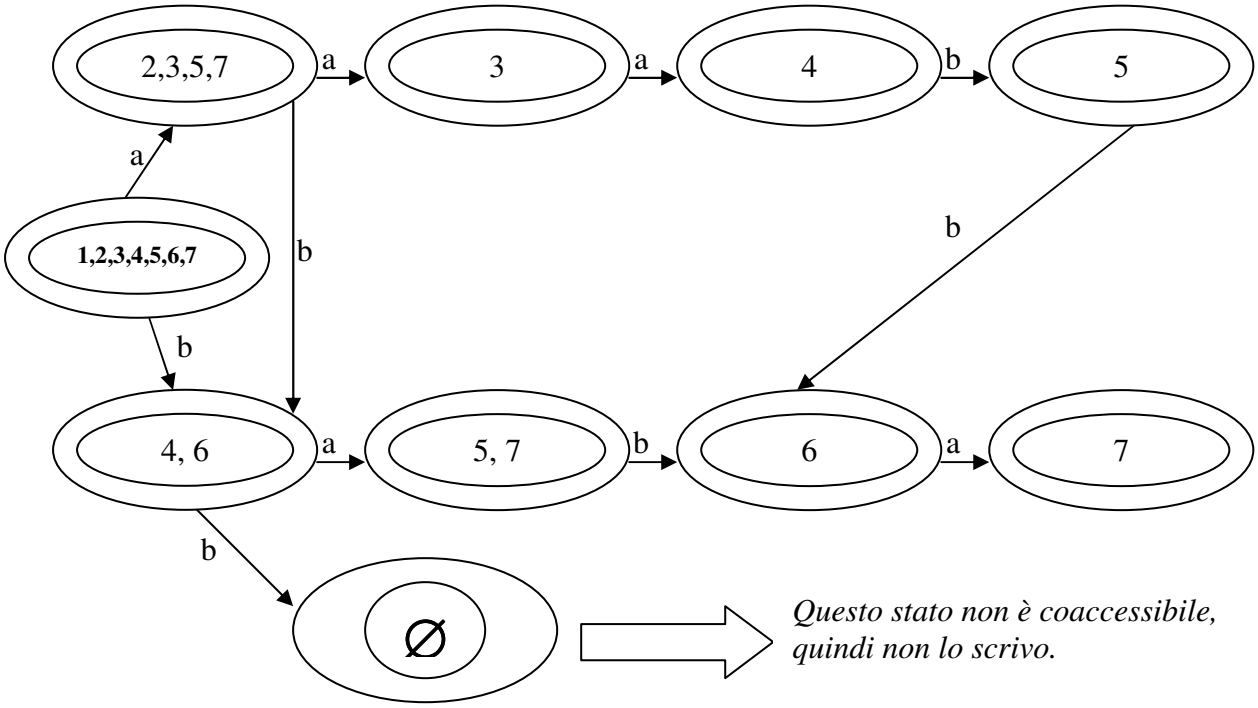


Se prendo tutti gli stati come stati iniziali e di accettazione, uno dei problemi sarebbe : "Da dove comincio?", ma non ci preoccupiamo di questo e vediamo che possiamo ugualmente costruire il nostro **DFA**, ma facciamo un esempio :

Esempio :
Consideriamo t = a a b a b a, il nostro **NFA** sarà :



Dove attraverso la solita Subset-construction il nostro **DFA** sarà così illustrato :



Il tempo di ricerca di tale automa sarà :

Tempo di ricerca = Tempo di costruzione (proporzionale a **n**) + **k** (lunghezza del pattern)

A questo punto è opportuno chiedersi : "Se ho un'esplosione esponenziale durante la costruzione di tale automa?". La risposta a tale domanda la troviamo nell'enunciato che segue :

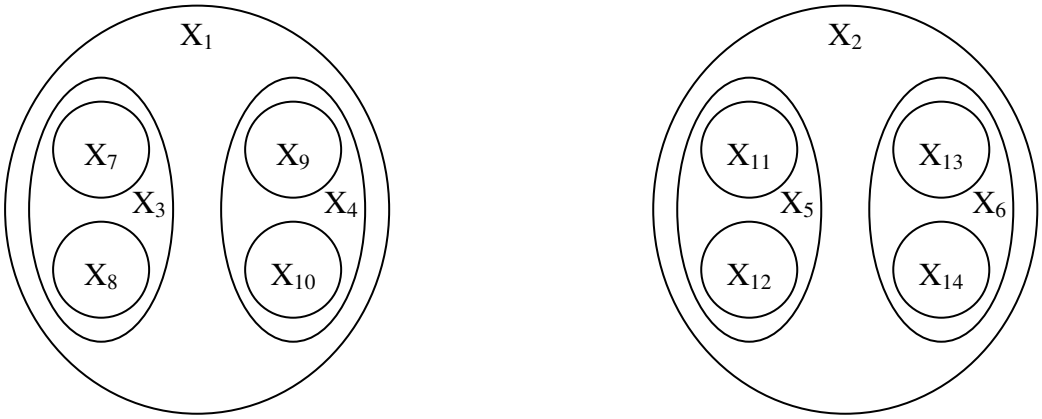
Teorema :
Dato un **NFA** con **n** stati, se costruiamo il **DFA** corrispondente, questo avrà al massimo 2n stati, e il numero di archi sarà al massimo di 3n.

Dimostrazione :
Per dimostrare tale teorema è opportuno prima dare una definizione :

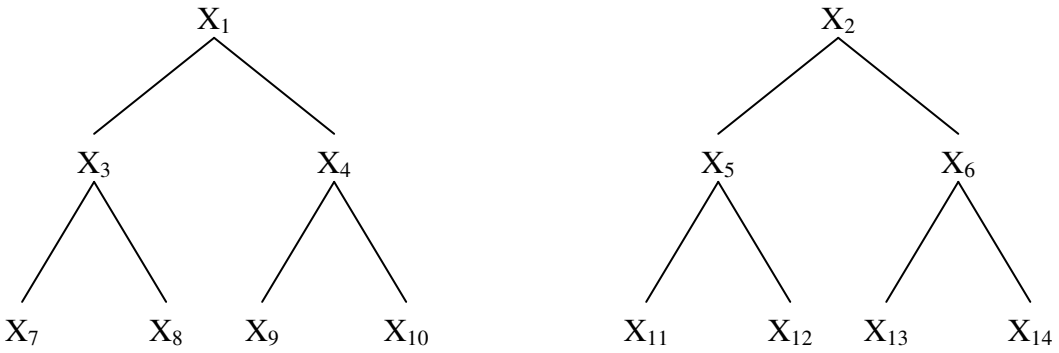
Definizione :
Siano dati due insiemi **S** ed **I**; con **S** insieme dei sottoinsiemi di **I**. **S** è detta *famiglia disgiuntiva di I*, se comunque presi X_1, X_2 appartenenti a **S** :

o $X_1 \cap X_2$ = insieme vuoto
o X_1 è sottoinsieme di X_2

Quindi dato un insieme : $|I| = n$
se **S** è una famiglia disgiuntiva di **I** allora : $|S| = 2n$
Infatti una famiglia disgiuntiva, secondo la definizione data la possiamo vedere così :



Cioè se S = {X_1, X_2, X_3, …, X_n} , due insiemi, o avranno intersezione vuota (ad esempio X_1, X_2), o uno sarà sottoinsieme dell'altro (ad esempio X_1, X_3).
Quindi una famiglia disgiuntiva è una struttura gerarchica ad albero :



Dove il massimo numero di foglie è **n**, e il numero massimo di elementi è ≤ 2n.

Dopo avere concluso tale definizione basta solo dire che quando si applica la Subset-construction su un automa di tipo **NFA**, gli elementi del **DFA** sono una famiglia disgiuntiva dell'**NFA**, e quindi gli elementi saranno ≤ 2n.
A questo punto possiamo dedurre che ci siamo assicurati di non avere un'esplosione esponenziale, almeno in questo caso.

Lezione 13 08/11/2002

Adesso introduciamo uno strumento che ci permette di stabilire se, dato un linguaggio **L**, esiste un automa a stati finiti in grado di riconoscerlo. Questo stabilisce una proprietà che devono avere le stringhe, per cui se questa non è soddisfatta allora non esiste un automa a stati finiti in grado di riconoscerle.

Lemma di Iterazione :
Dato un linguaggio **L** sottoinsieme Σ^*, se questo è riconosciuto da un **FSA**, esiste un intero positivo **N** tale che comunque prendo **w** appartenente a **L**, e comunque prendo una decomposizione di $w = w_1 * w_2 * w_3$ (con $|w_2| \geq N$), allora $w_2 = xyz$ (con $|y| > 0$) e si ha che comunque prendo $k \geq 0$, $w = w_1 * xy^kz * w_3$ appartiene a **L**.

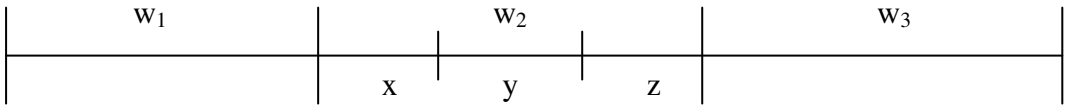
Dimostrazione :
Data una stringa **w** appartenente al linguaggio **L** :

$$w = w_1 * w_2 * w_3$$

dove :

$$w_2 = xyz$$

la possiamo rappresentare graficamente in modo :



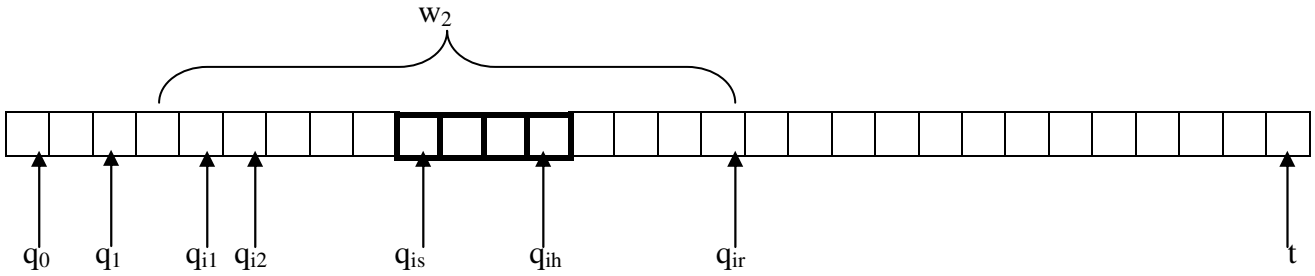
In questo caso **y** è il fattore iterante, se prendo un automa :

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

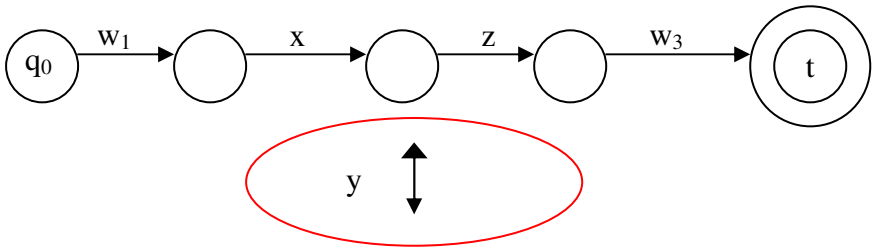
con :

$$N = |Q| \qquad e \qquad L(A) = L$$

quindi **N** è il numero di stati dell'automa, ma rappresentiamo la stringa **w** appartenente a **L** :



dove t è lo stato di accettazione.
Dal teorema se cancello questo pezzetto la stessa stringa è accettata dall'automa.
Ma per capire meglio cosa succede rappresentiamolo con un grafo:



la parte sottolineata in rosso rappresenta un pezzo del cammino w_2 che passa più volte in uno stesso stato, per cui lo posso anche cancellare e la stringa sarà ugualmente accettata.
La proprietà appena dimostrata è una condizione necessaria ma non sufficiente, perché se un certo linguaggio L è riconosciuto questa proprietà non vale. Adesso vediamo una possibile applicazione:

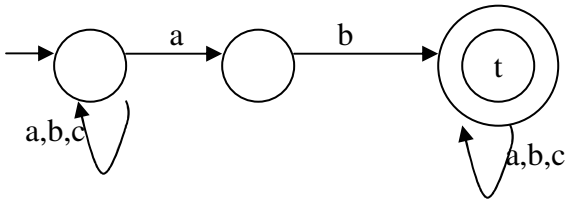
Esempio
Il linguaggio delle parentesi non può essere riconosciuto da un automa a stati finiti, infatti come sappiamo in tale linguaggio:

$$\Sigma = \{(,)\} = \{a, b\}$$

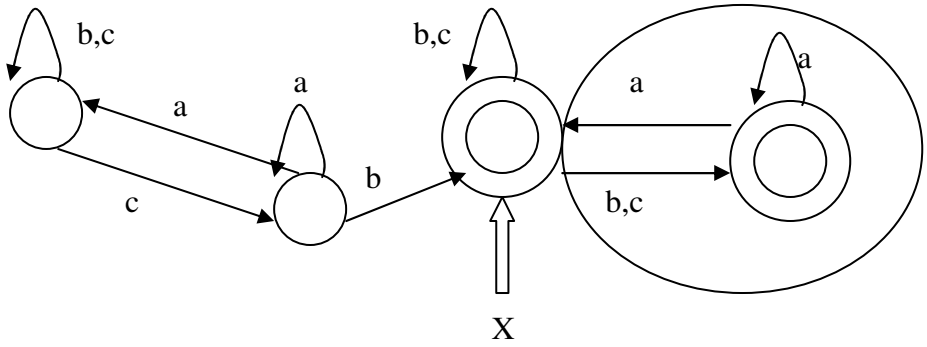
se considero una stringa di questo tipo:

$$(((((. . . . . ( )))) . . . . .))) = aaaaaa. . . .aaabbbb . . . .bbb$$

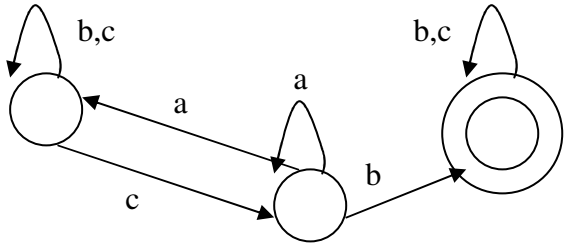
se tolgo il pezzo denotato con y, come nel teorema precedente l'automa non riconoscerà più la stringa, perché come sappiamo nel linguaggio delle parentesi ad ogni parentesi aperta ne corrisponde una chiusa.
Ritornando sull'applicazione della Subset-construction possiamo ancora fare delle considerazioni una delle quali è la seguente:
consideriamo un semplice automa NFA:



Applicando la Subset-construction il DFA sarà il seguente:



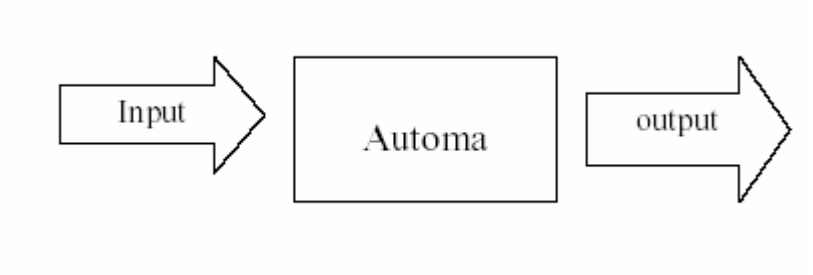
in questo caso notiamo che nel punto X, in qualsiasi maniera mi muovo mi troverò in uno stato di accettazione, quindi il DFA appena visto potrebbe anche essere rappresentato togliendo la parte sottolineata e aggiungendo una a allo stato X



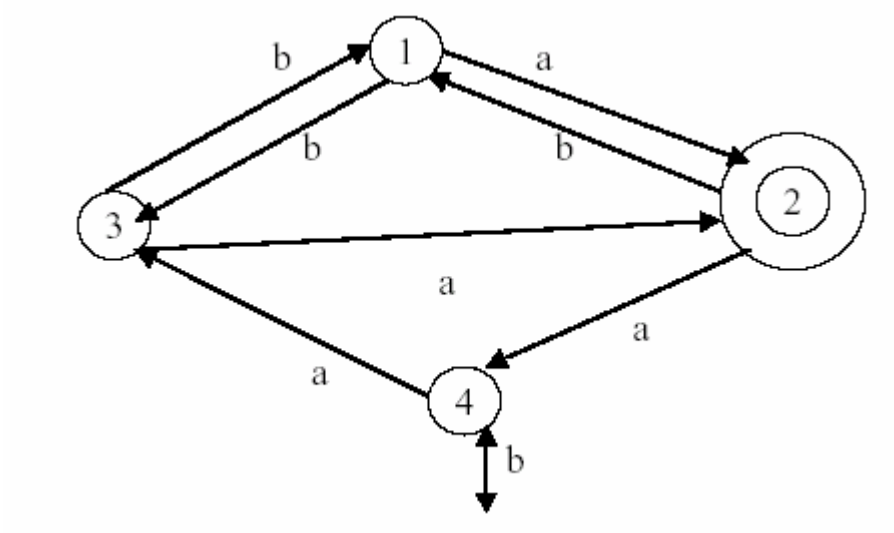
la domanda è: "Esiste un altro automa più piccolo?"
la risposta è negativa, infatti vi è l'esistenza di un teorema che dica che dato un automa NFA esiste un solo automa minimale DFA.

Tale problematica può essere vista considerando la riduzione, ossia in che maniera possiamo ridurre un automa?

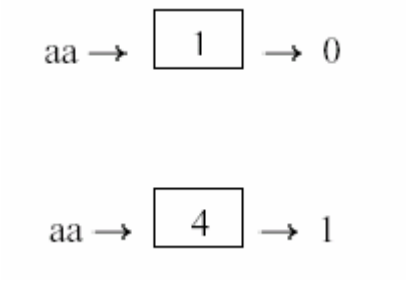
Riduzione
Consideriamo un automa A di cui sconosciamo il suo funzionamento, allora se vogliamo sapere in che maniera questo è costruito, mandiamo un informazione in input, e in base all'output possiamo dedurre il suo funzionamento.



Esempio
Per fare il nostro esempio consideriamo un automa rappresentato tramite un grafo:



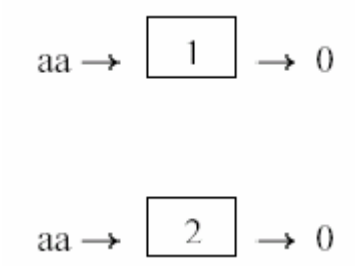
proviamo a mandare delle stringhe in input a questo automa:



quindi se mandiamo in input la stringa aa e l'automa si trova nello stato 1 la stringa non sarà riconosciuta e segneremo uno 0 in uscita, se invece mandiamo la stessa stringa aa ma l'automa si trova nello stato 4 la stringa sarà riconosciuta e segneremo un 1 in uscita.
Nel primo caso diremo che:

aa distingue (1,4)

mentre nel caso in cui:



diremo che:

aa non distingue (1,2)

queste stringhe vengono dette esperimenti.
Ma adesso passiamo a una definizione più formale di quanto detto fino ad ora.

Definizione
Dato un automa A, due stati (p, q) si dicono distinguibili se esiste un esperimento (stringa) che li distingue, quindi dato:

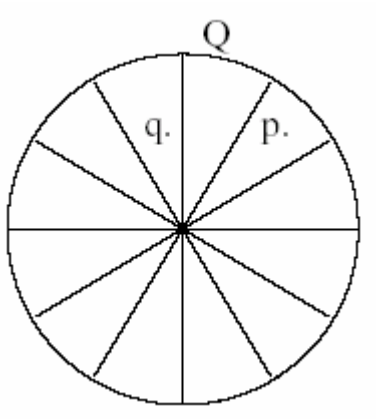
$$A = (\Sigma, Q, q_0, F, \delta)$$

se considero due stati p e q:
p I q se comunque prendo w appartenente a Σ^* (δ(p, w)appartiene a F $\Leftrightarrow$ δ(q, w) appartiene a F)
dove I è una relazione di equivalenza, che individua una partizione di Q .
Se denotiamo:

$$A_R = (\Sigma, Q_R, q_{0R}, F_R, \delta_R)$$

un automa ridotto, allora:

- $Q_R = Q / I$
  ossia l'insieme delle classi di equivalenza di Q:



Con la simbologia [q] denotiamo la classe che contiene q, dove q è un elemento qualsiasi della classe e un rappresentante della classe stessa.
Quindi secondo la definizione data prima:

$$[q] = [p] \Leftrightarrow q \; I \; p$$

- $q_{0R} = [q_0]$

- F_R = {[q] tale che q appartiene a F}

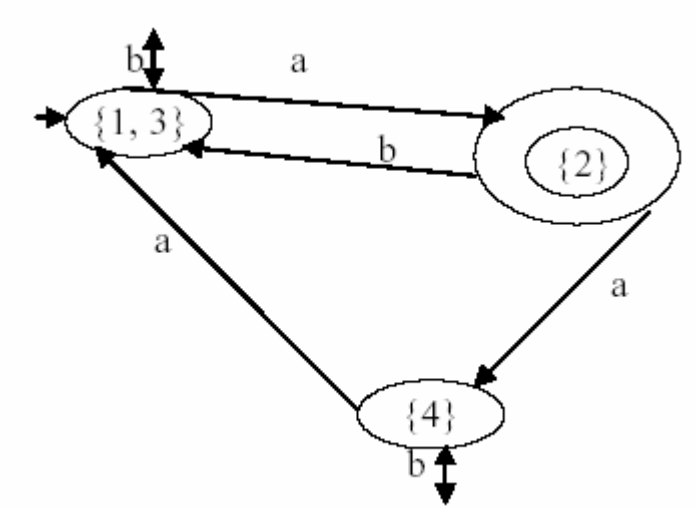
Infatti F è unione di classi di I o se ho q I p : q appartiene a F ⇔ p appartiene a F

- δ_R ([q], a) = [δ (q, a)]

ma adesso vediamo dall'esempio precedente come si ricava l'automa ridotto.
Prendiamo tutte le partizioni indistinguibili che appartengono a Q che sono:

$$\{1, 3\} ; \{4\} ; \{2\}$$

e costruiamo l'automa ridotto prendendo tanti stati quanti sono le classi:



Riduzione di un DFA

Precedentemente abbiamo visto che dato un automa:

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

potevamo definire l'indistinguibilità tra due elementi come segue:
p I q se comunque prendo w appartenente Σ^* (δ(p, w)appartiene a F ⇔ δ(q, w) appartiene a F)
altresì attraverso questa definizione abbiamo definito l'automa ridotto:

$$A_R = (\Sigma, Q_R, q_{0R} , F_R, \delta_R)$$

dove:

- $Q_R = Q / I$, ossia l'insieme delle classi di equivalenza di Q
- $q_{0R} = [q_0]$ la classe di q_0 dove q_0 è il rappresentante
- F_R = {[q] tale che q appartiene a F}
- δ_R ([q], a) = [δ (q, a)]

a questo punto dobbiamo dimostrare che L(A) = L(A_R), o meglio che l'automa iniziale è uguale all'automa ridotto, facciamolo attraverso dei passaggi qui sotto descritti, e partiamo dal linguaggio L definito dall'automa ridotto:

$$L(A_R) = \{ \text{ w appartenente } \Sigma^* / \delta_R (q_{0R} , w) \text{ appartiene a } F_R \} =$$
$$= \{ \text{ w appartenente } \Sigma^* / \delta_R ([q_0], w) \text{ appartiene a } F_R \} =$$
$$= \{ \text{ w appartenente } \Sigma^* / [\delta (q_0 , w)] \text{ appartiene a } F_R \} =$$
$$= \{ \text{ w appartenente } \Sigma^* / \delta (q_0 , w) \text{ appartiene a } F \} = L(A)$$

quindi alla fine otteniamo quella che è la definizione del linguaggio riconosciuto dall'automa di partenza.
La parte dimostrativa di p I q però non è algoritmica, infatti dati due stati qualsiasi di un automa chi ci dice che questi sono indistinguibili?
Bisognerebbe controllare i due stati per tutte le possibili stringhe definite su un determinato alfabeto, ma è ovvio che queste possono essere infinite.
La soluzione è di prendere un $k \geq 0$ e riscrivere la definizione dell'indistinguibilità così come segue:

$$p\ I_k\ q \text{ se comunque prendo una w appartenente a } \Sigma^*$$
$$\text{con } |\ w\ | \leq k \text{ risulta che } (\delta(p, w) \text{ appartiene a } F \Leftrightarrow \delta(q, w) \text{ appartiene a } F)$$

da questa relazione che abbiamo dato si vede subito che $p\ I_k\ q$ è calcolabile, se due stati sono indistinguibili per stringhe di lunghezza uguale a k, allora lo saranno per stringhe di lunghezza uguale a k-1
quindi:

$$p\ I_k\ q \quad \Rightarrow \quad p\ I_{k-1}\ q$$

allo stesso modo dobbiamo dimostrare che se esiste un intero k tale che :

$$I_k = I_{k+1},\ I_k = I_{k+2}\ , \ldots\ldots\ I_k = I_{k+r} \text{ per ogni } r \geq 1$$

Questo significa che $I_k = I$ e che quindi i due stati sono indistinguibili sempre.

Dimostrazione
Innanzitutto diciamo che due stati sono zero equivalenti quando:
$p\ I_0\ q$ se (q appartiene a $F \Leftrightarrow$ p appartiene a F) ossia p e q sono stati di accettazione

Se diamo la relazione che :

$$p\ I_{k+1}\ q \text{ se } (p\ I_0\ q) \text{and (comunque prendo una w appartenente a } \Sigma^*$$
$$\text{con } |\ w\ | \leq k \text{ risulta che } (\delta(p, w) \text{ appartiene a } F \Leftrightarrow \delta(q, w) \text{ appartiene a } F))$$

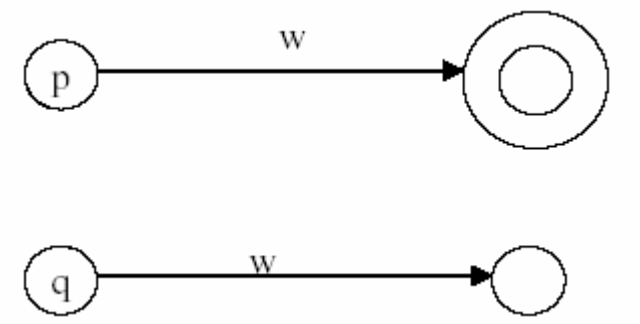
se poniamo w = av, la relazione precedente diventa:

$$p\ I_{k+1}\ q \text{ se } (p\ I_0\ q) \text{ and (comunque prendo una a appartenente a } \Sigma \text{ e una v appartenente a } \Sigma^*$$
$$|\ v\ | \leq k \text{ risulta che } (\delta(p, av) \text{ appartiene a } F \Leftrightarrow \delta(q, av) \text{ appartiene a } F))$$

se scindiamo ancora la relazione otterremo che :

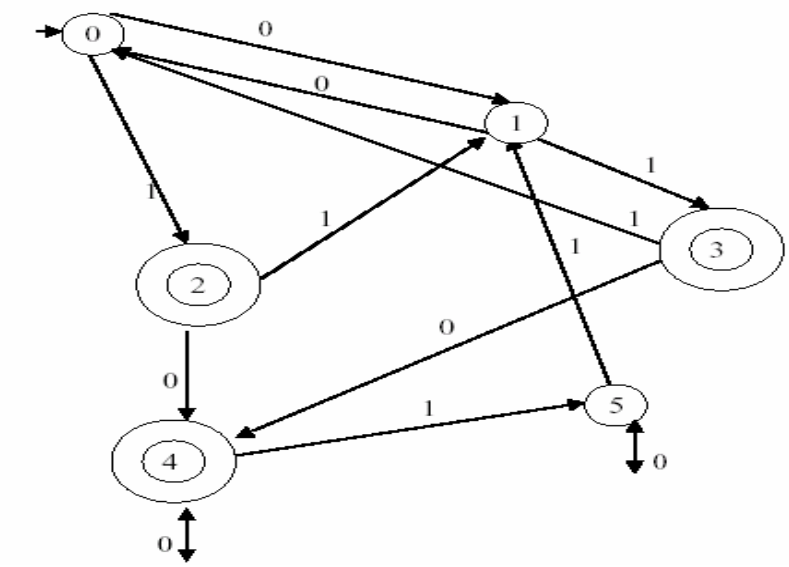
$$p\ I_{k+1}\ q \text{ se } (p\ I_0\ q\ ) \text{ and (comunque prendo una a appartenente a } \Sigma \text{ e una v appartenente a } \Sigma^*$$
$$|\ v\ | \leq k \text{ risulta che } (\delta(\delta\ (p, a), v) \text{ appartiene a } F \Leftrightarrow \delta(\ \delta\ (q, a), v) \text{ appartiene a } F))$$

Algoritmo per la riduzione
Per trattare meglio questo problema conviene ragionare sulla distinguibilità di due stati, invece che sulla indistinguibilità.
Quindi definiamo la distinguibilità come segue:
$p\ D\ q$ se esiste una w appartenente a Σ^*
tale che ($\delta(p, w)$ appartiene(o non appartiene) a $F \Leftrightarrow \delta(q, w)$ non appartiene (o appartiene) a F)
vediamo graficamente cosa succede:



A questo punto mostriamo l'algoritmo con un esempio per meglio capire.

Esempio
Prendiamo un alfabeto $\Sigma = \{0, 1\}$ e costruiamo un automa NFA che comparirà in questo modo:



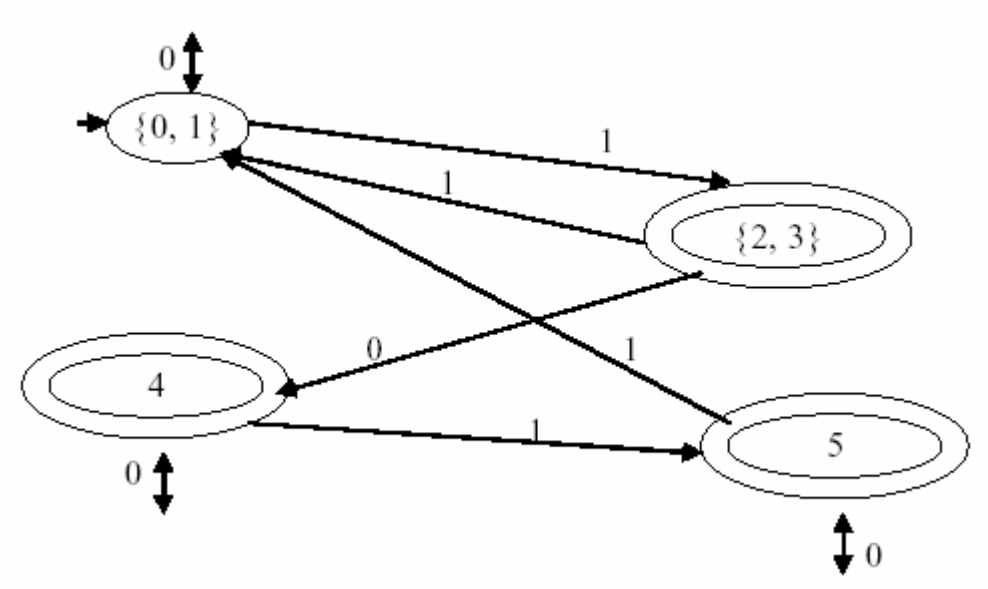
a questo punto costruiamo una tabella dove segniamo soltanto gli stati distinguibili e per quale valore questi sono distinguibili:

1					
2	D_0	D_0			
3	D_0	D_0			
4	D_0	D_0	*	*	
5	D_1	D_1	D_0	D_0	D_0
	0	1	2	3	4

Alla fine le classi indistinguibili saranno le seguenti:

$$\{0, 1\}, \{2, 3\}, \{4\}, \{5\}$$

e l'automa ridotto ci apparirà in questo modo:



attraverso l'esempio che abbiamo fatto adesso descriviamo l'algoritmo come segue:

- Passo 1.
  Marco gli stati (p, q) se sono distinguibili, quindi se p appartiene a F e q non appartiene F (o viceversa).
- Passo 2.
  Repeat
  se esiste (p, q) non marcata tale che (δ(p, a), δ(q, a)) è marcata per qualche a che appartiene a Σ, allora marca (p, q)
  Until
  Non avviene più alcuna modifica nella tabella
- Passo 3.
  p I q se (p, q) non è marcata
  .

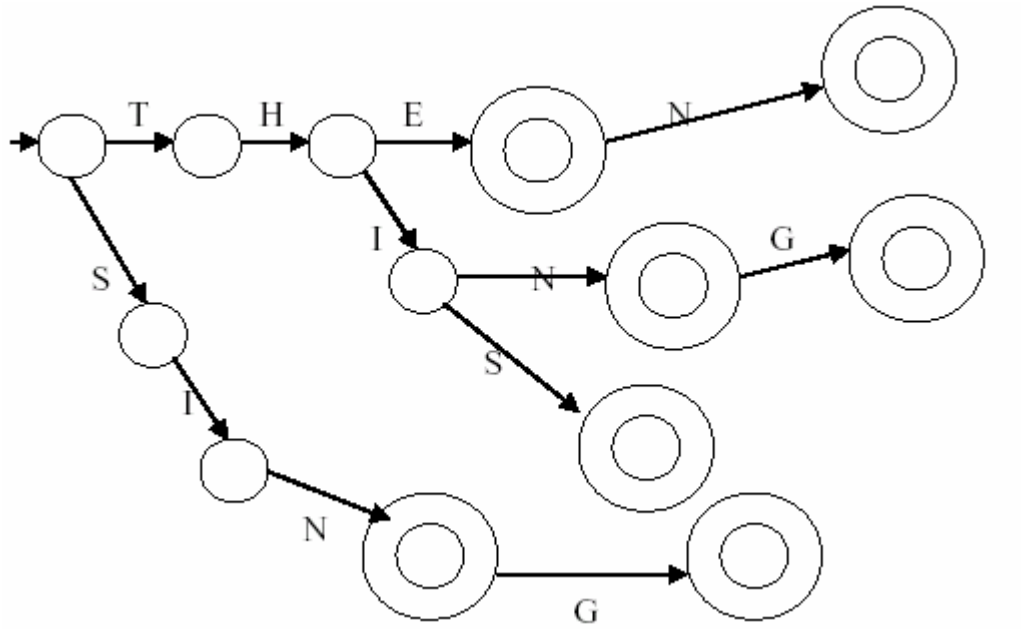
Le costruzioni che abbiamo finora rappresentato sono la Riduzione che ci permette di passare da un DFA dato ad un DFA più piccolo, e la Subset-construction che ci permette di passare da un modello di automa non deterministico(NFA) a un modello di automa deterministico(DFA).

Problema del Dizionario

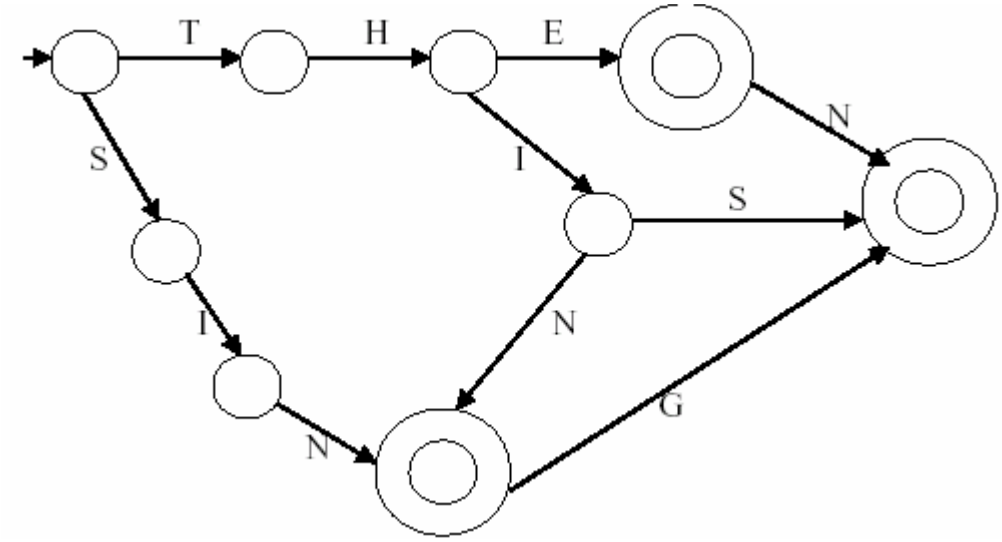
Supponiamo adesso di avere le seguenti parole inglesi:
- THE
- THEN
- THIN
- THING
- THIS
- SIN
- SING

Se costruiamo un automa DFA che riconosce queste stringhe, il numero degli stati sarà uguale al numero dei caratteri più il numero delle stringhe.

Infatti ci apparirà in questo modo:



Mentre l'automa ridotto sarà:



Se ho due automi A_1 e A_2, e voglio sapere se sono equivalenti, cioè riconoscono lo stesso linguaggio, ovvero $L(A_1) = L(A_2)$. Quello che facciamo è la **riduzione** di A_1 e di A_2

Se $A_{1\,R}$ e $A_{2\,R}$ sono uguali allora A_1 e A_2 sono equivalenti.
Dall'algoritmo di riduzione possiamo risolvere in modo algoritmico il problema dell'equivalenza.
Sappiamo che per un FSA tutti i problemi di decisione sono decidibili :

L'**Emptyness Problem** è decidibile, infatti se rappresentiamo l'automa col grafo degli stati, si tratta di vedere se gli stati finali sono accessibili, ossia se esiste un percorso dallo stato iniziale a uno degli stati di accettazione; siccome ho un grafo finito, si può andare ad esplorare tutti i percorsi, quindi è un problema di connessione.

Il **Fitness Problem** si riconduce pure ad algoritmi in grafi classici, perché se prendo stati accessibili (ossia, partendo da uno stato iniziale, esiste un cammino che mi porta a questo stato) e coaccessbili (ossia partendo da questo stato, esiste un cammino che mi porta ad uno stato di accettazione) , si tratta di vedere se ci sono cicli; in tal caso il linguaggio è infinito, altrimenti è finito.

L'**Inclusion Problem** è un problema più complicato rispetto all'EQP, infatti, se so risolvere l'INP allora saprò risolvere l'EQP, ma se so risolvere l'EQP, questo non implica che sappia risolvere l'INP.
Se so risolvere l'INP basta che applico due volte : $L(A_1) \subseteq L(A_2)$, $L(A_2) \subseteq L(A_1)$, allora $L(A_1) = L(A_2)$.
$L_1 \subseteq L_2$ equivale a dire $L_1 \cap L_2^C = \varnothing$
Se ho l'automa A_1 che riconosce L_1 e A_2 che riconosce L_2, mi costruisco l'automa che riconosce L_2^C e pertanto mi posso costruire l'automa che riconosce $L_1 \cap L_2^C$, dopodiché devo verificare se il linguaggio riconosciuto da questo automa è uguale a $\varnothing$.

Lezione 14 11\11\2002

Definiamo le equivalenze definite su $\sum^*$

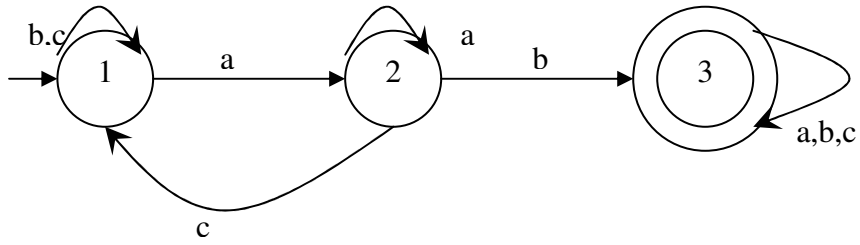
Proprietà delle equivalenze:
Date due stringhe u,v, appartenenti a $\sum^*$, in cui è definita un'operazione che si chiama concatenazione (con L sottoinsieme di $\sum^*$), se abbiamo la relazione di equivalenza:

$$u \rho v$$

definiamo la proprietà:

1. ρ è invariante a destra, cioè: u ρ v implica che per ogni w che appartiene a a $\sum^*$, allora uw ρ vw
2. ρ è compatibile con L se L è unione di classi ρ, o meglio: u ρ v implica che $u \in L \Leftrightarrow v \in L$.

Esempio:
Dato il seguente automa NFA:



Dalla definizione precedente sappiamo che due stati sono equivalenti se partendo da uno stato iniziale, arrivo ad uno stesso stato.
Adesso ci ricaviamo il DFA del seguente automa e sappiamo che:

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

Dato questo automa è possibile definire la seguente relazione di equivalenza:

$$u \rho_A v \text{ se } \delta^*(q_0,u)= \delta^*(q_0,v)$$

Esempio:
La stringa baaca = cba, infatti notiamo che:

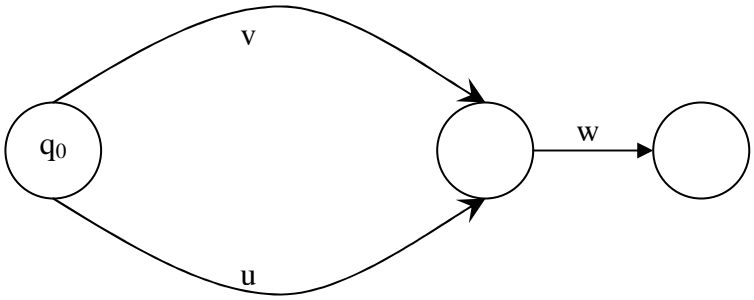
$$\delta^*(1,baaca) = 2$$
$$\delta^*(1, cba) = 2$$

questa equivalenza ci fa capire che a un certo punto l'automa con la sua capacità di memoria non riesce più a distinguere le stringhe; infatti tute e due lo portano allo stesso stato.
Un'altra considerazione che bisogna fare è che il numero delle classi di equivalenza sarà uguale al numero degli stati presenti nell'automa.
Si definiscono adesso le classi di equivalenza così come segue:

- R_1 : Tutte le stringhe che non contengono ab e non terminano per a
- R_2 : Tutte le stringhe che non contengono ab e terminano per a
- R_3 : Tutte le stringhe che contengono ab.

Secondo quanto detto prima, tale equivalenza avrà le seguenti proprietà:

1. ρ_A è invariante a destra, quindi: u ρ_A v implica che uw ρ_A vw
   graficamente significa che:



2. ρ_A è compatibile con L(A) (Linguaggio riconosciuto dall'automa considerato) quindi:
   u ρ_A v se $\delta^*(q_0, u) = \delta^*(q_0, v) \Leftrightarrow$ u,v appartengono a L(A).
3. ρ_A ha un indice finito: i $\rho_A = |A|$

Quindi dato un qualsiasi automa possiamo associare la relazione di equivalenza che ha le proprietà precedentemente elencate.
Ovviamente possiamo fare anche un procedimento inverso, ossia data una relazione di equivalenza che soddisfa le tre proprietà elencate vi possiamo associare un automa corrispondente.

Esempio:
Sia ρ una relazione di equivalenza $\sum^*$ di indice finito, invariante a destra, e compatibile con L sottoinsieme di $\sum^*$:

$$A \rho = (\sum, Qp, q_{0p}, F_p , \delta_p) \text{ e } A \rho = \sum^*/\rho$$

Lezione 15 15\11\2002

Adesso trattiamo un nuovo tipo di automa, detto automa Two-way o come lo abbiamo denominato noi automa 2-DFA.

Automa Two-way(o 2-DFA)

Questo tipo di automa è composto come un semplice automa DFA con l'unica differenza che cambia la funzione di transizione, quindi nella rappresentazione della quintupla avremo sempre la stessa rappresentazione letterale:

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

Con la differenza che δ sarà così definita:

$$\delta : Q \times \Sigma \rightarrow Q \times$$

dove:

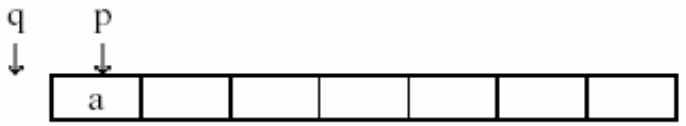
$$\Delta = \{-1, 0, 1\}$$

quindi facendo un esempio avremo che:

$$\delta (q, a) = (p, i) \text{ con } i = 0, 1, -1 \text{(si muove di i caselle)}$$

una volta definito il nostro automa dobbiamo vedere quando una stringa è accettata.
Ci possiamo trovare davanti a tre casi:

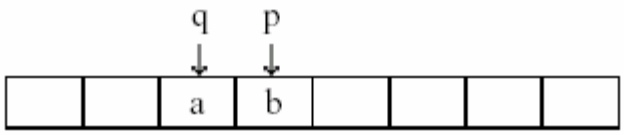
1. L'automa esce a sinistra: dopo un certo numero di mosse l'automa si trova a leggere il primo carattere in uno stato p, tale che $\delta (p, a) = (q, -1)$, graficamente:



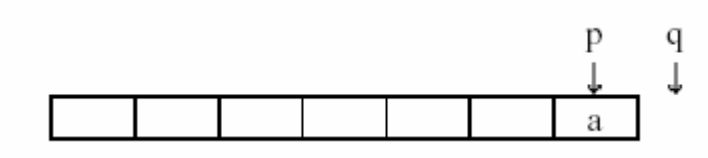
2. L'automa entra in un ciclo e non esce mai (loop infinito):dopo un certo numero di mosse ci troveremo nella condizione:

$$\delta (q, a) = (p, +1)$$
$$\delta (q, a) = (q, -1)$$

graficamente:



3. L'automa esce a destra:dopo un certo numero di mosse l'automa si trova a leggere il primo carattere in uno stato p tale che $\delta (p, a) = (q, +1)$, graficamente:



Nei due casi in cui l'automa esce a destra o a sinistra, la stringa sarà accettata solo nel caso in cui si verifica una di queste due condizioni, e ovviamente q sarà uno stato di accettazione.

A questo punto possiamo dire che:

$$L\text{(1-DFA) sottoinsieme non proprio di } L\text{(2-DFA)}$$
$$L\text{(1-DFA)} = L\text{(2-DFA)}$$

Ovviamente dobbiamo dimostrare quanto detto come segue:

Dimostrazione
L = L(A) infatti se consideriamo la relazione di equivalenza:

$$u\ \rho_L\ v \text{ se comunque prendo una stringa w appartenente a } \Sigma^* \text{ risulta che}$$
$$(uw \text{ appartiene a } L \Leftrightarrow vw \text{ appartiene a } L) \text{ quindi L appartenente a } L\text{(1-DFA)} \Leftrightarrow i(\rho_L) < \infty$$

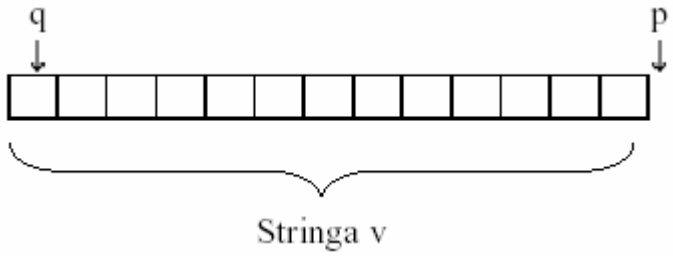
Introduciamo adesso due funzioni:
dato un automa 2DFA e data una stringa v appartenente a Σ^* :

$$\begin{cases} f_v : Q \rightarrow Q \cup \{\not{q}\}, \text{ con q che non appartiene a Q} \\ g_v : Q \rightarrow Q \cup \{\not{q}\}, \text{ con q che appartiene a Q} \end{cases}$$

definendole operativamente vediamo che succede:
q se esce a sinistra o non esce mai

$$f_v(q) = \begin{cases} \not{q} \text{ se esce a sinistra o non esce mai} \\ p \text{ se esce a destra nello stato p} \end{cases}$$

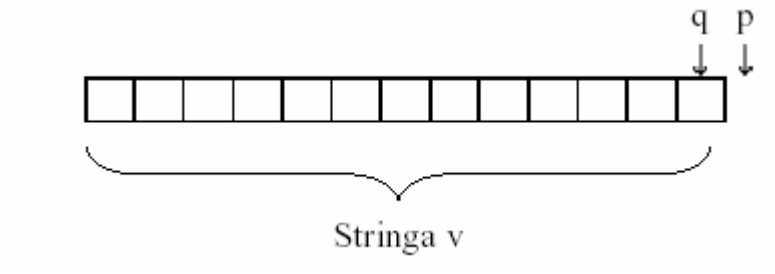
quindi per definire la funzione f_v prendo la stringa v e devo vedere quanto vale la funzione $f_v(q)$.
Supponiamo che l'automa si trovi nello stato q che punta al primo carattere della stringa:



Quindi se esce a sinistra o va in loop la funzione assumerà il valore q , mentre se esce a destra sullo stato p la funzione assumerà il valore p.
Allo stesso modo definiamo l'altra funzione:

$$g_v(q) = \begin{cases} q \text{ se esce a sinistra o non esce mai(va in loop)} \\ p \text{ se esce a destra nello stato p} \end{cases}$$

anche qui per vedere quanto vale la mia funzione g_v, supponiamo che l'automa si trovi nello stato q che punta all'ultimo carattere della stringa:



Quindi se esce a sinistra o va in loop la funzione assumerà il valore $\bar{q}$, mentre se esce a destra sullo stato p la funzione assumerà il valore p.
A questo punto date questa due funzioni mi definisco un'equivalenza dove:

$$u\ T_A\ v \text{ se } (f_v, g_v) = (f_u, g_u)$$

Una volta definita questa equivalenza dobbiamo dimostrare che questa ha indice finito, che è invariante a destra, e T_A è compatibile con L .
Quindi avremo:

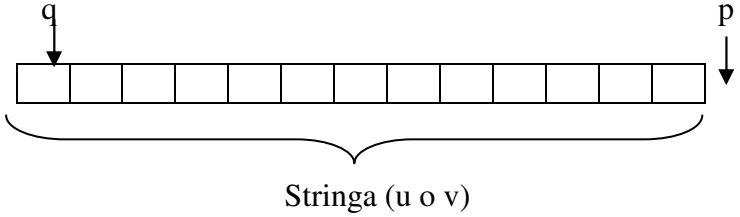
1. Ha indice finito perché il numero di possibili coppie è n^{n+2} e quindi:

$$i(T_A) = n^{2n+2}$$

2. T_A è compatibile con L perché:

$$u\ T_A\ v \text{ implica che } (u \text{ appartiene a } L \Leftrightarrow v \text{ Appartiene a } L)$$

Questo vuol dire che se ho una stringa u e l'automa si trova sullo stato q che punta al primo carattere



Se esce a destra sappiamo che per definizione la nostra funzione f_u assumerà lo stato p che abbiamo assunto come stato di accettazione e quindi possiamo dire che la stringa considerata appartiene a L.

Altresì per definizione anche la stringa v appartiene a L perché nella definizione di equivalenza abbiamo assunto che:

$$f_u = f_v \text{ che implica che } f_u(q) = f_v(q) = p$$

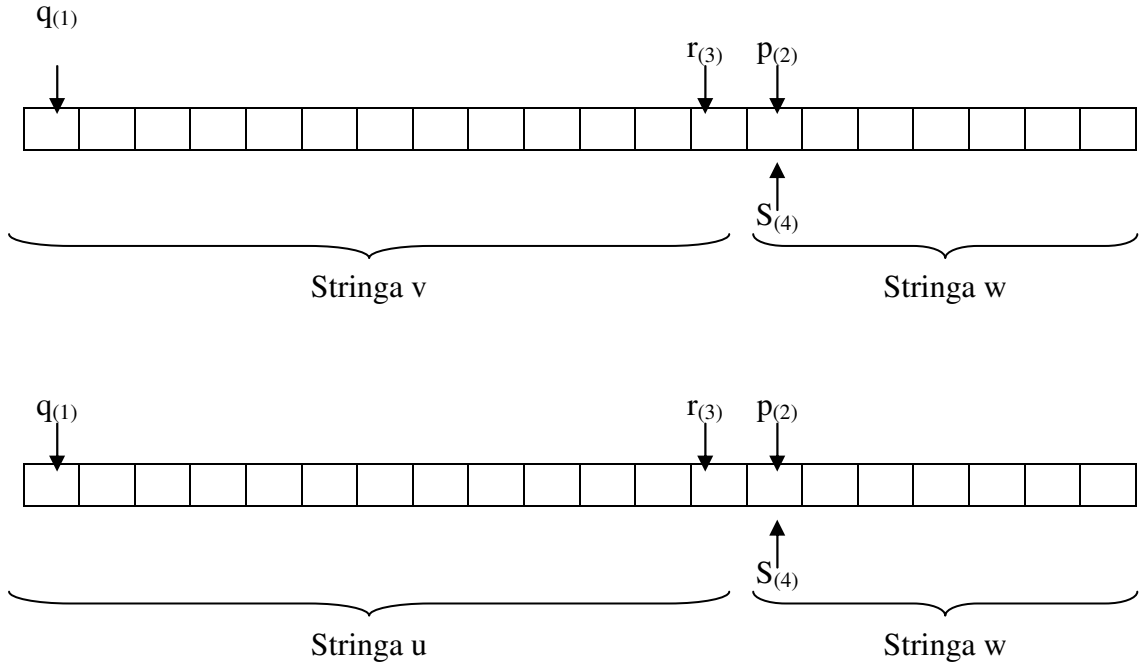
3. Inavariante a destra perché:

$$u\ T_A\ v \text{ implica che } uw\ T_A\ vw$$

quindi:

$$f_u = f_v \text{ implica che } f_{uw} = f_{vw}$$
$$g_u = g_v \text{ implica che } g_{uw} = g_{vw}$$

Adesso vediamo graficamente come sfruttiamo le funzioni appena definite, facendo attenzione che la linea di demarcazione segna la fine della stringa v e l'inizio della stringa w:



Adesso vediamo caso per coso cosa succede quando mi trovo in determinati stati :

Caso 1 : Mi metto in un generico stato q, può succedere che non supero mai la linea di demarcazione e quindi avrò che :

$$f_u(q) = q = f_v(q)$$

poiché sappiamo che $f_u = f_v$, supponiamo che supero la linea una volta sola. Avrò che :

$$f_u = p = f_v(p)$$

Caso 2 : Supponiamo di superare la linea di demarcazione ed entriamo in w, poi rientriamo in u, ma siccome la stringa w è la stessa sia in u che in v, allora rientriamo anche in v.

Caso 3 : Adesso ci troviamo nello stato r, e qui entra in gioco la seconda funzione definita, ossia g. Infatti se riattraverso la linea di demarcazione avremo che :

$$g_u(r) = g_v(r) = s$$

Adesso proviamo che se la testa dell'automa attraversa la linea di demarcazione in u un numero di volte, succederà lo stesso in v.
Tutto questo si dimostra per induzione, infatti abbiamo visto che per n=1 è vera, supponiamo che vera per n volte; non ci resta che dimostrare per n+1 volte.
Vediamo come si comporta la funzione f : il ragionamento è molto semplice, infatti supponiamo che attraversiamo la linea di demarcazione n volte e ci troviamo nello stato s. Se attraversiamo n+1 volte ci troveremo nello stato r, e quindi a questo punto o non riattraverserò più la linea di demarcazione, o esco a sinistra, o resto a ciclare, e quindi avrò che :

$$f_{uw} = f_{vw}$$

Per la funzione g la dimostrazione è analoga, e quindi abbiamo dimostrato che T_A è invariante a destra.

Lezione 16 18/11/2002

***Operazioni sui linguaggi* :**
Sappiamo che i linguaggi sono sottoinsiemi di Σ^* e quindi :

$$L \subseteq \Sigma^*$$

Possiamo introdurre però delle operazioni sui linguaggi come ad esempio quelle booleane :

- $\cup$, Unione $\Rightarrow$ $L_1 \cup L_2$
- $\cap$, Intersezione $\Rightarrow$ $L_1 \cap L_2$
- ^{C} , Complementare $\Rightarrow$ L_1^C

Vediamo delle proprietà su queste operazioni appena definite :

Proprietà di chiusura della famiglia L(FSA) :
Come sappiamo questa è la famiglia riconosciuta dagli automi FSA. Cosa possiamo dire se L_1 e $L_2 \in$ L(FSA), e applichiamo le operazioni appena definite :

Teorema :
La Famiglia L(FSA) rispetto a tali operazioni è chiusa

Dimostrazione :
Quello che dobbiamo dimostrare è il seguente :

$$\text{Se } L_1 \text{ e } L_2 \in L(FSA) \quad \Rightarrow \begin{cases} L_1 \cup L_2 \in L(FSA) & \mathbf{1} \\ L_1 \cap L_2 \in L(FSA) & \mathbf{2} \\ L_1^C \in L(FSA) & \mathbf{3} \end{cases}$$

1 :
Supponiamo di avere due automi DFA A_1 e A_2 dove :

- $A_1 = (\Sigma, Q_1, q_{0\,1}, F_1, \delta_1) \Rightarrow L(A_1) = L_1$
- $A_2 = (\Sigma, Q_2, q_{0\,2}, F_2, \delta_2) \Rightarrow L(A_2) = L_2$

Ora supponiamo di avere un NFA A dove :

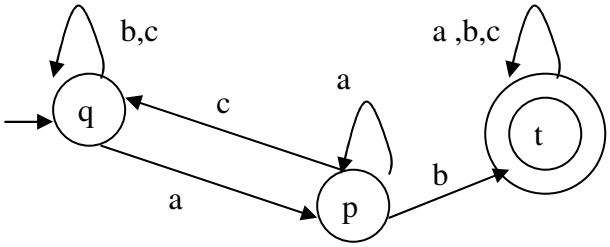
- $A = \{\Sigma, (Q_1 \cup Q_2), (q_{0\,1} \cup q_{0\,2}), F_1 \cup F_2, \delta\}$

E dove la transizione :

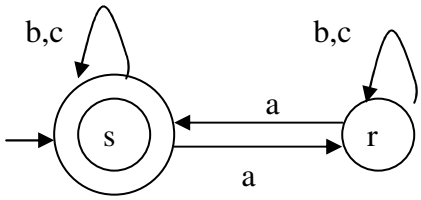
$$\delta(q,a) = \begin{cases} \text{si comporta come } \delta_1(q,a) \text{ se } q \in Q_1 \\ \text{si comporta come } \delta_2(q,a) \text{ se } q \in Q_2 \end{cases}$$

Adesso facciamo un esempio :

Esempio :
Supponiamo di avere un automa DFA A_1:

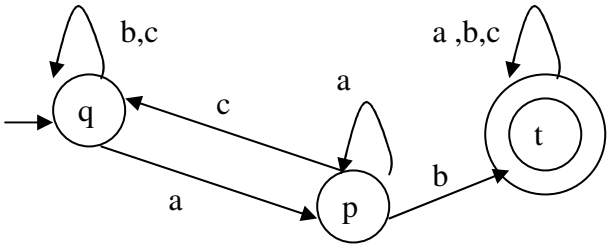


e quindi denotiamo con $L_1 = L(A_1)$
e poi un altro automa DFA A_2 :

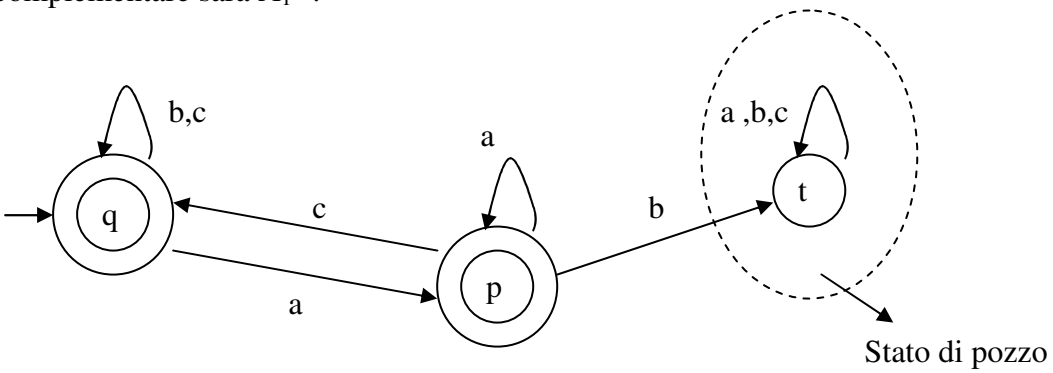


e quindi denotiamo con $L_2 = L(A_2)$.
L'automa di unione lo posso vedere come un unico automa $A = A_1 \cup A_2$ che genera un linguaggio $L = L(A_1) \cup L(A_2) = L(A)$.
Da notare che l'automa trovato è un NFA e se vogliamo trovare il DFA relativo basta applicare la subset construction.

3 :
Supponiamo di avere l'automa A_1 :



L'automa complementare sarà A_1^C :



Ovviamente questa operazione vale per gli automi completi, ma se ho un automa incompleto basta aggiungere uno stato di pozzo (nel caso precedente t). Quindi : $L_1^C = L(A_1^C)$.

2 :
Se applichiamo le leggi di De Morgan non si potrebbe dimostrare perché sappiamo che :

$$L_1 \cap L_2 = (L_1^C \cup L_2^C)^C$$

Supponiamo di avere sempre i due automi DFA A_1 e A_2 come definiti prima, l'automa intersezione A sarà :

$$A = \{\Sigma, Q_1 \times Q_2, (q_{0\,1}, q_{0\,2}), F_1 \times F_2, \delta\}$$

E dove :

$$\delta((q_1,q_2),a) = (\delta_1(q_1,a), \delta_2(q_2, a))$$

facciamo un esempio per capire meglio :

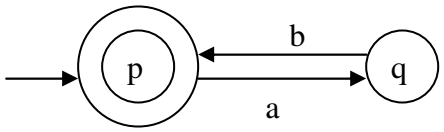
Esempio :
Supponiamo di avere due linguaggi :

- $L_1 = \{ (ab)^n / n \geq 0\} = \{\varepsilon, ab, abab, ababab, ...\}$
- $L_2 = \{ (a^nb / n \geq 0\} = \{\varepsilon, b, ab, ..., aaab, ...\}$

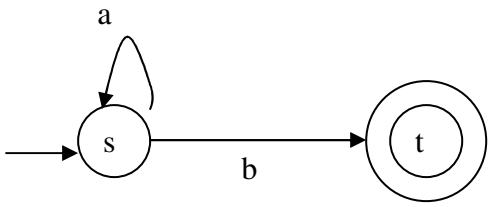
il linguaggio intersezione :

$$L = L_1 \cap L_2 = \{ab\}$$

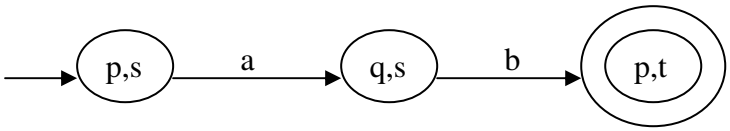
Adesso costruiamo l'automa A_1 che riconosce il linguaggio L_1 :



e l'automa A_2 che riconosce il linguaggio L_2 :



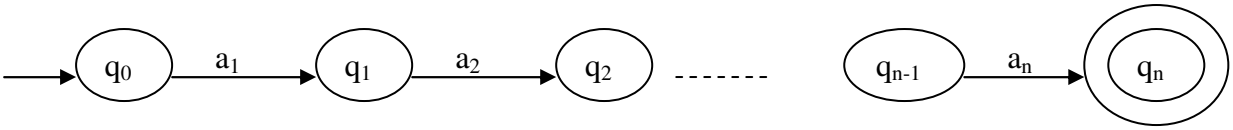
L'automa A che riconosce il linguaggio $L = L_1 \cap L_2$ sarà :



Teorema :
Un linguaggio fatto da una stringa è sempre riconosciuto da un automa, infatti se :

$L = \{w\}$ dove $w=a_1a_2a_3... a_n$ con $a_1...a_n \in \Sigma$

L'automa A che riconosce questo linguaggio sarà :



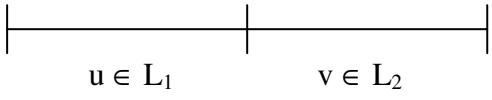
Questo teorema mi dice che posso costruire anche un linguaggio fatto da più stringhe finite.

Proposizione :
Tutti i linguaggi finiti e cofiniti (non finiti) sono riconosciuti da FSA.

Un altro tipo di operazione è la concatenazione che definiamo come segue :

Concatenazione (prodotto fra linguaggi) :
$L_1L_2 = \{uv / u \in L_1, v \in L_2\}$

Quindi se ho due stringhe u e v e le unisco, ottengo un'unica stringa uv che appartiene a L_1L_2.

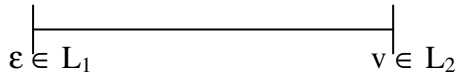


ovviamente bisogna definire meglio questa operazione e lo facciamo come segue :

$$\text{se } |L_1| = n \quad \text{e } |L_2| = m \quad \Rightarrow \quad |L_1L_2| \leq nm$$

Esempio :
se $L_1 = \{aa, aab, bb\}$
se $L_2 = \{ab, bab\}$
allora $L_1L_2 = \{aaab, aabab, aabab, aabbab, bbab, bbbab\}$ — Superfluo perché già presente

Un'altra proprietà :
se $\varepsilon \in L_1 \quad \Rightarrow \quad L_2 \subseteq L_1L_2$ infatti :



e vale anche il viceversa : se $\varepsilon \in L_2 \Rightarrow L_1 \subseteq L_1L_2$

Star (stella) :
L'operazione star (*) viene definita come segue :

$L^* = \{v_1v_2 \ldots v_n / n \geq 0\}$ con $v_i \in L$ e con $i = 1,2,\ldots,n$

Dove le nostre v_i rappresentano stringhe di L. Possiamo definire L^* come il sottomonoide generato da L. Ma sappiamo che $L \subseteq \Sigma^*$, ma possiamo vedere Σ^* come L^* e quindi L è contenuto dal sottomonoide L^*.
Tutte le operazioni definite sono dette *regolari*. Però le operazioni star e ^{C} sono più potenti delle altre perché partendo da linguaggi finiti otteniamo linguaggi infiniti.
Da notare che anche se $\varepsilon \notin L$, $\varepsilon \in L^*$ sempre quindi :

$L^* = \{\varepsilon\} \cup L \cup L^2 \cup \ldots$

Un'altra cosa è che in * e ^{C} sono definite operazioni unarie, mentre le altre binarie.
Quindi da quanto detto l'idea è quella di costruire dei linguaggi, partendo da linguaggi semplici e usando le operazioni viste. Ma prima definiamo i linguaggi elementari.

Linguaggi elementari :
Se ho $\Sigma = \{a, b, c\}$, posso costruire 5 linguaggi elementari che sono :

$$\varnothing, \varepsilon, \{a\}, \{b\}, \{c\}$$

questo formalismo che abbiamo introdotto ci risulta molto utile, dove partendo da linguaggi semplici o elementari ci costruiamo linguaggi più complessi.
Per semplificare la notazione si usano le espressioni regolari.

Espressioni Regolari :
Un'espressione regolare è un'espressione dove si aboliscono le parentesi {}; quindi :

$$(a^*b)^* \cdot a^* = (\{a\}^* \cdot \{b\})^* \cdot \{a\}^*$$

in modo ricorsivo per definire la sintassi si può dire che i simboli che uso sono :

$$\varnothing, \varepsilon, a, b, c, \ldots, +, ., *, (,)$$

e la sintassi viene definita ricorsivamente dicendo che :

- se E_1 e E_2 sono espressioni regolari $\Rightarrow$ $E_1 + E_2$, $E_1 . E_2$, $(E_1)^*$ sono espressioni regolari.

Lezione 17 21/11/2002

Espressioni Regolari :

Abbiamo visto che quando usiamo la seguente notazione e le seguenti operazioni abbiamo espressioni regolari :

$$\varnothing, \varepsilon, a \in \Sigma, +, . , *$$

Ma se in più usiamo le operazioni :

$$\cap, ^C$$

allora abbiamo le *espressioni regolari estese.*

Quindi definiamo con R la famiglia dei linguaggi regolari. Ma chi sono questi linguaggi regolari?

Esempio :

Supponiamo di avere un alfabeto Σ così definito :

$$\Sigma = \{a, b\}$$

ed un linguaggio L dove :

L = tutte le stringhe che non contengono ab

Queste stringhe quindi possono essere formate :

1. aaa……aa
2. bbb……bbaa……aa

Usando le espressioni regolari possiamo scrivere : $b^* a^*$

Esempio :

Supponiamo di avere un alfabeto Σ così definito :

$$\Sigma = \{a, b, c\}$$

e dove :

L = tutte le stringhe in cui non compare il blocco ab e che non terminano per a

Adesso proviamo a trovare un'espressione regolare :

questa sarà : $(b + a^* c)^*$ (la notazione + significa OR)

Proviamo a leggerla :

1. Non posso avere a seguita da b.
2. Non posso terminare per a, infatti dopo un certo numero di a avrò sempre almeno una c.

Da notare che quando abbiamo la star la lettera precedente a tale operazione può assumere il valore di parola vuota.

Posso avere espressioni regolari diverse che mi rappresentano lo stesso linguaggio.

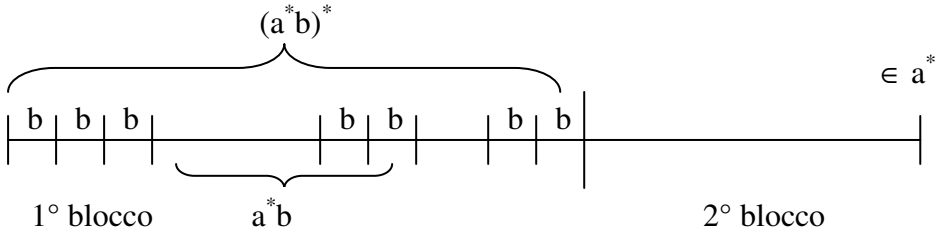
Esempio :
Se ho la seguente relazione :

$$(a^*b)^*a^* = (a + b)^* \quad ?$$

non posso stabilire immediatamente se sono uguali, posso farlo dimostrando l'inclusione inversia ossia :

$$(a + b)^* \subseteq (a^*b)^*a^*$$

Prendiamo una stringa così composta :



marchiamo solo dove è presente b.
La stringa è divisa in due pezzi, il secondo, non essendoci b, conterrà a per forza, non sappiamo quante sono, però possiamo dire che $\in a^*$. Vediamo il primo blocco :
dove non abbiamo b marcato avremo a quindi il primo blocco $\in (a^*b)^*$, e quindi abbiamo dimostrato che sono uguali.

Espressioni Regolari Estese :
Definiamo le espressioni regolari estese con la simbologia Rε e naturalmente per quanto detto :

$$R \subseteq R\varepsilon$$

Espressioni Star-Free :
Definiamo le espressioni regolari Star-free con la simbologia SF e saranno quelle che espressioni che usano solo : $+, ., ^C$

Esempio :
Supponiamo di avere un'espressione :

$$(ab)^* \in R$$

è possibile scriverla sotto orma di SF? O meglio : $(ab)^* \in SF$

Dimostrazione :
Possiamo scindere il problema definendo 3 linguaggi che saranno : $\Sigma = \{a, b\}$

1. Tutte le stringhe che cominciano per a, quindi avremo :
   $a \in \Sigma^* \quad \Rightarrow$ a……(qualsiasi cosa)….
   sotto forma di espressioni regolari : $a(a+b)^*$
2. Tutte le stringhe che terminano per b :
   Facendo un ragionamento analogo : $(a+b)^*b$

3. Non ci sono mai due a o due b di seguito, quindi $\nexists$ aa o bb :
   sotto forma di espressioni regolari lo possiamo scrivere : $((a+b)^* . (aa+bb)(a+b)^*)^C$
   ossia tutte le stringhe che non contengono aa o bb.

Da quanto detto possiamo scrivere la seguente uguaglianza :

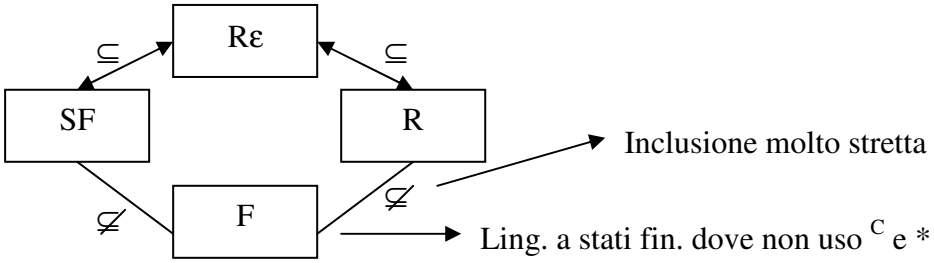
$$(ab)^* = a(a+b)^* \cap (a+b)^* b \cap ((a+b)^* . (aa+bb)(a+b)^*)^C$$

$$= \varnothing^C \quad = \varnothing^C \quad = \varnothing^C \quad = \varnothing^C$$

quindi :

$$a\,\varnothing^C \cap \varnothing^C\, b \cap (\varnothing^C\,(aa+bb)\,\varnothing^C)$$

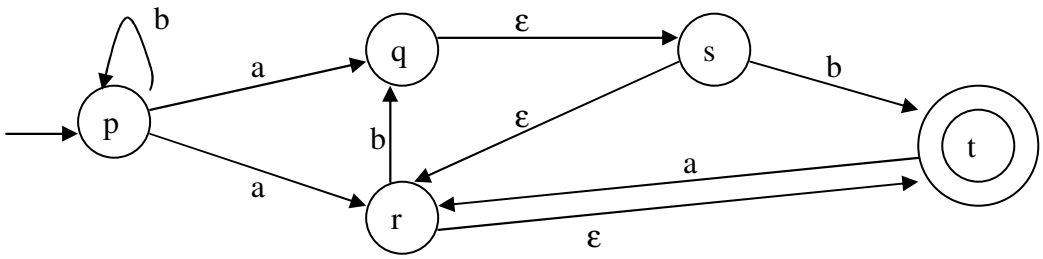
a questo punto posso definire la seguente gerarchia :



Teorema :
Fino ad ora abbiamo dimostrato che L(FSA) è chiusa rispetto alle operazioni booleane, ma ci resta da dimostrare che :

L(FSA) è chiusa rispetto a . e *

Dimostrazione :
Supponiamo di avere un automa con ε-transizioni, dove ε è la parola vuota :



Gli archi etichettati con ε sono la parola vuota. Il concetto di *stringa accettata* lo posso dare anche non leggendo ε e passando da uno stato all'altro. Se ad esempio ho :

$$a\,\varepsilon\,b = ab$$

una volta definito questo automa, me ne trovo un altro senza ε-transizioni che mi riconosce lo stesso linguaggio.

Ma come faccio a costruire questo automa ?

Definiamo il concetto di ε-chiusura, dove per ε-chiusura di uno stato q intendiamo l'insieme di tutti gli stati che posso raggiungere leggendo ε da q.
Nel nostro caso :

$$\varepsilon(q) = \{q, s, r, t\}$$

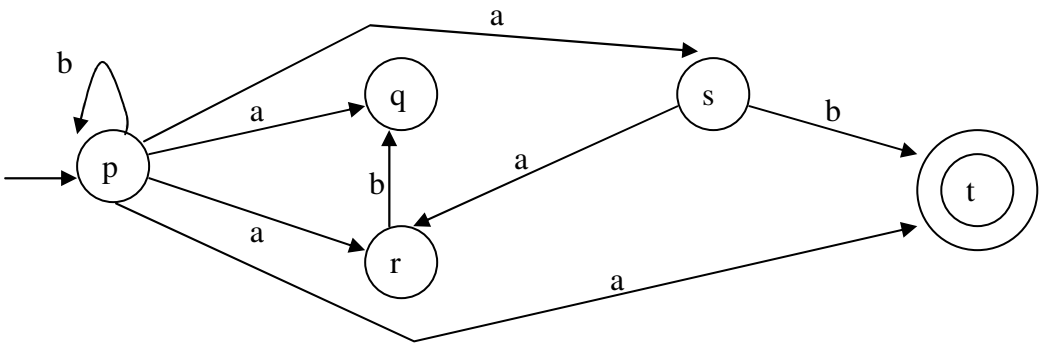
Dopo avere definito tale insieme vado a controllare tutte le alter transizioni che vanno in q.
Nel nostro caso :

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

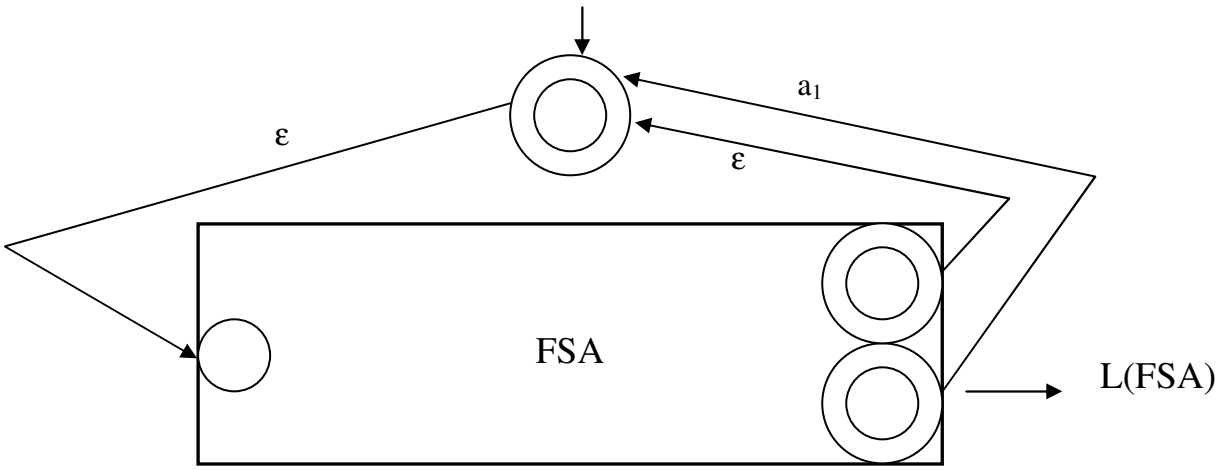
e sostituisco le ε-transizioni con quella ottenuta :

$$\delta(p, a) = q$$
$$\delta(p, a) = r$$
$$\delta(p, a) = s$$
$$\delta(p, a) = t$$

Facendo le sostituzioni suddette ottengo il seguente automa dove non compaiono più ε :

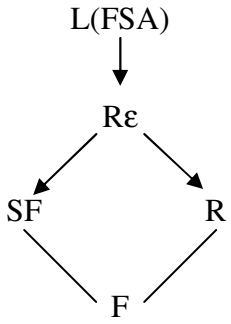


Da questo NFA ottenuto posso ricavarmi il DFA attraverso la subset construction.
Dopo il ragionamento fatto possiamo vedere la dimostrazione del teorema con la seguente figura :



Questo automa riconosce dei linguaggi finiti, se partiamo da uno stato iniziale esterno attraverso una ε-transizione e poi passiamo una stringa riconosciuta dall'automa, alla fine arriveremo a uno stato di accettazione e da lì attraverso un'altra ε-transizione ritorno al punto di partenza.

Da qui itero il procedimento e quindi posso dire che : L(FSA) è chiuso rispetto a . e *
Quindi avremo :



Lezione 18 22/11/2002

Ricapitolando quanto detto finora :

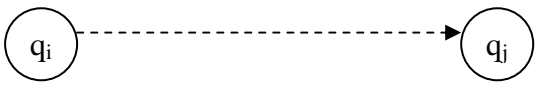
- Linguaggi elementari : $\varnothing$, ε, a con a ∈ Σ
- Operazioni regolari : +, ∩, ^{C}, . ,*

Con :

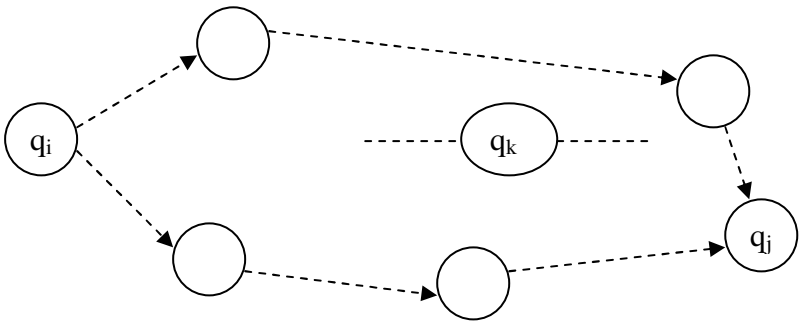
- {+, .} = F, famiglia dei linguaggi finiti
- {+, . , ^{C}} = SF famiglia dei linguaggi star-free
- {+, . , *} = R famiglia dei linguaggi regolari
- {+, . , ^{C}, *} = Rε famiglia dei linguaggi regolari estesi

Teorema di Kleene :
Se L(FSA) ⊆ R ⊆ Rε ⊆ L(FSA) allora : L(FSA) = R = Rε

Dimostrazione:
Dimostriamo che : L(FSA) ⊆ R
Quindi che un linguaggio regolare si può esprimere con un linguaggio L(FSA) , ossia con un linguaggio generato da un automa a stati finiti.
Supponiamo di avere un automa A :
A = (Σ, Q, q_0, F, δ) DFA
Q = {q_0, q_1, …,q_n}
Poi abbiamo tre interi i, j, k ≥ 0
Adesso denotiamo un linguaggio L con la seguente simbologia : L^K_{ij}
Definisco tale linguaggio come segue :



Parto dallo stato q_i e devo arrivare allo stato q_j passando per stati con indice minore di k

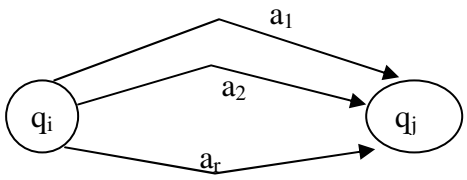


L^K_{ij} = insieme delle stringhe che corrispondono a cammini da q_i a q_j tali che gli stati intermedi hanno indice minore di k.
Una volta definito tale linguaggio dimostro che :

$$L^K_{ij} \in R$$

Per induzione, se

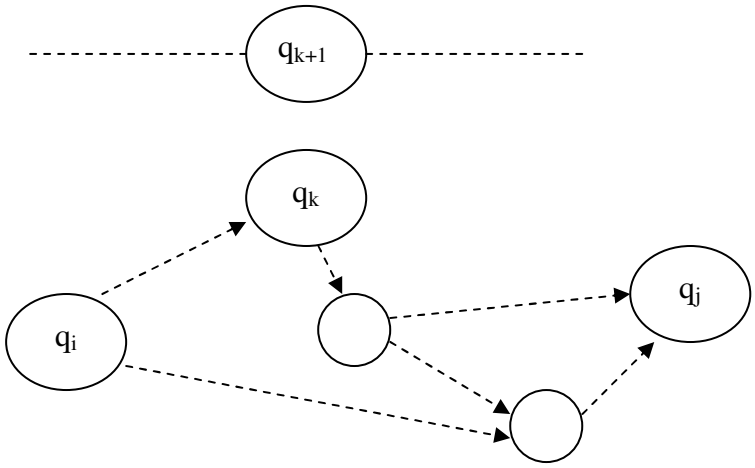
- $K = 0 \Rightarrow L^0_{ij} \Rightarrow \forall$ i, j non ci sono stati intermedi con indice minore di k



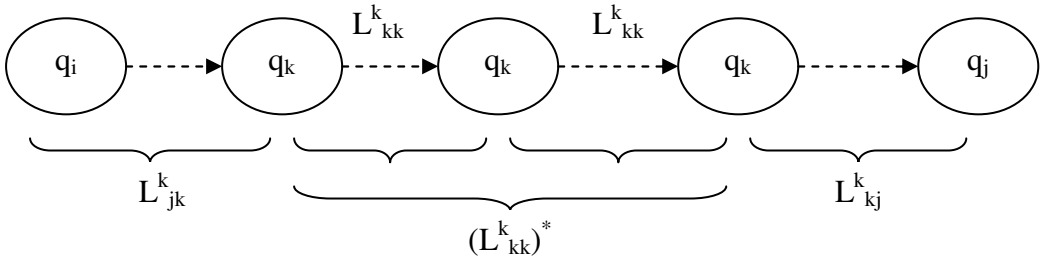
Quindi deduciamo che :

$L^0_{ij} = a_1 + a_2 + \ldots + a_r$ (espressione regolare)

Adesso supponiamo che sia vero per un valore k, se è vero per k+1 allora la dimostrazione è completa.
Consideriamo L^{k+1}_{ij}
Dobbiamo partire da uno stato q_i e arrivare a uno stato q_j passando per stati che hanno indice minore di k+1 :



Il cammino che va da q_i a q_j passa più volte da k :



Quindi possiamo benissimo dire che :

$$L^{k+1}_{ij} = L^{k}_{ij} \cup L^{k}_{ik} (L^{k}_{kk})^{*} L^{k}_{kj}$$

(R) (R) (R) (R)

$L(A) = \cup_{q_t \in F} L^{n+1}_{0t} \in R$ $\qquad$ $R \subseteq L(FSA)$

Quindi : $\underbrace{(a + (ba)^{*}b)ba}_{\text{Linguaggio regolare}} \longrightarrow$ A (automa FSA)

Quindi : **FSA** $\longleftrightarrow$ espressione regolare

Espressione regolare estesa

Lezione 19 25/11/2002

Abbiamo visto che per il teorema di Kleene

$$L \in L(\text{FSA}) = R$$

Ossia che uno stesso linguaggio può essere espresso o con un automa o con un' espressione regolare E.
Quando definiamo un linguaggio con un espressione regolare non abbiamo un algoritmo preciso, quindi è utile descriverne uno per passare da un'espressione regolare ad un automa.

$$E \longrightarrow A$$

Algoritmo di Berry e Sethi
Per capire meglio tale algoritmo bisogna introdurre alcuni concetti come quello di linguaggio locale

-Linguaggio Locale
Un linguaggio locale è un linguaggio dove la proprietà sono definite in maniera locale.
Per definirlo mi serve una quadrupla:

$$L(\lambda, P, S, W)$$

Dove:

- λ : è l'insieme $\{ \varepsilon, \varnothing \}$, ci dice se è presente o no la parola vuota.
- P : è un sottoinsieme dell'alfabeto $\Sigma \Rightarrow P \subseteq \Sigma$ simboli con il quale può iniziare una stringa
- S : è un sottoinsieme dell'alfabeto $\Sigma \Rightarrow S \subseteq \Sigma$ simboli con il quale può finire una stringa
- W : è un sottoinsieme di $\Sigma^2 \Rightarrow W \subseteq \Sigma^2$ definisce quali coppie di simboli possono occorrere in una stringa.

Esempio: se aa è proibito W={ ab,ac,bc,…..} su un alfabeto Σ={a,b,c}

Esempio :
Se abbiamo un alfabeto Σ= {a,b,c} definiamo la quadrupla di un linguaggio locale L come segue:
P = {a, b ,c};

S = { b, c};
w ={non ci può essere ab };
λ= { $\varnothing$}$\rightarrow$ non è presente la parola vuota
questa quadrupla definisce un linguaggio L che accetta le stringhe che iniziano per a, b, c, che finiscono con b,c; che non contengono a seguita da b; e non hanno la parola vuota.
Vediamo cosa succede se facciamo delle operazioni sui linguaggi:

Esempio :
Se ho due linguaggi locali cosi definite: Σ= {a, b}

$L_a \left\{ \begin{array}{l} \lambda = \varnothing \\ P = S = \{a\} \\ W = \Sigma^2 \end{array} \right\}$ $L_b \left\{ \begin{array}{l} \lambda = \varnothing \\ P = S = \{b\} \\ W = \Sigma^2 \end{array} \right\}$

Se facciamo l'unione avremo che :

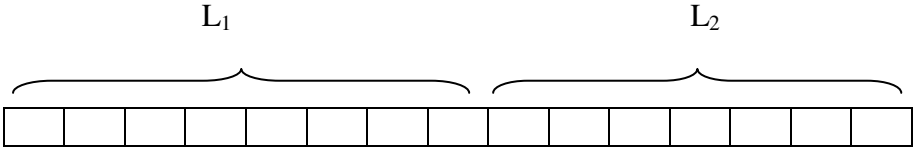
$$L_a \cup L_b = L_{ab}$$

Infatti :

$L_{ab} \left\{ \begin{array}{l} \lambda = \varnothing \\ P = S = \{a, b\} \\ W = \Sigma^2 \end{array} \right\}$

Secondo quanto detto non ho più l'indipendenza delle osservazioni definite dalla quadrupla; infatti se prendo L_a la stringa deve iniziare per a e finire per a, e se prendo L_b lo stesso (con b ovviamente); quindi posso dire che una stringa deve avere l'ultima lettera uguale alla prima.
Da quanto detto possiamo fare la seguente osservazione : *la concatenazione di due linguaggi L_1 e L_2 che sono locali non è più un linguaggio locale*.
Se prendo però due linguaggi locali L_1 e L_2 che hanno alfabeti disgiunti se applico le operazioni suddette ottengo ancora un linguaggio locale.

Esempio:
$L_1 = (\lambda_1, P_1, S_1, W_1)$ dove $L_1 \supseteq \Sigma_1^*$
$L_2 = (\lambda_2, P_2, S_2, W_2)$ dove $L_2 \supseteq \Sigma_2^*$
e dove $\Sigma_1^* \cap \Sigma_2^* = \varnothing$
l'unione sarà:
$L_{1/2} = (\lambda_1 \cup \lambda_2, P_1 \cup P_2, S_1 \cup S_2, W_1 \cup W_2)$
Questo linguaggio sarà ancora locale.



Vediamo la concatenazione dove:

$$L_1 \cdot L_2 = (\lambda, P, S, W)$$

-$\lambda = \lambda_1 \cap \lambda_2 = \lambda_1 \cdot \lambda_2$, se $\lambda_1 = \varepsilon$ e $\lambda_2 = \varepsilon \Rightarrow \lambda = \varepsilon$
-$P = P_1 \cup \lambda_1 P_2$ perché:

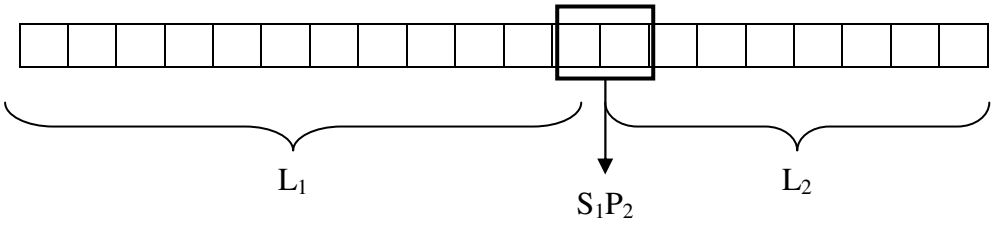
$\begin{cases} \text{se } \lambda_1 = \varnothing \Rightarrow \lambda_1 P_2 = \varnothing \Rightarrow \text{La stringa deve iniziare con } P_1 \\ \text{se } \lambda_1 = \varepsilon \Rightarrow \lambda_1 P_2 = P_2 \Rightarrow \text{La stringa può iniziare con } P_2 \end{cases}$

-$S = S_2 \cup \lambda_2 S_1$ perché:

$\begin{cases} \text{se } \lambda_2 = \varnothing \Rightarrow \lambda_1 S_1 = \varnothing \Rightarrow \text{La stringa deve finire con } S_2 \\ \text{se } \lambda_2 = \varepsilon \Rightarrow \lambda_2 S_1 = S_1 \Rightarrow \text{La stringa può finire con } S_1 \end{cases}$

-$W = W_1 \cup W_2 \cup S_1 P_2$

Quindi, permessi del primo linguaggio + permessi del secondo linguaggio + permessi della combinazione che può capitare tra la fine della prima stringa e l'inizio della seconda.

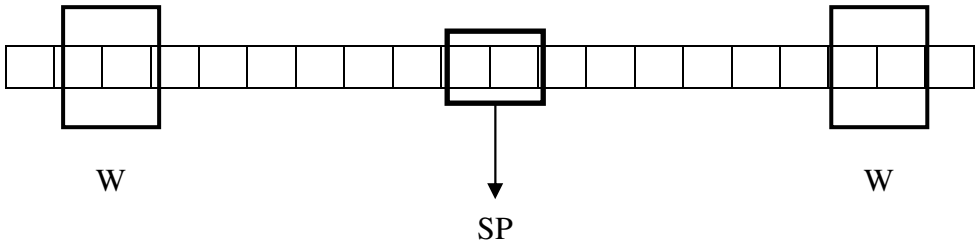


Star :

$$L(\lambda, P, S, W)$$

Se facciamo L^* :

- $\lambda = \varepsilon$ indipendentemente dal valore di λ, perché $\{\varepsilon\}$ appartiene all'operazione *star*, e quindi possiamo avere la parola vuota.
- P è uguale, perché la prima stringa deve iniziare sempre per **P**.
- S è uguale, perché l'ultima stringa deve finire sempre per **S**.
- $W = W \cup SP$; ossia i permessi che già avevamo + i permessi di ogni fine stringa ed inizio dell'altra.



Quindi possiamo dire che i linguaggi locali sono chiusi rispetto a tutte le operazioni, con l'accorgimento di prendere due alfabeti disgiunti, tranne per *, ovviamente, perché avremo sempre lo stesso alfabeto.

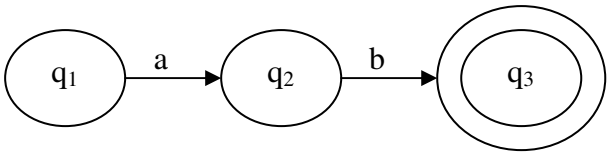
Automi locali :
un Automa DFA è locale se gode di determinate proprietà:

1. Nello stato iniziale non entra alcun arco
2. Tutti gli archi con la stessa etichetta entrano nello stesso stato.

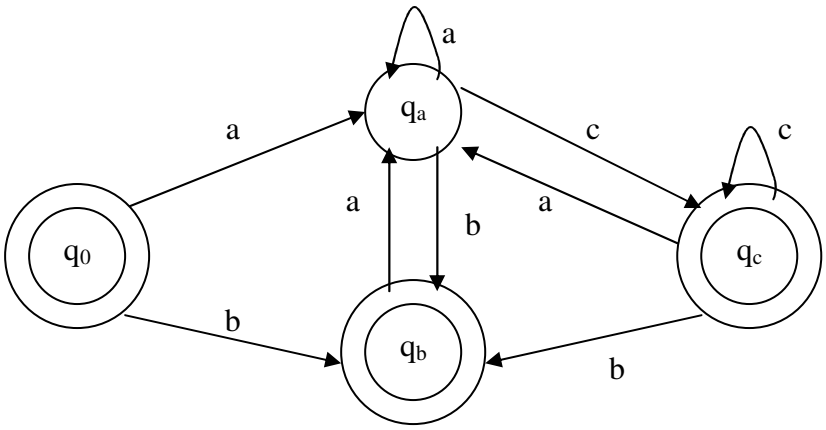
Un automa locale è un FSA.
Possiamo dire che :

$$\text{se } |\Sigma| = n \qquad \Rightarrow \qquad |Q| = n + 1$$

Esempio :
Se $\Sigma = \{a, b\} = 2$; $|Q| = 3$, se l'automa è completo, ci saranno almeno n + 1 stati, infatti :



Ritornando al discorso di prima, avremo un automa di questo tipo :



Teorema :
Ogni Automa locale riconosce un linguaggio locale e viceversa.

Dimostrazione:
Supponiamo di avere un linguaggio locale L dove:

$$L\ (\lambda, P, S, W)$$

Dove:

- $\lambda = \varepsilon$ perché q_0 è uno stato di accettazione
- $P = \{a, b\}$
- $S = \{b, c\}$
- W= insieme degli archi entranti combinati con quelli uscenti, quindi :

$$w=\{\underbrace{aa, ab, ac,}_{\text{da } q_a} \quad \underbrace{ba,}_{\text{da } q_b} \quad \underbrace{cc, ca, cb}_{\text{da } q_c}\}$$

e il linguaggio riconosciuto da tale automa è locale.
Adesso facciamo il viceversa, partiamo da un linguaggio locale L e costruiamo l'automa corrispondente.

$$L(\lambda, P, S, W)$$

La dimostrazione è costruttiva…..

Teorema:
I linguaggi elementari sono locali. Infatti se:

- $\varnothing \Rightarrow L = (\varnothing, \varnothing, \varnothing, \varnothing)$
- $\varepsilon \Rightarrow L = (\varepsilon, \varnothing, \varnothing, \varnothing)$
- $a \in \Sigma \Rightarrow L = (\varnothing, a, a, \varnothing)$

prima di passare all'algoritmo di Berry e Sethi dobbiamo definire le *espressioni regolari lineari*

Espressioni regolari lineari:
Un' *espressione regolare lineare,* è un'espressione dove ogni simbolo dell'alfabeto Σ compare al più una sola volta.
Quindi:

$(a + ba)^*$ $\Rightarrow$ non lineare
$(ab^*+c)^*d^*$ $\Rightarrow$ lineare

Teorema:
Ogni espressione regolare lineare rappresenta un linguaggio locale.

Dimostrazione:
Partendo dall'espressione regolare lineare: $(ab^*+c)^*d^*$ sappiamo per quanto detto sui linguaggi elementari:

$b \Rightarrow L = (\varnothing, b, b, \varnothing)$
$b^* \Rightarrow L = (\varepsilon, b, b, bb)$
$a \Rightarrow L = (\varnothing, a, a, \varnothing)$
$ab^* \Rightarrow L = (\varnothing, a, \{a, b\}, \{bb, ab\})$
….
…
…
etc.

quindi partendo da un'espressione regolare lineare possiamo definirci un linguaggio locale.

Esempio: (come linearizzare un espressione?)
$E = (b + a^*b)^*$
Ci creiamo una tabella di conversione dove:

$a_1 = b$
$a_2 = a$
$a_3 = b$

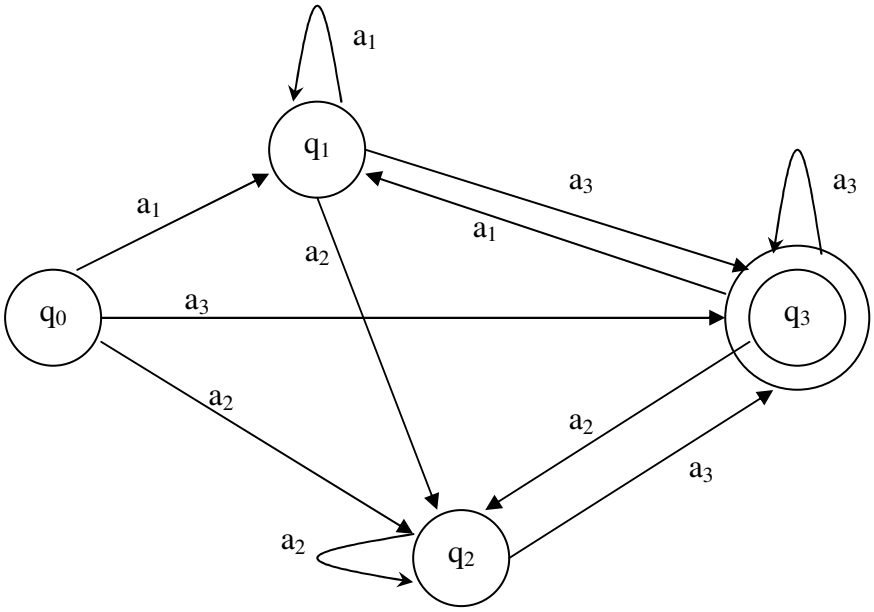
adesso sostituisco all'espressione e ottengo:

$$(a_1 + a_2^* a_3)^*$$

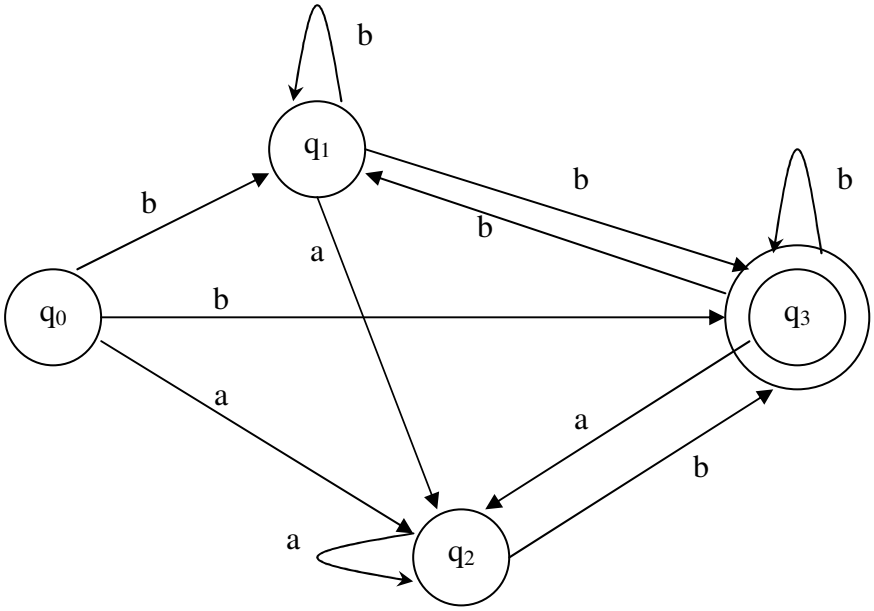
dai linguaggi elementari sappiamo che:

$a_1 \Rightarrow L_1 = (\varnothing, \{a_1\}, \{a_1\}, \varnothing)$
$a_2^* \Rightarrow L_{a2} = (\varepsilon, \{a_2\}, \{a_2\}, \{aa\})$
$a_3 \Rightarrow L_{a3} = (\varnothing, \{a_3\}, \{a_3\}, \varnothing)$
$a_2^* a_3 \Rightarrow L_2 = (\varnothing, \{a_2, a_3\}, \{a_3\}, \{a_2a_2, a_2a_3\})$
$(a_1 + a_2^* a_3) \Rightarrow L = L_1 \cup L_2 = (\varnothing, \{a_1, a_2, a_3\}, \{a_1, a_3\}, \}, \{a_2a_2, a_2a_3\})$
$(a_1 + a_2^* a_3)^* \Rightarrow L^* = (\{\varepsilon\}, \{a_1, a_2, a_3\}, \{a_1, a_3\}, \}, \{a_2a_2, a_2a_3, a_1a_1, a_1a_2, a_1a_3, a_3a_1, a_3a_2, a_3a_3\})$

dal linguaggio L ottenuto ci ricaviamo l'automa locale:



Da questo automa usando la tabella di conversione mi ricavo l' NFA.



A questo punto abbiamo concluso il nostro algoritmo passando da un'espressione regolare a un automa a stati finiti FSA.
Altresì abbiamo visto che:

$$\text{Linguaggi Locali} \subseteq SF \subseteq R = RE = L\ (FSA)$$

Quindi star-free è inclusa propriamente.

Se abbiamo un linguaggio locale L definito da una quadrupla :

$$L\ (\lambda, P, S, W)$$

Lo possiamo definire con le operazioni di (+, . , ^{C}, *) in questo modo :

$$L = \lambda + (P\Sigma^* \cap \Sigma^* S \cap (\Sigma^* F \Sigma^*)^C)$$

Dove :

- $P\Sigma^* = xv$, dove $x \in P$ e v è una stringa arbitraria;
- $\Sigma^* S = vy$, dove v è una stringa arbitraria e $y \in S$;
- $\Sigma^* F \Sigma^* = ?$

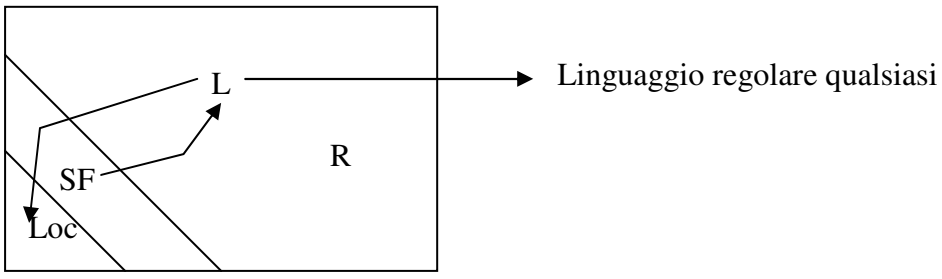
quindi :

- $P_+ = \{x_1, x_2, \ldots, x_k\}$ con $x_i \in \Sigma^+$
- Se $S = \{y_1, y_2, \ldots, y_h\}$ con $y_i \in \Sigma$
- Se $F = \{z_1t_1, z_2t_2, \ldots, z_rt_r\}$ con $z_it_i \in \Sigma$ (le coppie ordinate)

Allora :

$$L = \varepsilon + (x_1, x_2, \ldots, x_k)\, \varnothing^C \cap \varnothing^C\, (y_1, y_2, \ldots, y_h) \cap (\varnothing^C(z_1t_1, z_2t_2, \ldots, z_rt_r)\, \varnothing^C)^C$$

che è una star-free.

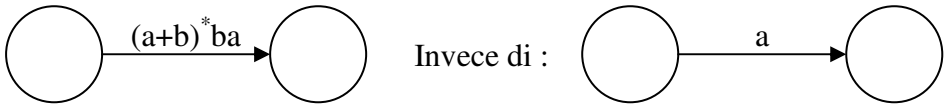


Quindi da un linguaggio regolare mi posso ricavare l'automa locale e viceversa.
Adesso supponiamo di avere un automa A e di volere passare a un'espressione regolare :

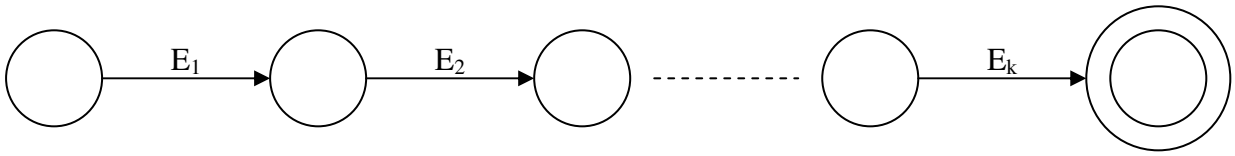
$$A \rightarrow E$$

Per vedere questo concetto dobbiamo prima dare un concetto di automa più generale.
Quando andiamo a rappresentare l'automa mediante grafo invece di mettere nell'etichetta dell'arco la lettera mettiamo un'espressione regolare.

Esempio :



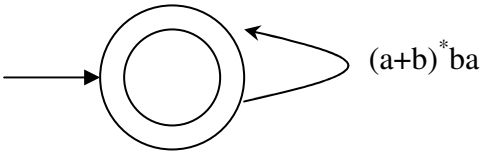
Quindi avremo un automa di questo tipo :



Se scrivo :

$E_1E_2\ldots E_k$ = Linguaggio riconosciuto dall'automa

Esempio :
Se ho un automa A così rappresentato :

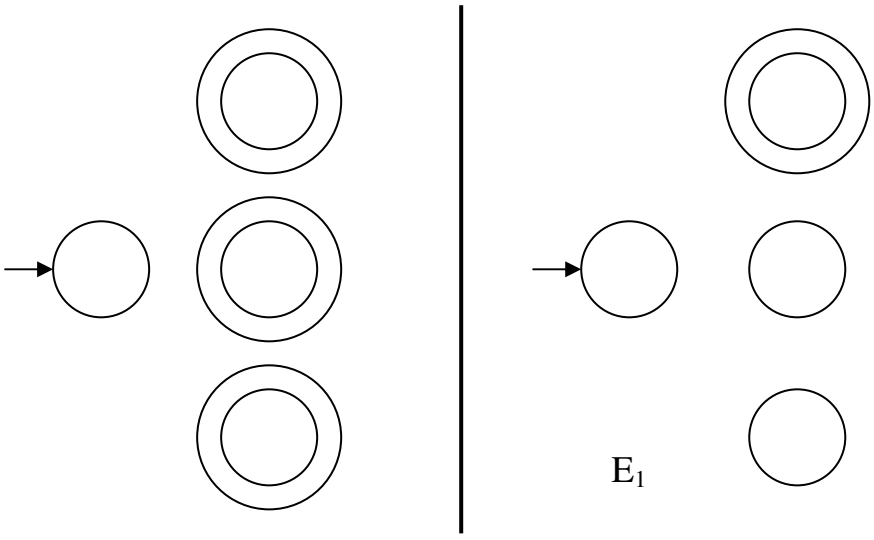


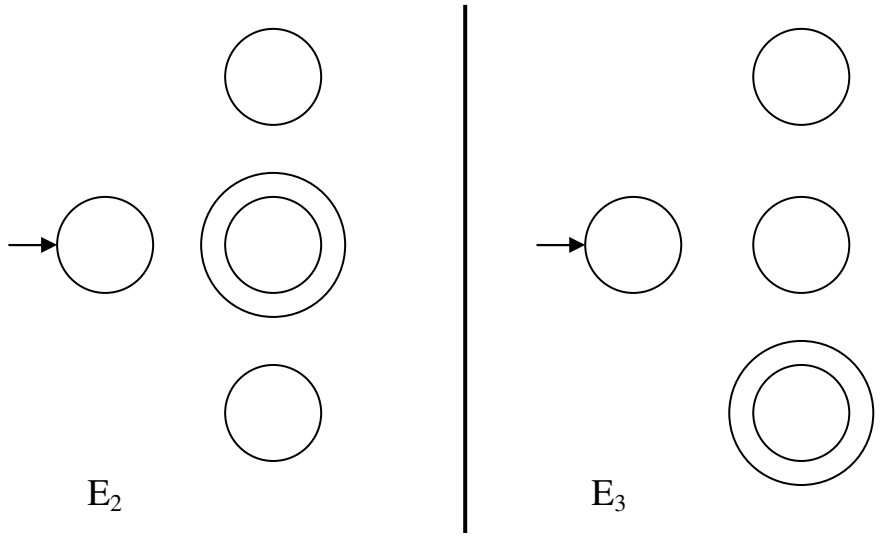
Chi è il linguaggio riconosciuto da questo automa ?

$$L(A) = (a+b)^*ba$$

L'algoritmo per determinare tale automa consiste in due fasi :

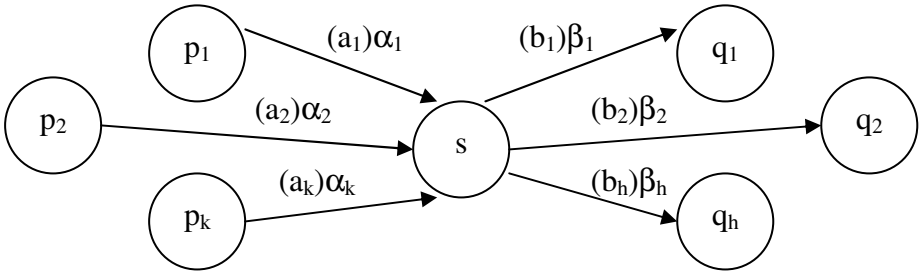
- **1° Fase :**
  Parto da un automa DFA che ha uno stato di partenza e uno stato di accettazione :
  Ad esempio se esempio se disegno un DFA in questo modo :





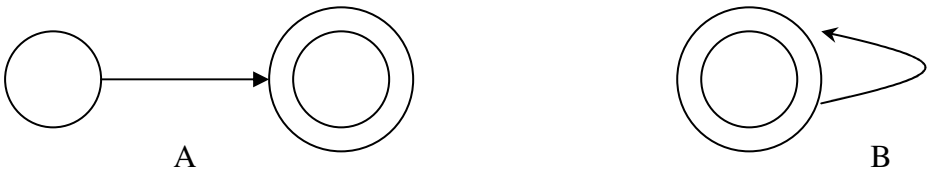
Il linguaggio di questo automa è dato da dall'unione di queste tre coppie di automa che mi sono costruito.
A questo punto se so ricavare E_1, E_2, E_3 ho trovato un'espressione regolare per l'automa iniziale che sarà : $E = E_1 + E_2 + E_3$

- **2° Fase :**
  Eliminazione degli stati :



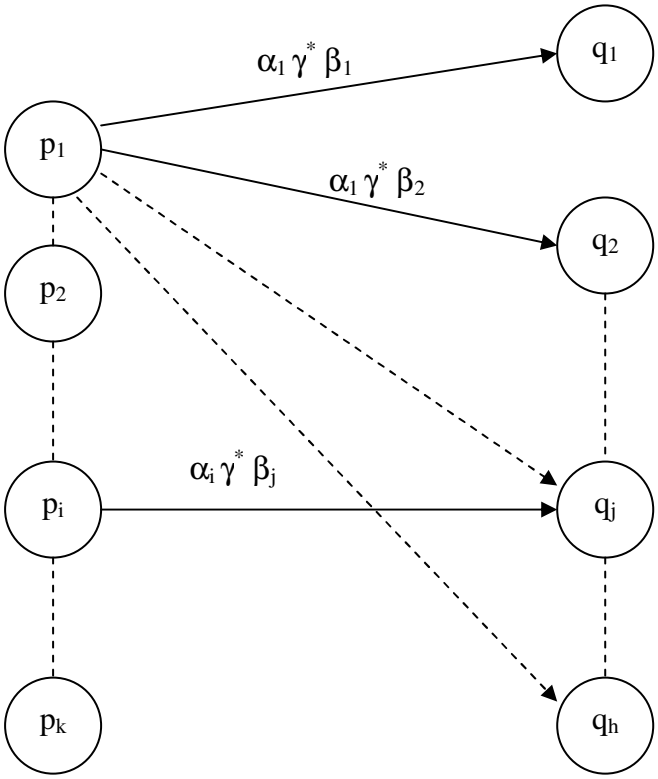
- ✓ L'insieme formato dalle mie p_i è l'insieme degli stati in cui ci sono degli archi che entrano in s.
- ✓ L'insieme formato dalle mie q_i è l'insieme degli stati in cui ci sono degli archi che entrano e che arrivano da s.

Se sostituisco **a** e **b** con **α** e **β** dove α e β sono espressioni regolari; alla fine eliminando opportunamente, non considerando lo stato iniziale e di accettazione otterrò :



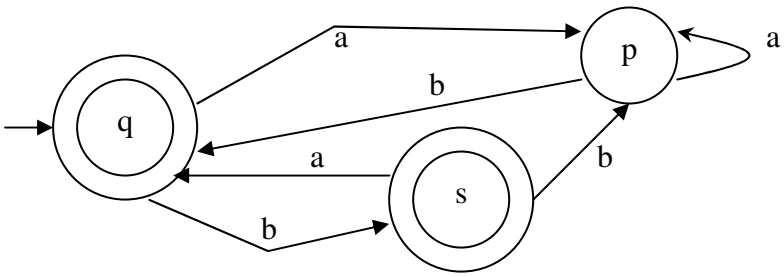
A. Se lo stato di accettazione e lo stato iniziale sono disgiunti;
B. Se lo stato di accettazione e lo stato iniziale sono uguali.

Alla fine avrò :



Esempio :
Supponiamo di avere un automa A così definito :

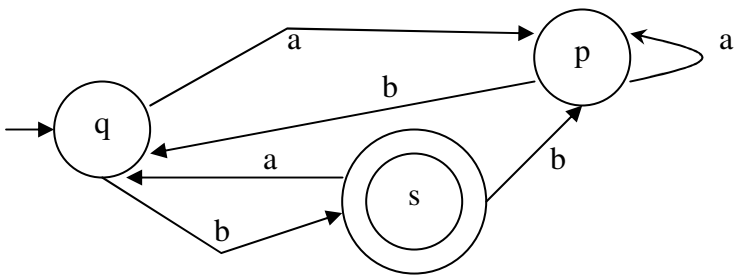
A :



Fase 1 :

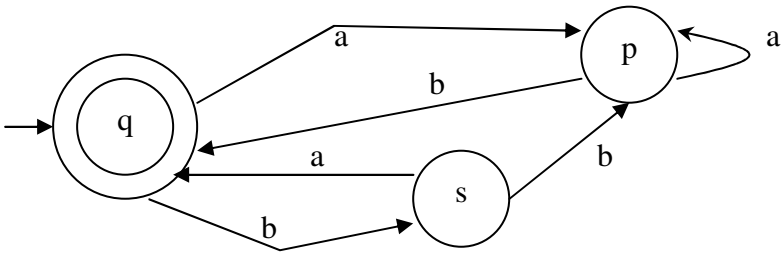
La prima copia sarà :

A_1 :



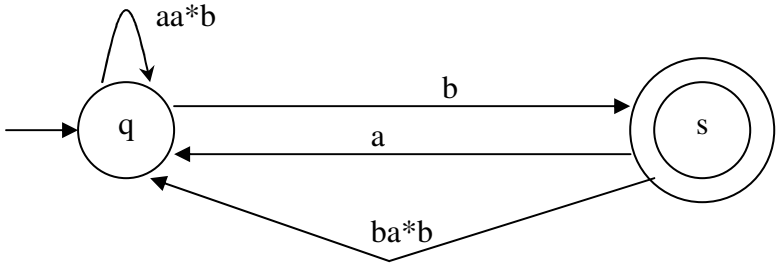
La seconda copia sarà :

A_2 :

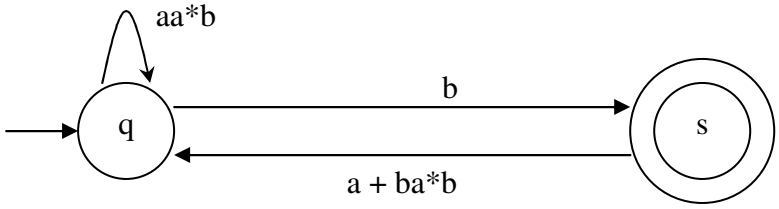


Fase II per A_1 :

Il mio **s** sarà **p**, parto da **q** stato iniziale dove :



Che posso scrivere ancora come :

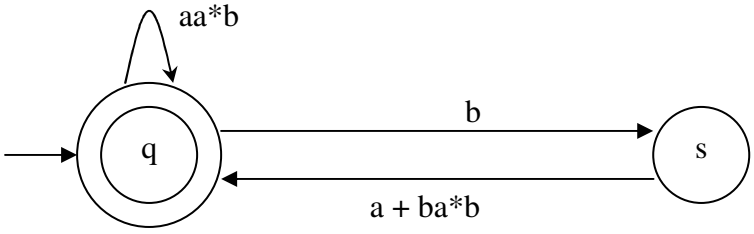


Da cui otteniamo $\mathbf{E_1}$ dove :

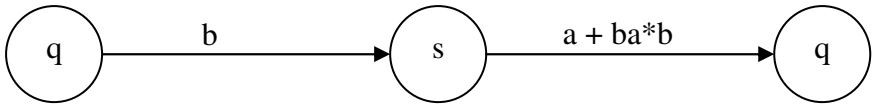
$$E_1 = (aa^*b + b(a + ba^*b))^* b = L(A_1)$$

Fase 2 per A_2 :

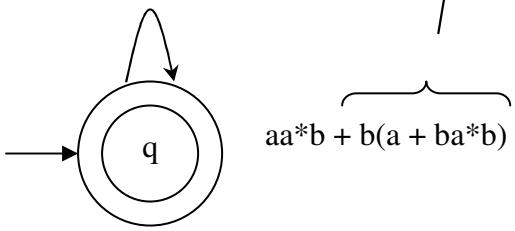
Il mio **s** sarà sempre **p**, parto da **q** stato iniziale dove :



Da cui iteriamo ottenendo :



Adesso faccio :



Da cui:

$$E_2 = (aa^*b + b(a + ba^*b))^*$$

A questo punto avremo che:

$$E = E_1 + E_2 = (aa^*b + b(a + ba^*b))^* b + (aa^*b + b(a + ba^*b))^* = (aa^*b + b(a + ba^*b))^* (b + \varepsilon)$$

Lezione 18 29/11/2003

Abbiamo visto diversi modelli di riscrittura definiti nel seguente modo:

$$P \{u \rightarrow v\}$$

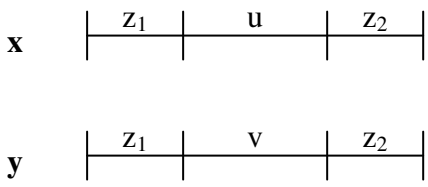
Presi $x, y \in \Sigma^*$ si definisce la relazione :

$$x \Rightarrow y$$

che si legge : “da x si deriva y in un passo”

se $x = z_1 u z_2$
se $y = z_1 v z_2$
se $u \rightarrow v \in P$

e quindi avremo :



Da cui si ricava l’altra relazione :

$$x \overset{*}{\Rightarrow} y \text{ (da x si deriva y)}$$

se esiste una successione finita di parole $x_1, x_2, \ldots, x_n$ tali che :

- $x_1 = x$
- $x_n = y$
- $\forall$ i con $(1 \leq i < n)$ $\quad x_i \Rightarrow x_{i+1}$

ossia $\forall$ x_i si deriva x_{i+1}. $\overset{*}{\Rightarrow}$
In termini più formali la $\Rightarrow$ è la chiusura riflessiva e transitiva della $\Rightarrow$.

Definizione :
Definiamo adesso le grammatiche :

$$G = (\Sigma, V, P, \Omega)$$

- **Σ** = alfabeto di simboli terminali
- **V** = alfabeto di simboli non terminali
- **P** = insieme finito di regole di produzione dove :
  $P \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$
- **Ω** = assioma, dove $\Omega \in V$

Per convenzione avremo che :

- **Σ** = {a, b, c, …} $\rightarrow$ caratteri minuscoli
- **V** = {A, B, C, …} $\rightarrow$ caratteri maiuscoli
- Parole di Σ^* $\rightarrow$ caratteri minuscoli e si usano le ultime lettere dell'alfabeto, quindi u, v, w, x, …
- Parole di $(\Sigma \cup V)^*$ $\rightarrow$ si usano lettere dell'alfabeto Greco, quindi α, β, γ

Linguaggio generato da una grammatica :
Il linguaggio generato da una grammatica viene così definito :

$$L(G) = \{w \in \Sigma^* / \Omega \overset{*}{\Rightarrow} w\}$$

Esempio :
Date le regole di produzione : $\Sigma = \{a, b\}$

1° = $\Omega \rightarrow \Omega\Omega$
2° = $\Omega \rightarrow a\Omega b$
3° = $\Omega \rightarrow ab$

Consideriamo un esempio di derivazione :

$\Omega \Rightarrow \Omega\Omega \Rightarrow a\Omega b\Omega \Rightarrow a\Omega\Omega b\Omega \Rightarrow aab\Omega b\Omega \Rightarrow aababb\Omega \Rightarrow aababbab \in L(G)$

I linguaggi che genero con le grammatiche sono riconosciuti dalle macchine di Turing, e più in particolare :

AUTOMI		GRAMMATICHE
MT	⟷	Tipo 0
LBA	⟷	Tipo 1
PDA	⟷	Tipo 2
FSA	⟷	Tipo 3

Grammatiche di Chomski :
Chomsky introdusse delle grammatiche di tipo particolare, imponendo che le regole fossero di una certa forma :

- **Tipo 0 :**
  Quella definita in generale;

- **Tipo 1 :**
  In ogni derivazione l'elemento che sta a destra deve essere maggiore o uguale a quello che sta a sinistra : $|\alpha| \leq |\beta|$

  In questo caso **MP** è decidibile, infatti : $\Omega \overset{1}{\Rightarrow} w_1 \Rightarrow w_2 \ldots\ldots\ldots \Rightarrow w_n = w$
  dove : $1 \leq |w_i| \leq |w|$; il numero di stringhe sarà limitato, infatti se :
  $|w| = k$; e $|\Sigma| = d$; le stringhe sono : $d^k + d^{k-1}$ ……d

- **Tipo 2 :**
  Le regole di produzione sono : $A \rightarrow \alpha$ con $\alpha \neq \varepsilon$

- **Tipo 3 :**
  Sono un caso particolare del Tipo 2 dove le regole sono :

  $\begin{cases} A \rightarrow a \\ A \rightarrow aB \end{cases}$ O $\begin{cases} A \rightarrow a \\ A \rightarrow Ba \end{cases}$

  Detta *Lineare a Destra* Detta *Lineare a Sinistra*

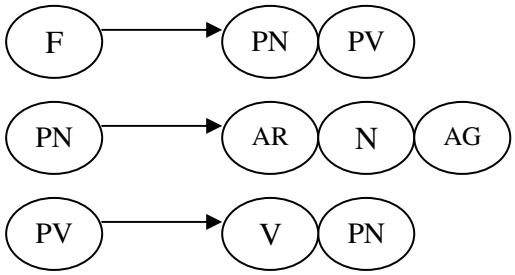
  Se mescolo ottengo grammatiche che non appartengono a questi tipi di linguaggi.

Lezione 19 2/12/2002

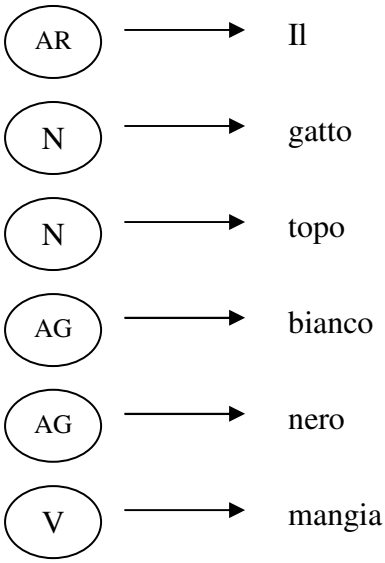
Esempio :

Supponiamo di avere un linguaggio naturale che segue le seguenti regole di produzione dove :

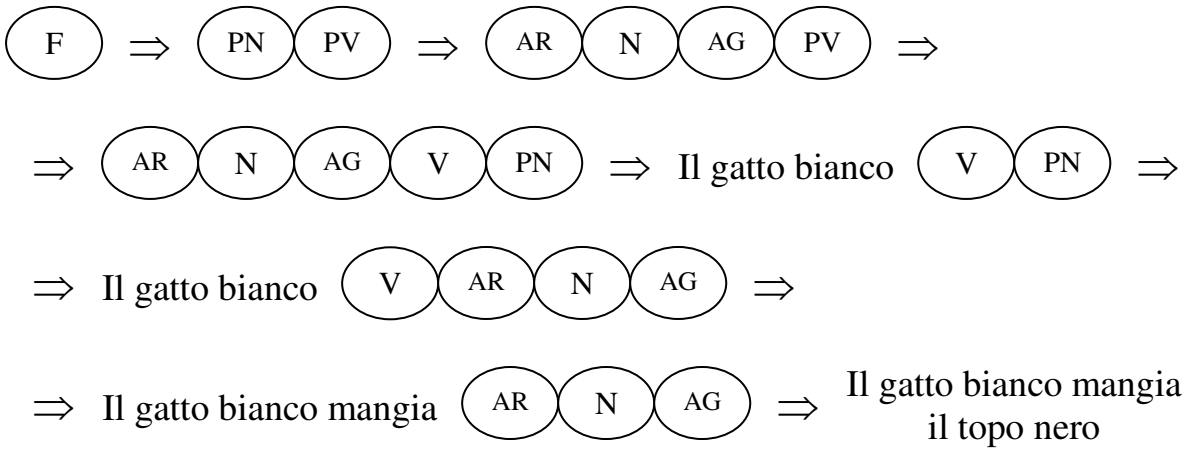
F = Frase
PN = Parte nominale
PV = Parte verbale
AR = Articolo
N = Nome
AG = Aggettivo
V = Verbo



Poi abbiamo :



F sarà l'assioma quindi :



Grammatiche Context-Sensitive (CS) :
Una grammatica si dice Context-Sensitive (CS) quando :

$$A \to \alpha \qquad e \qquad A \to \beta$$

Cioè :

$$\gamma_1 A \gamma_2 \to \gamma_1 \alpha \gamma_2$$
oppure
$$\varphi_1 A \varphi_2 \to \varphi_1 \beta \varphi_2$$

ossia quando cambia a seconda di ciò che c'è prima e dopo.
Le grammatiche **CS** sono un caso particolare del tipo 1.

Esempio :
Supponiamo di volere passare da una grammatica di tipo 1 a una grammatica CS :

se $AB \to CDE$ } grammatica di tipo 1

allora :

$AB \to CB$
$CB \to C\underline{DE}$ } Grammatica di tipo CS

Adesso ricordiamo che il linguaggio delle parentesi era riconosciuto da una PDA ed era così definito :

(( ) ( )) ( ) dove { (= a;) = b };
a a b a b b a b

Definiamo la grammatica :

$$G = \{ [(,)], [\Omega], P, \Omega \}$$

Adesso diamo una definizione ricorsiva di tale linguaggio :

Definizione :
- Data un'espressione corretta di parentesi, se la faccio precedere da (, e seguita da), ottengo ancora un'espressione corretta di parentesi.
- Se concateno due espressioni corrette di parentesi, ottengo un'espressione corretta di parentesi.
- ( ) è un'espressione corretta di parentesi.

Quindi diremo che :

$$\Omega = \text{Espressione corretta di parentesi}$$

e diamo le seguenti regole di produzione :

1° : $\Omega \rightarrow a\Omega b$
2° : $\Omega \rightarrow \Omega\Omega$
3° : $\Omega \rightarrow ab$

una derivazione sarà :

$\Omega \Rightarrow \Omega\Omega \Rightarrow a\Omega b\Omega \Rightarrow a\Omega\Omega b\Omega \Rightarrow aab\Omega b\Omega \Rightarrow aababb\Omega \Rightarrow aababbab \in L(G)$

Il linguaggio delle palindrome pari :
$L = \{vv^R / v \in \Sigma^*\}$ dove $\Sigma = \{a, b\}$; e se v = abab , la riverse v^R = baba

Se faccio riferimento alla definizione ricorsiva :

- Data un'espressione corretta di lunghezza n, aggiungendo sia prima che dopo una lettera uguale ottengo un'espressione corretta di lunghezza n+2
- aa o bb è un'espressione corretta.

Adesso definisco :

Ω = Espressione corretta di parentesi

e do le seguenti regole di produzione seguendo la definizione ricorsiva :

1° : $\Omega \rightarrow a\Omega a$
2° : $\Omega \rightarrow b\Omega b$
3° : $\Omega \rightarrow aa$
4° : $\Omega \rightarrow bb$

una possibile derivazione sarà :

$\Omega \Rightarrow a\Omega a \Rightarrow ab\Omega ba \Rightarrow abb\Omega bba \Rightarrow$ ……

Il linguaggio delle espressioni aritmetiche :
Se abbiamo : $\Sigma = \{x, y, z, +, *\}$ con x + y * z… → espressione aritmetica

Diamo le regole di produzione :

1° : $\Omega \rightarrow \Omega + \Omega$
2° : $\Omega \rightarrow \Omega * \Omega$
3° : $\Omega \rightarrow x$
4° : $\Omega \rightarrow y$
5° : $\Omega \rightarrow z$

una possibile derivazione sarà :

$\Omega \Rightarrow \Omega + \Omega \Rightarrow x + \Omega \Rightarrow x + \Omega * \Omega \Rightarrow x + y * \Omega \Rightarrow x + y * z$

nel BNF avrò la seguente notazione :

1. <espressione aritmetica> ::= <espressione aritmetica> + <espressione aritmetica>
2. <espressione aritmetica> ::= <espressione aritmetica> * <espressione aritmetica>

Albero di derivazione :
Data una grammatica :

$$G = (\Sigma, V, P, \Omega) \qquad \text{CF o tipo 2}$$

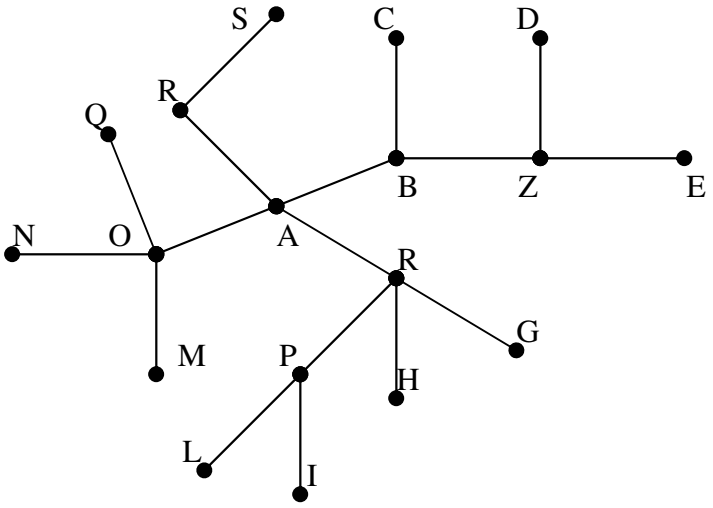
Adesso vediamo cos'è l'*albero sintattico* di una stringa $w \in L(G)$ relativo a una grammatica G. Sappiamo che quando partiamo dall'assioma attraverso la derivazione arrivo ad una espressione terminale :

$$\Omega \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots\ldots\ldots \Rightarrow w$$

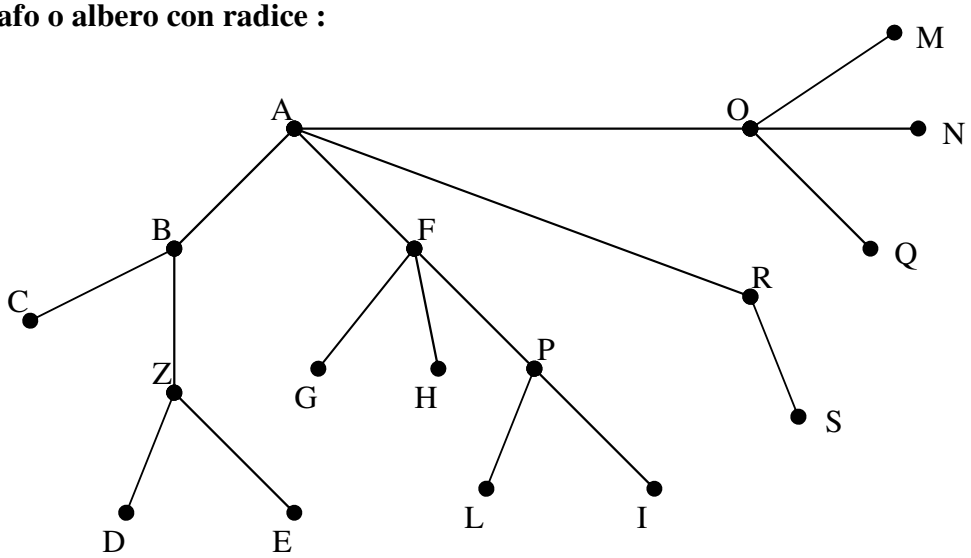
Adesso vediamo la descrizione dei vari tipi di alberi :

Alberi :

1. **Grafo non orientato senza cicli e connesso :**



2. **Grafo o albero con radice :**



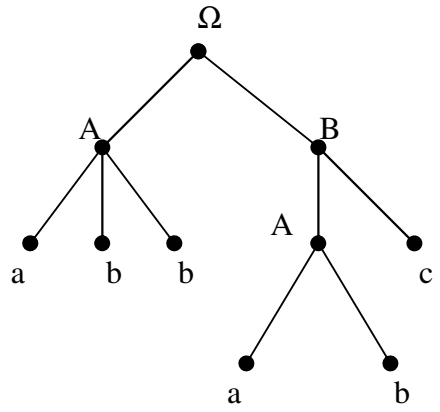
ad esempio diciamo che :

3. **Albero con radice ordinato**, nell'esempio precedente i due alberi sono diversi;
4. **Albero k-ario;**
5. **Albero orientato etichettato :**
   5.1. Tutti i nodi interni hanno come etichette un elemento di v;
   5.2. La radice ha come etichetta l'assioma Ω;
   5.3. Le foglie hanno come etichetta elementi di Σ;
   5.4. Se il nodo con l'etichetta A ha k figli con etichette $x_1, x_2, \ldots, x_k$ con $x_i \in (V \cup \Sigma)$, allora $A \rightarrow x_1, \ldots, x_k$ è una regola di P

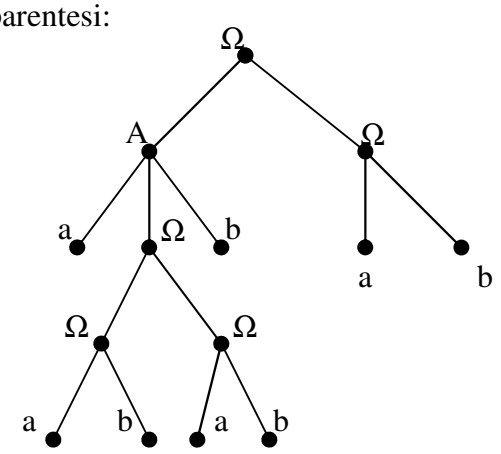
Esempio :
Date le seguenti regole di produzione :

$$\Omega \rightarrow AB$$
$$A \rightarrow abb$$
$$B \rightarrow Ac$$
$$A \rightarrow ab$$

Avremo:



Dal linguaggio delle parentesi:



Questo albero mi da la struttura sintattica data la stringa (( ) ( )) ( )
a a b a b b a b

Lezione 20 5/ 12/ 2002

Albero di Derivazione Sintattica

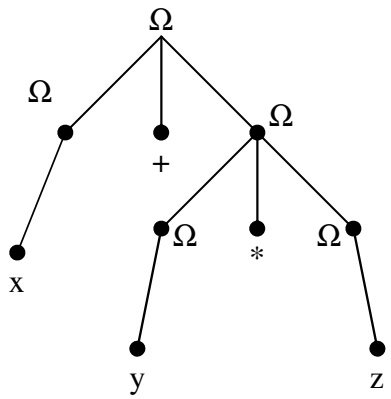
Supponiamo di avere:

$$\Sigma = \{ x,y,z,+, * \}$$
$$V= \{ \Omega \}$$

Con le seguenti regole di produzione :

1. $\Omega \rightarrow \Omega + \Omega$
2. $\Omega \rightarrow \Omega * \Omega$
3. $\Omega \rightarrow x$
4. $\Omega \rightarrow y$
5. $\Omega \rightarrow z$

se abbiamo :



Questo è un albero ordinato perché:

1. Etichette dei nodi interni sono simboli non terminali
2. le foglie sono simboli terminali

in questo caso la stringa è w= x + y *z

Ambiguità :

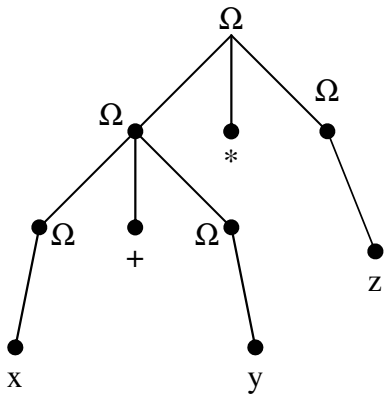
Una grammatica G è ambigua se esiste una parola w ∈ L(G) che ammette più di un albero di derivazione.

Esempio :

Nell'esempio precedente la grammatica vista è ambigua infatti l'espressione:

$$x + y * z \qquad \text{(sintassi)}$$

può essere rappresentata anche dall'albero di derivazione :



(semantica)

Questa ambiguità semantica nasce già da un' ambiguità sintattica, ed è una debolezza per questa grammatica.
Questa osservazione ci dice che una grammatica CF può essere ambigua, però non esiste un algoritmo che mi permette di decidere se una grammatica è ambigua o meno.
Però esistono grammatiche che ammettono grammatiche non ambigue equivalenti (che generano lo stesso linguaggio). Il problema è in decidibile.
A questo punto facciamo delle precisazioni:

1. Non esiste un algoritmo che mi permette di generare una grammatica non ambigua da una grammatica ambigua.
2. Esistono grammatiche ambigue che ammettono grammatiche non ambigue.
3. Un linguaggio context-free è ambiguo se la grammatica che lo genera è ambigua.
4. Esistono linguaggi context-free ambigui.

Per rimuovere tale ambiguità ci sono varie strategie, ma una di queste è quella di cercare un'altra grammatica.

Esempio:
Dato l'alfabeto:

$$\Sigma = \{ x, y, z, +, *, (,) \}$$

Costruiamo una grammatica dove le stringhe sono del tipo :

$$((x + y) * z)$$

ossia le espressioni aritmetiche con parentesi.
Oppure generare una grammatica che ci da espressioni in forma prefissa (o polacca) ad esempio:

$$((x + y) * (z * x)) + (y + (x + z))$$ corrisponde a **+*** + x y * z x + y + x z

A questo punto dobbiamo porci un altro problema:
Esiste una grammatica minimale?

Prima di tutto dobbiamo dire che l' EQP per i linguaggi generati da grammatiche CF è indecidibile.
Infatti se abbiamo due linguaggi $L_1(G_1)$ e $L_2 (G_2)$ non sappiamo se:

$$L_1(G_1) = L_2 (G_2)$$

Dove G_1 e G_2 sono CF.
Da questo possiamo capire perché non esiste una grammatica minimale.
Però le grammatiche si possono semplificare attraverso le forme normali che sono di due tipi:

1. Forma normale di Chomsky o (CNF)
2. Forma normale di Greibach o (GNF)

Forma normale di Chomsky
Data una grammatica G questa si dice che è CNF se le regole di produzione sono nella forma:

$$A \to a$$
$$A \to BC$$

***Teorema** :*
Data una grammatica CF ne possiamo ricavare sempre una in forma normale CNF e viceversa.

Dimostrazione:
Dimostriamo solo la prima inclusione

$$CF \Rightarrow CNF$$

Esiste un algoritmo che consta di alcuni passi:

1° Passo:

Prendiamo una regola di produzione:

$$A \to X_1 X_2 X_3 \ldots X_n \text{ con } x_i \in (\Sigma \cup V)$$

e la sostituisco con:

$$A \to B_1 B_2 B_3 \ldots B_n$$

dove:

$B_i = X_i$ se $x_i \in V$ (simboli non terminali)
Se $X_i \in \Sigma$, se per esempio $X_i = a \in \Sigma$, introduco un nuovo simbolo non terminale **Ba**, lo sostituisco a x_i ed aggiungo una nuova regola **Ba → a**.

Esempio :
Se ho la seguente regola :

$$A \to BaCDbC$$

Questa diventa :

$A \to BB_aCDB_bC$
$B_a \to a$
$B_b \to b$

Il linguaggio generato è lo stesso.

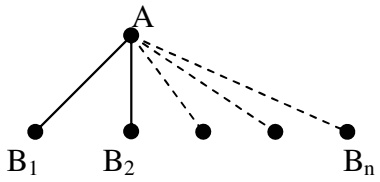
Dopo il primo passo ho una grammatica che ha due tipi di regole :

$A \rightarrow B_1B_2\ldots\ldots B_n$
$A \rightarrow a$

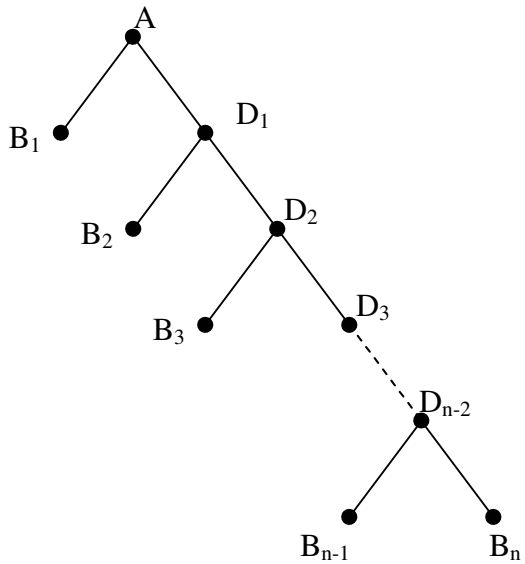
Per ognuna di queste introduco nuovi simboli non terminali $D_1D_2\ldots\ldots$ e sostituisco la regola con il seguente insieme di regole :

$A \rightarrow B_1D_1$
$D_1 \rightarrow B_2D_2$
$D_2 \rightarrow B_3D_3$
……
$D_{n-1} \rightarrow B_nD_n$

Quindi con $A \rightarrow B_1B_2\ldots\ldots B_n$ avrò l'albero :



Applicando le regole ottengo un albero binario :



Esempio :
Se ho le seguenti regole di produzione :

$\Omega \rightarrow bA$
$\Omega \rightarrow aB$
$A \rightarrow a$
$A \rightarrow bAA$
$B \rightarrow b$
$B \rightarrow b\Omega$
$B \rightarrow aBB$

Introduco :

$C_a \rightarrow a$
$C_b \rightarrow b$

Sostituendo ottengo :

$\Omega \rightarrow C_bA$
$\Omega \rightarrow C_aB$
$A \rightarrow C_a$
$A \rightarrow C_bAA$ (*)
$B \rightarrow C_b$
$B \rightarrow C_b\Omega$
$B \rightarrow C_aBB$ (*)

Sono tutte nella forma CNF tranne quelle con (*). Allora introduco :

$\left.\begin{array}{l} A \rightarrow C_bD_1 \\ D_1 \rightarrow AA \end{array}\right\}$ ma anche $\left\{\begin{array}{l} B \rightarrow C_aD_2 \\ D_2 \rightarrow BB \end{array}\right.$

E le vado a sostituire. (Posso mettere il linguaggio delle parentesi in questa cerchia di linguaggi).

Lezione 21 6/12/2002

Grammatiche lineari destre (sinistre) Tipo 3 :
Queste grammatiche sono così definite :

$\left\{\begin{array}{l} A \rightarrow a \\ A \rightarrow aB \end{array}\right.$ O $\left\{\begin{array}{l} A \rightarrow a \\ A \rightarrow Ba \end{array}\right.$

Detta *Lineare a Destra* Detta *Lineare a Sinistra*

Se abbiamo una grammatica :

$$G = (\Sigma, V, P, \Omega)$$

E un automa A :

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

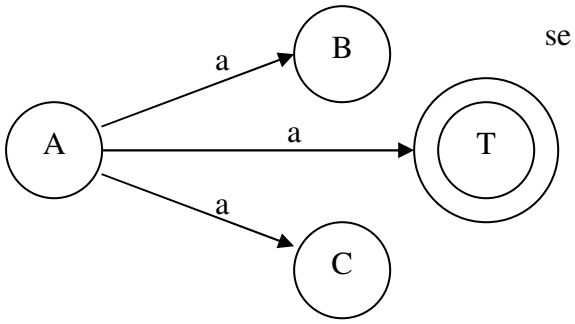
Dove :

$Q = V \cup \{T\}$ con $T \notin V$
$q_0 = \Omega$

Diremo che :

- $\delta(A, a) = B$ allora $A \rightarrow aB$
- $\delta(A, a) = T$ allora $A \rightarrow a$

In poche parole mi costruisco un NFA dove :



se

$A \rightarrow aB$

$A \rightarrow a$

$A \rightarrow aC$

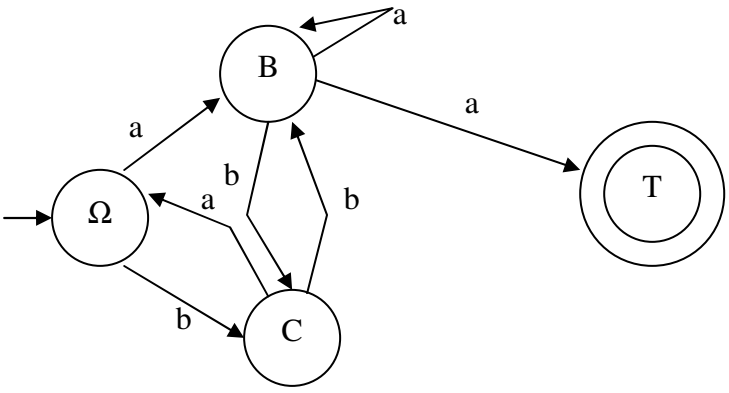
E quindi

$F = \{T\}$ con $T \notin V$

Esempio :
Se ho una grammatica con le seguente regole di produzione :

1° : $\Omega \rightarrow aB$
2° : $\Omega \rightarrow bC$
3° : $B \rightarrow aB$
4° : $B \rightarrow a$
5° : $B \rightarrow bC$
6° : $C \rightarrow a\Omega$
7° : $C \rightarrow bB$

se costruiamo l'automa mettendo in input una stringa **x** avrò in output un determinato stato in uscita. Proviamo a generare l'automa :



Generiamo una derivazione:

$\Omega \Rightarrow_1 aB \Rightarrow_3 aaB \Rightarrow_5 aabC \Rightarrow_6 aaba\Omega \Rightarrow_2 aababC \Rightarrow_7 aababbB \Rightarrow_4 aababba$

se la facciamo leggere all'automa:

$\Omega^a B^a B^b C^a \Omega^b C^b B^a T$

questo è un modo di vedere una grammatica come un automa, ma posso costruirmi una grammatica partendo da un automa a stati finiti..

Esempio:
Supponiamo di avere un automa:

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

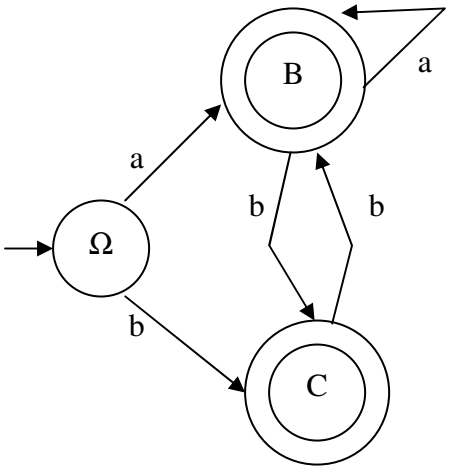
E una grammatica:

$$G = (\Sigma, V, P, \Omega)$$

dove poniamo:

$$V = Q$$
$$\Omega = q_0$$

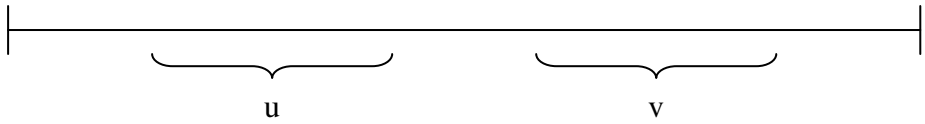
inoltre :
se $\delta(q_i , a) = s \Rightarrow$ per ogni arco dell'automa $q \rightarrow as$
se $s \in F$ (è uno stato di accettazione) allora aggiungo la regola $q \rightarrow a$
quindi dato l'automa:



1. $\Omega \rightarrow aB$
2. $\Omega \rightarrow bC$
3. $B \rightarrow aB$
4. $B \rightarrow a$
5. $B \rightarrow bC$
6. $B \rightarrow b$
7. $C \rightarrow bB$
8. $C \rightarrow b$
9. $\Omega \rightarrow a$
10. $\Omega \rightarrow b$

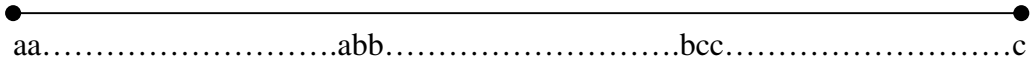
Ovviamente questo tipo di costruzione la posso fare solo se ho un automa deterministico.

Lemma di Iterazione per i linguaggi CF :
Dato un linguaggio $L \in L(CF)$ (linguaggi generati da grammatiche Context-free) ci chiediamo se esiste uno strumento per capire se può essere generato da una grammatica CF.
Già per i linguaggi riconosciuti da automi a stati finiti, abbiamo introdotto il Lemma di Iterazione, possiamo quindi introdurne uno anche per i linguaggi Context-free.
Prima di dare l'enunciato del lemma, diamo un'idea intuitiva. Nel caso del lemma d'iterazione per i linguaggi riconosciuti da automi a stati finiti, l'idea intuitiva è questa : il lemma dice che se il linguaggio è riconosciuto da un FSA, se prendo una stringa abbastanza lunga, è possibile individuare un blocco di tale stringa, tale blocco può essere cancellato, o ripetuto più volte, e quello che otteniamo è sempre una stringa del linguaggio.
Il lemma di iterazione per i linguaggi Context-free, dice che se prendo una parola abbastanza lunga per il linguaggio Context-free, ho una coppia di fattori iteranti (u, v) che si chiama **coppia iterante**.

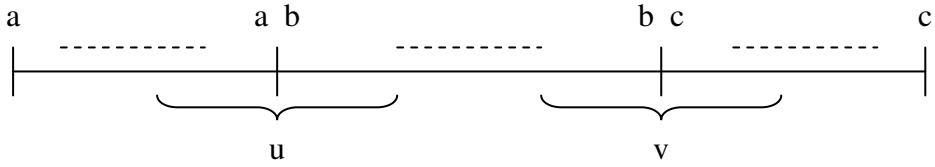


Quindi se cancello entrambi i blocchi o li itero più volte, ottengo ancora una stringa che sta nel linguaggio, se non gode di questa proprietà, allora non è Context-free.
Questa coppia (u, v) la posso scegliere con u, v *non troppo lontani*.

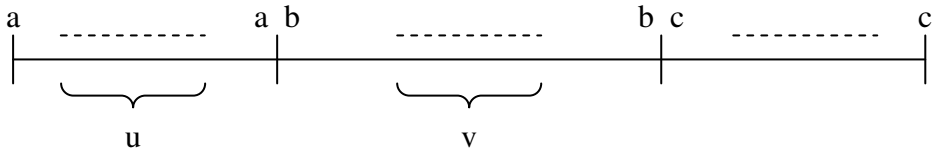
Esempio:
Prendiamo il linguaggio $L = \{a^n, b^n, c^n \mid n \geq 1\}$ sull'alfabeto $\Sigma = \{a, b, c\}$.
Le stringhe di questo linguaggio sono del tipo:



il numero di a deve essere uguale al numero di b e al numero di c.
Questo linguaggio non può essere generato da una grammatica Context-Free perché non soddisfa le condizioni date.
Infatti devo trovare due blocchi u,v, questi blocchi non possono stare a cavallo di questa linea di demarcazione, ovvero così :



Poiché iterandoli ottengo una cosa in cui mescolo le **a** e le **b**, e quindi ottenni qualcosa che non appartiene al linguaggio. Questa coppia di fattori iteranti deve stare ad esempio **u** dentro le **a**, e **v** dentro le **b**.

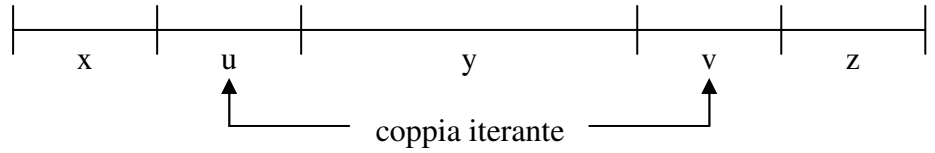


In questo esempio non mi serve la condizione che debbono essere non troppo lontani.
Se cancello **u** e **v**, modifico le **a** e le **b** ma non modifico le **c**, e pertanto non ottengo una stringa del linguaggio. Quindi $L = \{a^u b^u c^u / u \geq 1\} \notin L(CF)$.

Vediamo adesso l'enunciato vero e proprio del *Lemma d'iterazione per i Linguaggi CF* :
Se abbiamo un linguaggio $L \subseteq \Sigma^*$ che è CF allora :

> $\exists$ un $N \in Z_+$ tale che $\forall\ w \in L$, con $|w| > N$, $\exists$ una fattorizzazione $w = xuyvz$ con $|uv| > 0$ e con $|uyv| < N$ tale che $\forall\ n \geq 0 : xu^n yv^n z \in L$

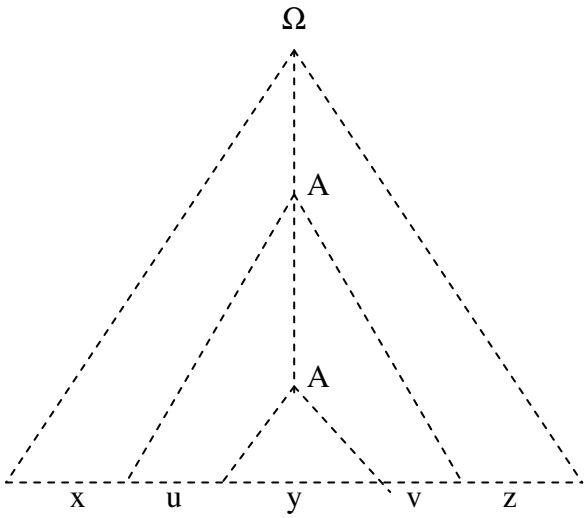
Quindi data la stringa :



Dimostrazione :
Se abbiamo una grammatica CF :

$$G = (\Sigma, V, P, \Omega)$$

Che genera un linguaggio L = L(G), se abbiamo |w|>N ciò implica che l'albero ha una profondità $p \geq |V|$ quindi avrò due o più volte lo stesso simbolo.



Ho un numero di nodi maggiore di |V| quindi ho due nodi almeno con la stessa etichetta, il che vuol dire che vado a prendere il sottoalbero che aprte da A, e dopo aver individuato questi due sottoalberi come in figura scompongo la stringa w nella stringa: x u y v z.
Mi ricavo le regole o le derivazioni come segue:

1. $\Omega \Rightarrow^* x\ A\ z$
2. $A \Rightarrow^* u\ A\ v$
3. $A \Rightarrow^* y$

Facciamo alcune derivazioni e vediamo la coppia iterante:
$\Omega \Rightarrow^* x\ A\ z \Rightarrow^* x\ u\ A\ v\ z \Rightarrow^* x\ u\ y\ v\ z$
oppure :
$\Omega \Rightarrow^* x\ A\ z\ \Omega \Rightarrow^* x\ u\ A\ v\ z\ \Omega \Rightarrow^* x\ x\ u\ u\ A\ v\ v\ z\ \Omega \Rightarrow^* x\ u\ u\ y\ v\ v\ z$
oppure
$\Omega \Rightarrow^* x\ A\ z \Rightarrow^* x\ y\ z$
quindi

$$x\ u^n\ y\ v^n\ z$$

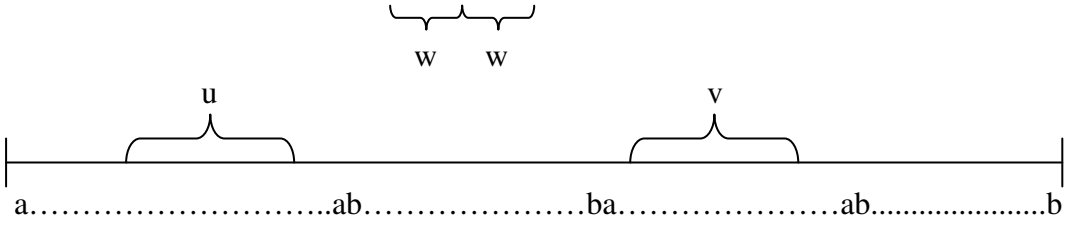
questo lemma rende ancora più chiaro il linguaggio delle parentesi ; infatti noi sappiamo che se vogliamo semplificare una stringa dobbiamo lavorare a coppie di parentesi.

Esempio:
Se abbiamo il linguaggio delle palindrome L dove:

$$L = \{ww \ / \ w \in \Sigma^*\}$$

Questo linguaggio non è generato da una grammatica CF.

Uso il lemma d'iterazione. Ci mettiamo sull'alfabeto $\Sigma = \{a, b\}$ prendo una stringa molto lunga, ed è fatta in modo tale che la prima metà sia uguale alla seconda metà.
Quindi tale stringa sarà della forma $t= a^n b^m \; a^n b^m$.



Il lemma di Iterazione mi dice che posso scegliere una coppia iterante in modo che i due elementi della coppia non siano troppo lontani, cioè che abbiano una distanza minore di un k fissato.
Se prendo n,m maggiori di k dove vado a mettere i fattori iteranti?
Se uno lo metto tra le a della prima w, per ottenere una stringa corretta l'altra coppia dovrei metterla tra le a della seconda w.
Poiché m>k allora u e v sono più lontani di k e quindi non vale il lemma di Iterazione.
Quindi se prendo una stringa di questa forma non riesco a trovare fattori iteranti vicini.
Possiamo concludere dicendo che il linguaggio copy non è generato da una grammatica CF.

Lezione 22 9/12/2002

Proprietà di chiusura dei linguaggi CF

I linguaggi CF sono:

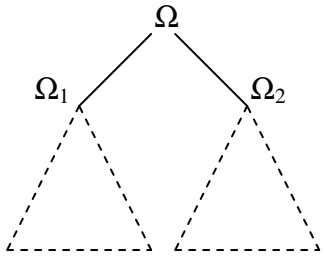
- **Chiusi rispetto a $\cup$ (unione)**
  Presi due linguaggi L_1 L_2 dove:
  $L(G_1) = L_1$ e $G_1= \{\Sigma, V_1, P_1, \Omega_1\}$
  $L(G_2) =L_2$ e $G_2= \{\Sigma, V_2, P_2, \Omega_2\}$
  L'unione sarà:
  $L_1 \cup L_2 \Rightarrow G = \{\Sigma, V_1 \cup V_2, \cup \{\Omega\}\; P_1 \cup P_2 \cup \{\Omega \rightarrow \Omega_1, \Omega \rightarrow \Omega_2\}, \Omega\}$
  Dove ovviamente:

  $$\Omega \notin V_1 \cup V_2$$

  quindi avremo:

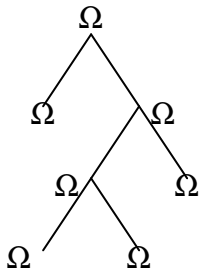
  $\Omega \Rightarrow \Omega_1$ ……..$w \in L_1$
  $\Omega \Rightarrow \Omega_2$ ……..$w \in L_2$

- **Chiusi rispetto alla concatenazione**
  Presi due linguaggi L_1 L_2 dove:
  $L(G_1) = L_1$ e $G_1= \{\Sigma, V_1, P_1, \Omega_1\}$
  $L(G_2) =L_2$ e $G_2= \{\Sigma, V_2, P_2, \Omega_2\}$
  La concatenazione sarà:
  $L_1L_2 \Rightarrow G = \{\Sigma, V, P, \Omega\}$
  Dove:
  $V = V_1 \cup V_2, \cup \{\Omega\}$
  $P= P_1 \cup P_2, \cup \{\Omega \rightarrow \Omega_1\Omega_2\}$
  Ossia:

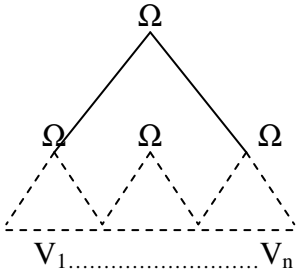


- **Chiusi rispetto a star ***
  Dato un Linguaggio L generato da una grammatica G = {Σ, V, P, Ω} ⇒ L = L(G)
  L'operazione * sarà così definita:
  L* è generato da una grammatica G' = {Σ, V, P, Ω}

  Dove:
  $P = P \cup \{\Omega \rightarrow \Omega\Omega\}$
  Ossia:



ossia



- **Non Chiuso rispetto all'intersezione (∩)**
  Dato il linguaggio

$$L_1 = \{ a^n b^m c^k / n, m, k \geq 1 \text{ and } n=m \}$$
$$L_2 = \{ a^n b^m c^k / n, m, k \geq 1 \text{ and } m=k \}$$

  Il primo lo possiamo dividere in:

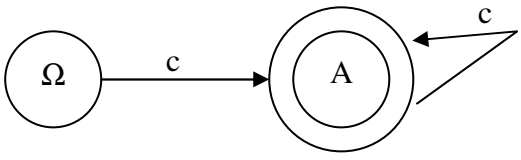
$$L_1 = \{ a^n b^n / n \geq 1 \} \cdot \{ c^k / k \geq 1 \}$$

  Vediamo che la concatenazione di un linguaggio regolare con un linguaggio CF, da un linguaggio CF; infatti :

$$\{a^n b^n / n \geq 1\} \quad \Rightarrow \quad \begin{array}{l} S \rightarrow aSb \\ S \rightarrow ab \end{array}$$

  mentre :

$$\{c^k / k \geq 1\} \quad \Rightarrow$$



dove :

. $\Omega \to cA$
. $\Omega \to c$
. $A \to cA$
. $A \to c$

} Grammatica

mentre il L_2 non succede.

Forma normale di Greibach (GNF) :
Una grammatica è nella **GNF** se ha le seguenti regole :

$$A \to a\alpha \qquad \alpha \in V^*$$

Ogni grammatica può essere trasformata in una GNF equivalente.

Derivazione <u>Left-Most</u> (Canonica sinistra) : si sostituisce il simbolo non terminale più a sinistra
Se abbiamo le seguenti regole di produzione :

$A_1 \to bB_1 \ldots B_k$
$B_1 \to cC_1 \ldots C_r$

Applicando la *derivazione Left-Most* otteniamo :

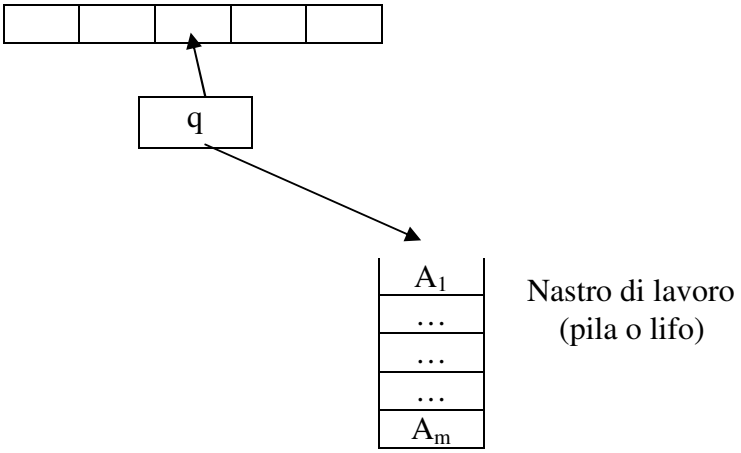
$\Omega \Rightarrow aA_1A_2 \ldots A_n \Rightarrow abB_1 \ldots B_kA_2 \ldots A_n \Rightarrow abcC_1 \ldots C_rB_2 \ldots B_kA_2 \ldots A_n \Rightarrow \ldots\ldots$

Con questo tipo di derivazione, se parto dall'assioma alla fine avrò la seguente forma :

$\Omega \Rightarrow \ldots\ldots \Rightarrow a_1a_2 \ldots \underbrace{a_kA_1A_2 \ldots A_m} \Rightarrow \ldots\ldots$

Questi stati sono in numero limitato, questo è il motivo per cui posso farlo come una pila.

In una derivazione *left-most* avrò ad un certo punto, una parte fatta solo da simboli terminali, ed una parte da non terminali :



Se applico

$A_1 \rightarrow b$ avrei : $a_1a_2\ldots a_kbA_2\ldots A_m$

$A_1 \rightarrow bB_1B_2$ avrei : $a_1a_2\ldots a_kbB_1B_2A_2\ldots A_m$

Teorema :

Un linguaggio è generato da una grammatica CF se e solo se è riconosciuto da un NPDA.

Indice degli argomenti :