Università degli Studi di Ferrara

Capitolo 3
Automi Riconoscitori

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Dipartimento di Ingegneria

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

RICONOSCIBILITÀ DEI LINGUAGGI

- Perché il traduttore possa essere realizzato in modo efficiente, conviene adottare **linguaggi generati da** (classi speciali di) **grammatiche di Tipo 2**
  - Tutti i linguaggi di programmazione sono infatti *context free*
  - Il riconoscitore prende il nome di **PARSER**
- Per ottenere *particolare efficienza* in sotto-parti di uso *estremamente frequente*, si adottano spesso per esse **linguaggi generati da grammatiche di Tipo 3**
  - identificatori & numeri
  - Il riconoscitore prende il nome di **SCANNER** (o ***lexer***)

RICONOSCIBILITÀ DEI LINGUAGGI

- **I linguaggi generati da grammatiche di Tipo 0 *possono in generale NON essere riconoscibili* (decidibili)**
  - Non è garantita l'esistenza di una MdT capace di decidere se una frase appartiene o meno al linguaggio
- **Al contrario, i linguaggi generati da grammatiche di Tipo 1** (e quindi di Tipo 2 e 3) **sono *riconoscibili***
  - Esiste sempre una MdT capace di decidere se una frase appartiene o meno al linguaggio
  - **L'efficienza del processo di riconoscimento, però, è un'altra faccenda…**

QUALI MACCHINE PER QUALI LINGUAGGI?

Chi riconosce i diversi tipi di linguaggi?

GRAMMATICHE	AUTOMI RICONOSCITORI
• **Tipo 0**	• ***Se L(G) è riconoscibile*, occorre una Macchina di Turing**
• **Tipo 1**	• **Macchina di Turing (con nastro limitato)**
• **Tipo 2 *(context-free)***	• **Push-down automaton (ASF + stack)**
• **Tipo 3 *(regolari)***	• **Automa a Stati Finiti (ASF)**

RICONOSCITORI A STATI FINITI

Un automa riconoscitore a stati finiti è una quintupla

$$\langle A, S, S_0, F, sfn \rangle$$

dove

- **A = alfabeto**
- **S = insieme degli stati**
- **S_0 = stato iniziale $\in$ S**
- **F = insieme degli *stati finali* $\subseteq$ S**
- **sfn: $S \times A \rightarrow S$** *(state function)*

RICONOSCITORI A STATI FINITI

ESEMPIO

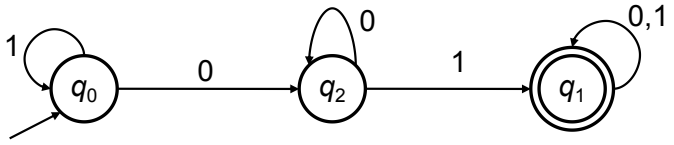
Esempio: Un automa A che accetta

L = {x01y : x, y $\in$ {0, 1}* }

L'automa A = ({0, 1}, {q_0 , q_1 , q_2 }, q_0 , {q_1}, sfn) dove sfn è data dalla tabella di transizione:

sfn	0	1
q_0	q_2	q_0
q_1	q_1	q_1
q_2	q_2	q_1

oppure dal diagramma di transizione



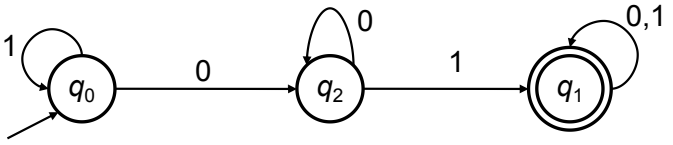
ACCETTAZIONE

Un riconoscitore a stati finiti (FA) accetta una stringa
w = $a_1a_2\ldots a_n$

se esiste un cammino nel diagramma di transizione che

1. Inizia nello stato iniziale
2. Finisce in uno stato finale (di accettazione)
3. Ha una sequenza di etichette a_1 a_2 … a_n

Ad esempio, l'automa sotto riconosce la stringa 110010



Funzione di accettazione sfn*

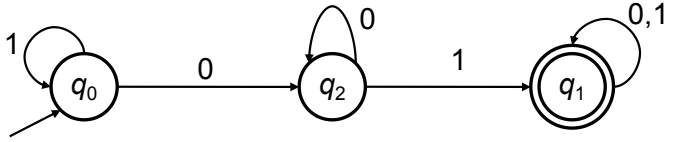
La funzione *sfn*, che opera su coppie (*stato, simbolo*),

può essere estesa ad una funzione *sfn**, che opera su coppie (*stato*, *stringa*)

$sfn^*(q,\varepsilon) = q$

$sfn^*(q, sc) = sfn(sfn^*(q,s), c)$ ($s \in A^*$ è una stringa, $c \in A$ è un simbolo)

Es: $sfn^*(q_0,100) = sfn(sfn^*(q_0,10),0) = sfn(sfn(sfn^*(q_0,1),0),0)$
$= sfn(sfn(sfn(sfn^*(q_0,\varepsilon),1),0),0)$



Linguaggio accettato

Formalmente il linguaggio accettato dall'automa

$R = \langle A, S, S_0, F, sfn \rangle$

è l'insieme delle parole *w* che portano dallo stato iniziale S_0 ad uno degli stati finali:

$L(R) = \{ w : sfn^*(S_0,w) \in F \}$

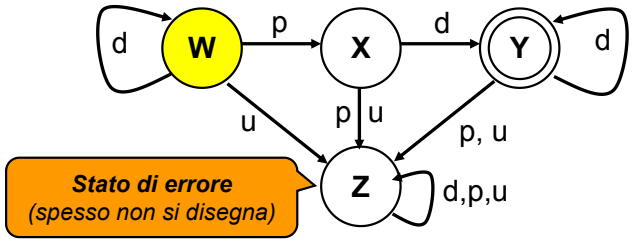
ESEMPIO

ESEMPIO di RSF

Sia:

- **A = { d, p, u }**
- **S = { W, X, Y, Z }**
- **S_0 = W**
- **F = { Y }**
- **sfn: A × S → S**

	d	p	u
W	W	X	Z
X	Y	Z	Z
Y	Y	Z	Z
Z	Z	Z	Z



Stato di errore
(spesso non si disegna)

stato iniziale *stato finale*

Frasi accettate: **dpd, dddpd, pd, pdd, …**
Frasi non accettate: **dp, u, dpdp...**

Il linguaggio L(R) accettato dal riconoscitore R è *finito* se la rappresentazione grafica non presenta cicli.

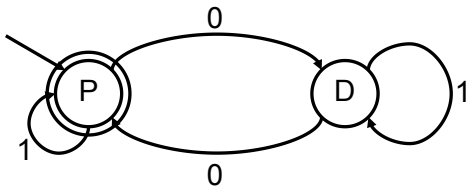
Esercizi

Si scrivano dei riconoscitori a stati finiti per i seguenti linguaggi sull'alfabeto {0,1}:

- Insieme di tutte le stringhe che hanno un numero pari di 0 e un numero pari di 1
- Insieme di tutte le stringhe che finiscono con 00
- Insieme di tutte le stringhe con tre 0 consecutivi
- Insieme di tutte le stringhe che cominciano o finiscono con 01 (o entrambe le cose)

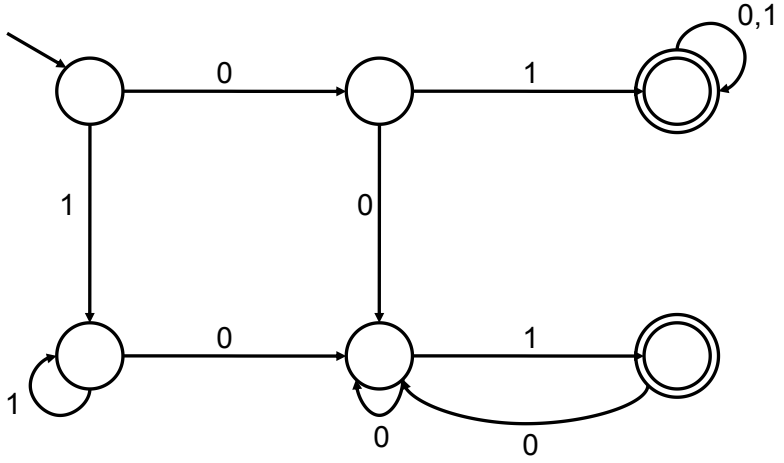
Si scrivano dei riconoscitori a stati finiti per i seguenti linguaggi sull'alfabeto {0,1}:

- Insieme di tutte le stringhe che hanno un numero pari di 0



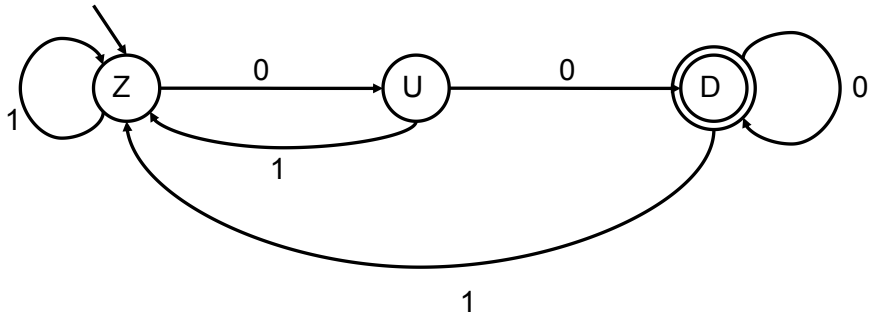
- Insieme di tutte le stringhe che hanno un numero pari di 0 e un numero pari di 1

- Insieme di tutte le stringhe che cominciano o finiscono con 01 (o entrambe le cose)



Si scrivano dei riconoscitori a stati finiti per i seguenti linguaggi sull'alfabeto {0,1}:

- Insieme di tutte le stringhe che finiscono con 00



- Insieme di tutte le stringhe con tre 0 consecutivi

Università degli Studi di Ferrara

Capitolo 3.2
Automi Riconoscitori

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Dipartimento di Ingegneria

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

TEOREMI

TEOREMA 1

Un linguaggio L(R) è *non vuoto* se e solo se il riconoscitore R accetta una stringa x di *lunghezza L_x minore del numero di stati N* dell'automa

DIMOSTRAZIONE

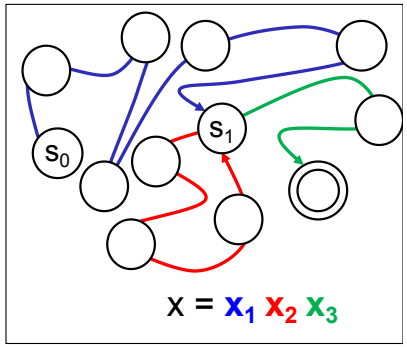
Se L(R) non è vuoto, R accetta una stringa x.

Se $L_x > N$, l'automa R deve passare più di una volta per uno stesso stato s_1.

Decomponendo allora la stringa x come $x_1\ x_2\ x_3$, con $x_2 \neq \varepsilon$, deve aversi:

$sfn^*(x_1, s_0) = s_1 \quad sfn^*(x_2, s_1) = s_1 \quad sfn^*(x_3, s_1) \in F$

Ma allora anche la stringa $y = x_1 x_3$ è accettata, poiché anch'essa porta R in uno stato finale; e la lunghezza L_y di tale stringa è certamente minore di L_x.



Se $L_y < N$, il teorema è dimostrato; altrimenti si può iterare il ragionamento fino a trovare una stringa z tale che $L_z < N$.
Il viceversa è banale, in quanto se R accetta una stringa (di qualsiasi lunghezza, quindi anche < N), L(R) è - ovviamente - non vuoto.

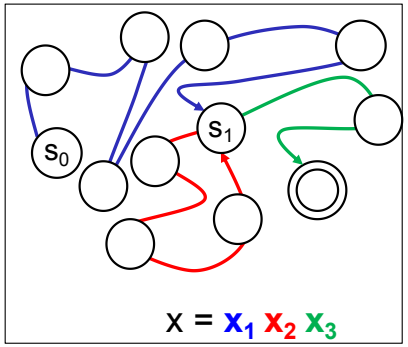
TEOREMA 2

Un linguaggio L(R) è *infinito* se e solo se il riconoscitore R accetta una stringa x di lunghezza $N \leq L_x < 2N$, con N numero di stati dell'automa.

DIMOSTRAZIONE

⇒) Sia $L(R)$ infinito. Essendo infinito, non può contenere solo stringhe di una lunghezza limitata. Sia $x \in L(R)$ una stringa di lunghezza $|x| > N$. Ripetendo il ragionamento del Teorema 1, deve potersi scrivere $x = x_1\ x_2\ x_3$, con $x_2 \neq \varepsilon$ e con $sfn^*(x_1, s_0) = s_1 \quad sfn^*(x_2, s_1) = s_1 \quad sfn^*(x_3, s_1) \in F$.

Poiché $sfn^*(x_2, s_1) = s_1$, R accetta in realtà tutte le stringhe della forma $x_1\ x_2^k\ x_3$, con $k \geq 0$.



Se $L_x < 2N$, il teorema è dimostrato.

Altrimenti, se $L_x > 2N$, consideriamo la lunghezza di x_2 e quella di $x' = x_1\ x_3$ (che è una stringa accettata dall'automa).

Se $|x_2| < N$, e $|x_1\ x_3| < N$, allora $|x| = |x_1\ x_3| + |x_2| < 2N$.

Altrimenti, se $|x_2| > N$, oppure $|x_1\ x_3| > N$, vuol dire che in una di queste sottostringhe c'è una parte ripetuta e itero il ragionamento su quella sottostringa.

⇐) Viceversa, se accetta una stringa di lunghezza $N \leq L_x < 2N$, esiste una sottostringa che può essere ripetuta infinite volte, dunque $L(R)$ è infinito.

CONSEGUENZE

- **In conseguenza di questi due teoremi, decidere se un *linguaggio regolare* sia *vuoto* o *infinito* è un problema *risolubile***
  - nel primo caso, basta esaminare se esiste una stringa accettata di lunghezza minore di N
  - nel secondo caso, basta verificare se esiste una stringa accettata fra quelle di lunghezza compresa fra N (incluso) e 2N (escluso).
- **Come si vedrà più avanti, tali proprietà *sono decidibili anche nel Tipo 2*, mentre *non lo sono nel Tipo 1 (e 0).***

Riconoscitori e grammatiche

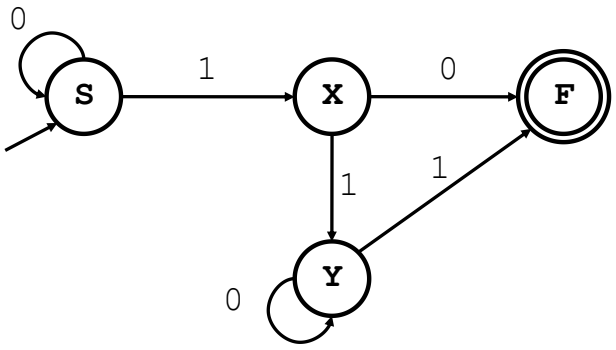
S → 1 X | 0 S

X → 1 Y | 0

Y → 1 | 0 Y

11001

- dove lo scopo è S.



RICONOSCITORI “top down”

Data una grammatica regolare *lineare a destra*, il riconoscitore:

- ha tanti stati quanti i simboli non terminali, più uno stato finale F
- ha come stato iniziale lo scopo S
- per ogni regola del tipo X → x Y l’automa, con ingresso x, si porta dallo stato X allo stato Y
- per ogni regola del tipo X → x l’automa, con ingresso x, si porta dallo stato X allo *stato finale F.*

Derivazione *top down*

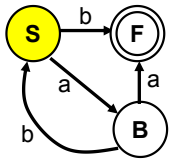
- si parte dallo scopo della grammatica
- e si tenta di *coprire la frase data* tramite produzioni successive.

ESEMPIO

Sia G una grammatica lineare a destra caratterizzata dalle produzioni:

S → a B | b
B → b S | a

Automa top-down corrispondente:



La frase a b a a è riconosciuta partendo dallo scopo S e generando via via la forma di frase, trovandosi, a simboli esauriti, nello *stato finale F*:

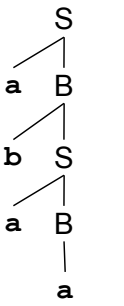
S → a B → a b S → a b a B → a b a a

RICONOSCITORI “top down”

Derivazione *top down*

- si parte dallo scopo della grammatica
- e si tenta di *coprire la frase data* tramite produzioni successive.

a b a a

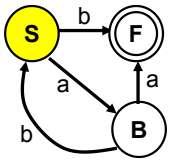


ESEMPIO

Sia G una grammatica lineare a destra caratterizzata dalle produzioni:

S → a B | b
B → b S | a

Automa top-down corrispondente:



La frase a b a a è riconosciuta partendo dallo scopo S e generando via via la forma di frase, trovandosi, a simboli esauriti, nello *stato finale F*:

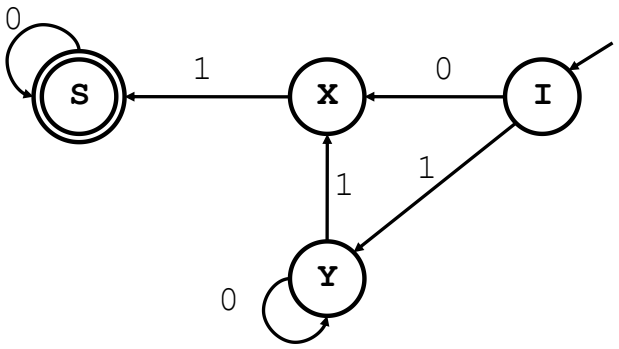
S → a B → a b S → a b a B → a b a a

Riconoscitori e grammatiche

S → X 1 | S 0
X → Y 1 | 0
Y → 1 | Y 0

10011

- dove lo scopo è S.



RICONOSCITORI “bottom up”

Data una grammatica regolare *lineare a sinistra*, il RSF:

- ha tanti stati quanti i simboli non terminali, più uno stato iniziale I
- ha come stato finale lo scopo S
- per ogni regola del tipo X → Y x l’automa, con ingresso x, riduce lo stato Y allo stato X
- per ogni regola del tipo X → x l’automa, con ingresso x, riduce lo *stato iniziale I* allo stato X.

Derivazione *bottom up*

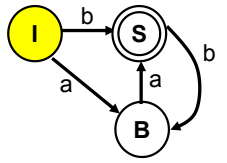
- si parte dalla frase data
- e si risale verso lo scopo S tramite *riduzioni successive.*

ESEMPIO

Sia G una grammatica lineare a sinistra con le produzioni:

S → B a | b
B → S b | a

Automa bottom-up relativo:



La frase a a b a è riconosciuta partendo dallo stato iniziale I e riducendosi ogni volta a una forma di frase più corta, fino allo scopo S:

a a b a → B a b a → S b a → B a → S

RICONOSCITORI “bottom up”

Derivazione *bottom up*
- **si parte dalla frase data**
- **e si risale verso lo scopo S tramite *riduzioni successive.***

a a b a

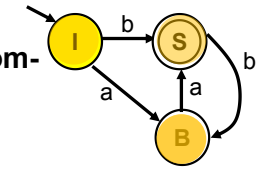
S
B a
S b
B a
a

ESEMPIO

Sia G una grammatica lineare a sinistra con le produzioni:

S → B a | b
B → S b | a

Automa bottom-up relativo:



La frase a a b a è riconosciuta partendo dallo stato iniziale I e riducendosi ogni volta a una forma di frase più corta, fino allo scopo S:

a a b a → B a b a → S b a → B a → S

Esercizio (tratto da 21 set 2016)

- Si consideri la seguente grammatica:

  S → Aa | Ba | a

  A → Ab | Bb

  B → Sa

- Si disegni l’automa riconoscitore del linguaggio generato.

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 3.3
Riconoscitori a Stati Finiti
Dall’automa alla grammatica

Corso di Laurea Magistrale in Ingegneria Informatica e dell’Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

MAPPING FRA GRAMMATICHE & RICONOSCITORI

Il mapping fra automa riconoscitore e grammatica:

- stati dell’automa ↔ metasimboli della grammatica
- transizioni dell’automa ↔ produzioni della grammatica
- uno stato dell’automa ↔ scopo della grammatica

presenta alcuni *gradi di libertà:*

- se la grammatica è regolare a destra, si ottiene un automa riconoscitore “top down”
- se la grammatica è regolare a sinistra, si ottiene un automa riconoscitore “bottom up”

Viceversa, *dallo stesso RSF* si possono trarre:

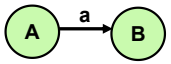
- una grammatica regolare a destra, interpretandolo top-down
- una grammatica regolare a sinistra, interpretandolo bottom-up

DALL'AUTOMA ALLE GRAMMATICHE

Dato un automa riconoscitore, se ne possono trarre:

- **una grammatica regolare a destra, interpretandolo top-down**
  - *approccio a **generazione**, si cerca di produrre la frase*
- **una grammatica regolare a sinistra, interpretandolo bottom-up**
  - *approccio a **riduzione**, dalla frase si cerca di raggiungere lo scopo*

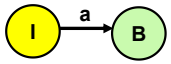
Caso generico:



Top-down: A → a B

Bottom-up: A a ← B

Caso iniziale:



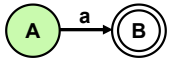
Top-down: S → a B (S ≡ I)

Bottom-up: I a ← B

Il caso dello stato iniziale è particolare nell'analisi *bottom-up*;

il caso dello stato finale è particolare nell'analisi *top-down*

Caso finale:

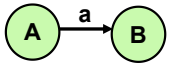


Top-down: A → a F

Bottom-up: A a ← S (S ≡ B)

UN ESEMPIO (1/2)

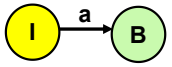
Caso generico:



Top-down: A → a B

Bottom-up: A a ← B

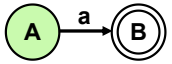
Caso iniziale:



Top-down: S → a B (I ≡ S)

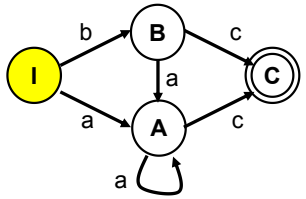
Bottom-up: I a ← B

Caso finale:



Top-down: A → a F

Bottom-up: A a ← S (B ≡ S)



La situazione è ideale, poiché vi è un solo stato finale, senza archi uscenti, e lo stato iniziale non ha archi in ingresso

Analisi *top-down (C finale, si omette)*:

I → b B B → a A B → c

I → a A A → a A A → c

Analisi *bottom-up* (*C iniziale, C ≡ S*):

b ← B B a ← A B c ← S

a ← A A a ← A A c ← S

UN ESEMPIO (2/2)

Grammatica *G1 regolare a destra*:

I → b B | a A

B → a A | c

A → a A | c

L(*G1*) = b L(B) + a L(A)

ma le produzioni per A e B sono identiche, quindi L(B)=L(A).

L(A) = a*c

Quindi L(G1)= (b+a) a*c

Grammatica G2 regolare a sinistra:

S → B c | A c

A → B a | A a | a

B → b

L(*G2*) =

L(B) c + L(A) c = (b + L(A)) c =

(b + (b a + a) a*) c = (b + a) a* c

perché (b + b a a*) = b (ε+aa*)=ba*

Non sempre però le cose vanno così lisce:

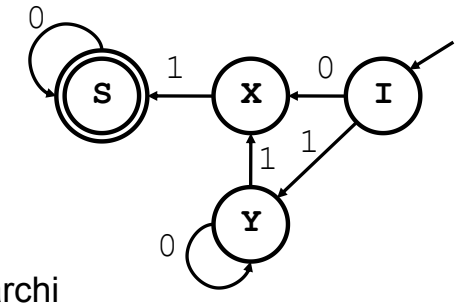
- l'analisi bottom-up *può non essere immediata in presenza di più stati finali,* che corrisponderebbero a *scopi multipli* di G2, o *in presenza di archi entranti nello stato iniziale*
- l'analisi top-down richiede attenzione se vi sono archi uscenti da stati finali.

Da RSF a grammatica lin Sx

S → X 1 | S 0

X → Y 1 | 0

Y → 1 | Y 0



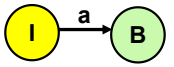
- Costruendo riconoscitori bottom-up abbiamo aggiunto un nodo iniziale, che ha solo archi uscenti
- Quindi dato un RSF in cui lo stato iniziale ha solo archi uscenti, possiamo applicare "*all'indietro*" le stesse regole viste per passare da grammatica ad automa
- Ma se l'automa dato ha archi entranti nello stato iniziale?

Caso finale:

(B ≡ S)

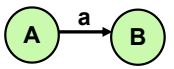
Caso iniziale:

B → a



Caso generico:

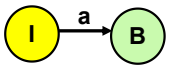
B → A a



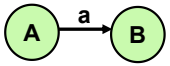
ESEMPIO

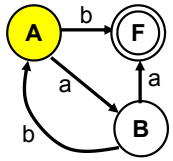
Caso finale:
(**F ≡ S**)

Caso iniziale:
B → a



Caso generico:
B →A a





F → b **B → a**
F → B a **A → B b**

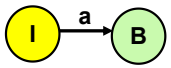
In questa grammatica, il linguaggio generato da **F** non contiene la frase **abb**. Non è lo stesso linguaggio riconosciuto dall'automa.

Non si tiene in considerazione che **A** non è solo lo stato iniziale: visto che si riesce a raggiungere da altri stati, funge anche da stato "generico"

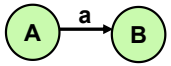
ESEMPIO

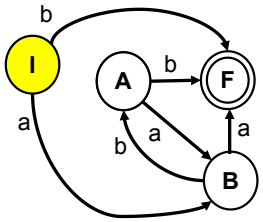
Caso finale:
(**F ≡ S**)

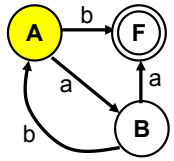
Caso iniziale:
B → a



Caso generico:
B →A a







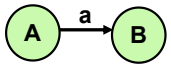
F → b **B → a**
F → B a **A → B b**
F → A b **B → A a**

Per avere un automa che riconosce lo stesso linguaggio, ma senza archi entranti nello stato iniziale, si potrebbe duplicare lo stato iniziale

Lo stesso risultato si può ottenere considerando per il nonterminale **A**
- *sia* la regola del caso iniziale
- *sia* la regola del caso generico

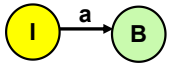
PIÙ STATI FINALI (1/2)

Caso generico:



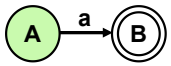
Top-down: **A → a B**
Bottom-up: **A a ← B**

Caso iniziale:

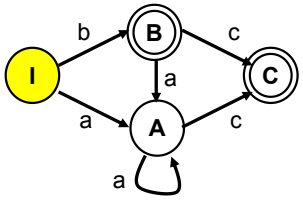


Top-down: **S → a B** (***S ≡ I***)
Bottom-up: **a ← B**

Caso finale:



Top-down: **A → a**
Bottom-up: **A a ← S** (***S ≡ B***)



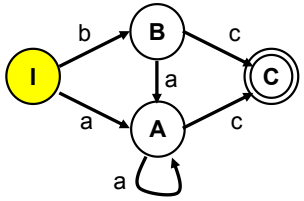
Ora anche lo stato B è finale.
L'analisi top-down non dà problemi...

Analisi *top-down (c'è una regola in più)*:
I → b B | b **B → a A** **B → c**
I → a A **A → a A** **A → c**

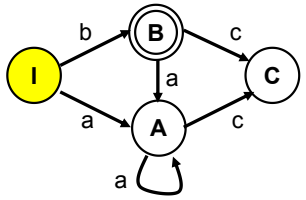
… ma l'analisi bottom-up non sa più che scopo adottare!! B o C ??

Analisi *bottom-up* (***B,C iniziali… S ??***):
b ← B **B a ← A** **B c ← C**
a ← A **A a ← A** **A c ← C**

PIÙ STATI FINALI (2/2)



Per uscire dal dilemma, è utile **esprimere il linguaggio riconosciuto o generato come *unione di due linguaggi,*** ciascuno denotato da una sua grammatica:



Grammatica G2' (assume **C ≡ S**):
S → B c | A c
A → B a | A a | a
B → b
L(*G2'*) = (**b** + **a**) **a*** **c**

Grammatica G2" (assume **B ≡ S**):
S → b
le altre regole sono inutili, essendo i loro metasimboli irraggiungibili
L(*G2"*) = **b**

Grammatica G2 (riunificazione di G2' e G2"):
S → B c | A c | b
A → B a | A a | a **B → b**
L(G2) = (**b** + **a**) **a*** **c** + **b**

Attenzione: rimappando G2 in un automa, esso potrebbe risultare ***non-deterministico.***

PIÙ IN GENERALE: (1/2)

Nel caso bottom-up, *in presenza di più stati finali:*
- si assume come (sotto-)scopo S_k uno stato finale per volta
- si scrivono le regole bottom-up corrispondenti
- si esprime il linguaggio complessivo come *unione dei vari sotto-linguaggi,* definendo lo scopo globale come $S \rightarrow S_1 \mid S_2 \mid \ldots \mid S_n$

Assumendo $S_1 \equiv$ A:

$A \rightarrow A\ c \mid A\ d \mid c$

Assumendo $S_2 \equiv$ B:

$B \rightarrow B\ d \mid d$

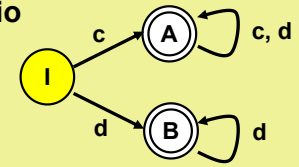
Linguaggio complessivo:

$S \rightarrow S_1 \mid S_2 = A \mid B$

!! che *NON È* semplicemente

$S \rightarrow S\ c \mid S\ d \mid S\ d \mid c \mid d$

Esempio



Questo equivarrebbe a unificare i due stati A e B, con ciò accettando anche stringhe come dc, escluse dall'automa

PIÙ IN GENERALE: (2/2)

Nel caso bottom-up, *in presenza di più stati finali:*
- si assume come (sotto-)scopo S_k uno stato finale per volta
- si scrivono le regole bottom-up corrispondenti
- si esprime il linguaggio complessivo come *unione dei vari sotto-linguaggi,* definendo lo scopo globale come $S \rightarrow S_1 \mid S_2 \mid \ldots \mid S_n$

Linguaggio complessivo:

$S \rightarrow A \mid B$

$A \rightarrow A\ c \mid A\ d \mid c$

$B \rightarrow B\ d \mid d$

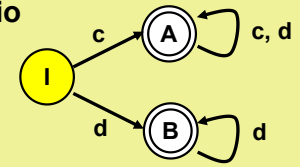
ovvero, sostituendo:

$S \rightarrow A\ c \mid A\ d \mid B\ d \mid c \mid d$

$A \rightarrow A\ c \mid A\ d \mid c$

$B \rightarrow B\ d \mid d$

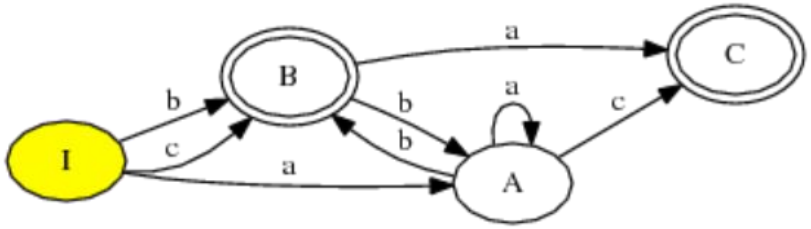
Esempio



La forma corretta
(non genera stringhe come dc, escluse dall'automa)

Esercizio (15 giugno 2017)

- Si consideri il seguente automa, dove lo stato **I** è lo stato iniziale e **B** e **C** sono stati finali:



- Si scriva una grammatica regolare lineare a sinistra che genera il linguaggio riconosciuto dall'automa.
- La stringa **caa** appartiene al linguaggio generato dalla grammatica?
- Si mostri il funzionamento dell'automa su tale stringa.

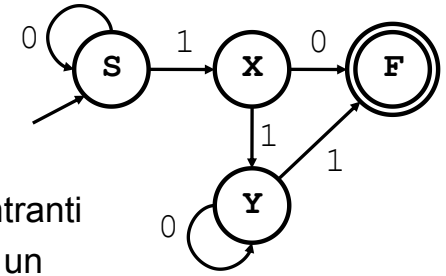
Da Automa a Grammatica lin Dx

- Quando abbiamo costruito i riconoscitori top-down, abbiamo aggiunto uno stato finale

$S \rightarrow 1\ X \mid 0\ S$

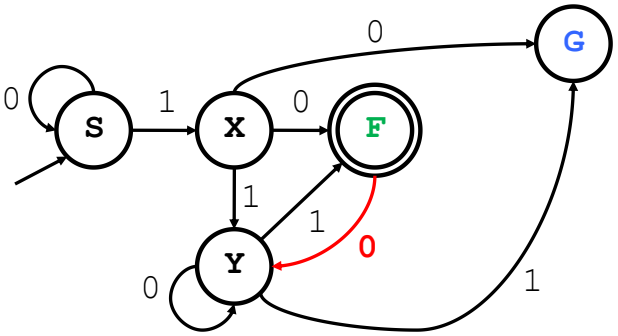
$X \rightarrow 1\ Y \mid 0$

$Y \rightarrow 1 \mid 0\ Y$



- Questo automa ha un solo stato finale, con soli archi entranti
- Quindi, dato un RSF che ha un solo stato finale e questo ha solo archi entranti, si possono ri-applicare "all'indietro" le regole viste e si ottiene la grammatica top-down
- Dato un RSF in cui lo stato finale ha archi uscenti, bisogna modificare leggermente la procedura

Da Automa a Grammatica lin Dx



In questo caso, si può duplicare lo stato finale:

- una versione (G) che funge da stato finale, senza archi uscenti
- una versione (F) che funge da stato generico

```
S → 1X | 0S
X → 1Y | 0 | 0F
Y → 1 | 0Y | 1F
F → 0Y
```

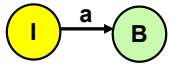
Questo equivale ad applicare, per lo stato finale,
• sia la regola per lo stato finale,
• sia quella per lo stato generico

In caso ci siano più stati finali, si duplicano tutti

OSSERVAZIONE

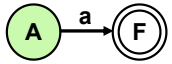
- **Il mapping tratta in modo particolare lo stato finale (nel caso top-down) o, rispettivamente, iniziale (nel caso bottom-up)**
- **Ciò garantisce che le produzioni *non presentino mai* ε *-rules.***
- **Alternativa: trattare tali casi nel modo standard** (senza eliminare, quindi, lo stato finale F o, rispettivamente, lo stato iniziale I) **ma accettando la presenza di produzioni *con* ε *-rules.***

Caso iniziale:



Bottom-up: I a ← B
ε ← I

Caso finale:



Top-down: A → a F
F → ε

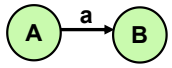
VANTAGGIO:
un solo tipo di mapping, valido sia per il caso generale, sia per i casi "particolari"

SVANTAGGIO:
produzioni con ε -rules, da eliminare eventualmente in un secondo tempo

Riassumendo

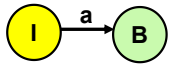
- Dato un RSF, si possono ottenere una grammatica regolare lineare a sinistra e una lineare a destra
- Le regole sono le stesse usate per ottenere l'automa data la grammatica, a parte
  - Nelle grammatiche lineari a destra, si duplica lo stato finale (o gli stati finali) qualora abbia archi uscenti
  - Nelle grammatiche lineari a sinistra,
    - si duplica lo stato iniziale qualora abbia archi entranti
    - in caso ci siano più stati finali, si crea una grammatica per ciascuno stato finale

Caso generico:



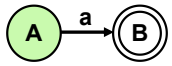
Top-down: A → a B
Bottom-up: A a ← B

Caso iniziale:



Top-down: S → a B (S ≡ I
Bottom-up: a ← B

Caso finale:



Top-down: A → a
Bottom-up: A a ← S (S ≡ B)

In alternativa, si possono usare ε-rules

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 3.4

Riconoscitori a Stati Finiti

Riconoscitori a Stati Finiti Non Deterministici

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

IMPLEMENTAZIONE DI RSF DETERMINISTICI

Un *riconoscitore a stati finiti deterministico* è facilmente realizzabile in un linguaggio imperativo, ma le scelte che si fanno possono influenzare fortemente l'estendibilità, la modularità e la riusabilità.

Esempi:

- **Ciclo `while` con serie di `if` annidati**
  - automa *cablato nel codice*
  - estendibilità nulla, leggibilità scarsa
- **Ciclo `while` con selezioni su due livelli**
  - automa ancora *cablato nel codice*
  - più struttura, estendibilità simile, leggibilità un po' migliore
- **Ciclo `while` con tabella separata**
  - automa *non più cablato nel codice*
  - estendibilità superiore, leggibilità ottima
- **Altre possibilità (da esplorare…)**
  - un oggetto configurabile
  - un automa in grado di "simulare" ogni automa a stati
  - …

IMPLEMENTAZIONE – 1° caso

- **Iª possibilità: ciclo `while` con serie di `if` annidati**
  - **automa *cablato nel codice***
  - **estendibilità nulla, leggibilità scarsa**

```
enum Stato {S0, S1, S2, S3  };
Stato stato = S0;
int curPos = 0;
while (curPos < strlen(frase))
{  char currentChar = frase[curPos];  curPos++;
   if (stato==S0 && currentChar=='a') stato=S1; else
   if (stato==S0 && currentChar=='b') stato=S3; else
   if (stato==S1 && currentChar=='a') stato=S2; else
   if (stato==S1 && currentChar=='b') stato=S0; else
   if (stato==S2 && currentChar=='a') stato=S2; else
   if (stato==S2 && currentChar=='b') stato=S0; else
   if (stato==S3 && currentChar=='a') stato=S3; else
   if (stato==S3 && currentChar=='b') stato=S3;
}
```

stato corrente

posizione corrente dell'input

automa cablato nel codice

IMPLEMENTAZIONE – 2° caso

- **IIª possibilità: ciclo `while` con selezione su due livelli**
  - **automa ancora *cablato nel codice***
  - **più struttura, estendibilità simile, leggibilità un po' migliore**

```
while (curPos < strlen(frase))
{  char currentChar = frase[curPos];  curPos++;
   if (stato==S0)
      {   if (currentChar=='a') stato=S1; else
          if (currentChar=='b') stato=S3;
      } else
   if (stato==S1)
      {  if (currentChar=='a') stato=S2; else
         if (currentChar=='b') stato=S0;
      } else
      ...
   if (stato==S3)
      { if (currentChar=='a') stato=S3; else
        if (currentChar=='b') stato=S3
      }
}
```

automa cablato nel codice

PERÒ comincia a nascere una *struttura*, più chiara e forse riusabile

IMPLEMENTAZIONE – 3° caso

- **IIIª possibilità: ciclo `while` con tabella separata**
  - **automa *non più cablato nel codice***
  - **estendibilità superiore, leggibilità ottima**

	a	b
S0	S1	S3
S1	S2	S0
S2	S2	S0
S3	S3	S3

```
char alfabeto[] = {'a', 'b'};
int tabellaTransizioni[/* stato */][/* char input */] =
{     {S1,S3},
      {S2,S0},
      {S2,S0},
      {S3,S3}
};
int stato = S0;
int curPos = 0;
while (curPos < strlen(frase))
{  char curChar = frase[curPos];
   int pos= ricerca(alfabeto,curChar);
   stato = tabellaTransizioni[stato][pos];
   curPos++;
}
```

La descrizione dell'automa può essere letta dall'esterno

automa *non più* cablato nel motore !

la `ricerca` si può implementare in *O*(1)

indice (0 o 1) corrispondente al carattere corrente `curChar` nell'array `alfabeto`

AUTOMI NON DETERMINISTICI

Certe grammatiche possono portare a un *automa non deterministico,* cioè nella cui tabella di transizioni compaiono *più stati futuri per una stessa configurazione.*

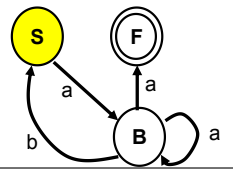
ESEMPIO

Sia G una grammatica lineare a destra caratterizzata dalle produzioni:

S → a B

B → a B | b S | a

Automa risultante:



In corrispondenza dello stato B, l'automa, con ingresso a, deve «sapere» se è meglio portarsi in B o in F.

È un automa *non deterministico:* per ogni frase di L(G) esiste *almeno una computazione* che porta l'automa dallo stato iniziale S allo stato finale F.

La corrispondente **tabella di transizioni** (**completata con uno *stato di errore E*** per gli ingressi non previsti) è la seguente:

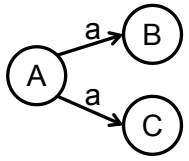
	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

In generale

- Data una grammatica regolare lineare a destra con produzioni

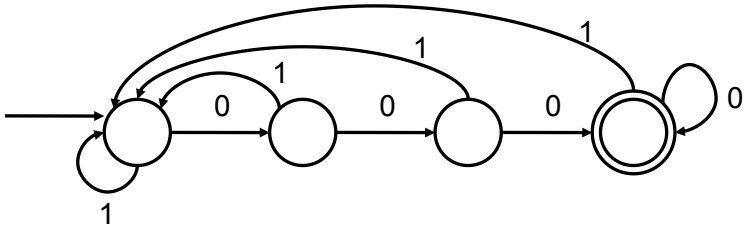
  A → aB | aC | …
- oppure una grammatica regolare lineare a sinistra con produzioni

  A → Ba | Ca | …
- l'automa risulta nondeterministico



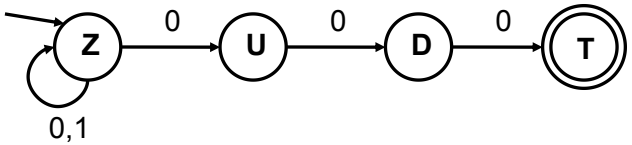
Esercizio

- Si scriva un automa riconoscitore a stati finiti per riconoscere il linguaggio sull'alfabeto {0,1}:
- Insieme di tutte le stringhe che finiscono in 000

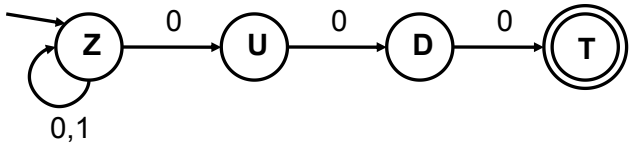


Automi a stati finiti non deterministici (NFA)

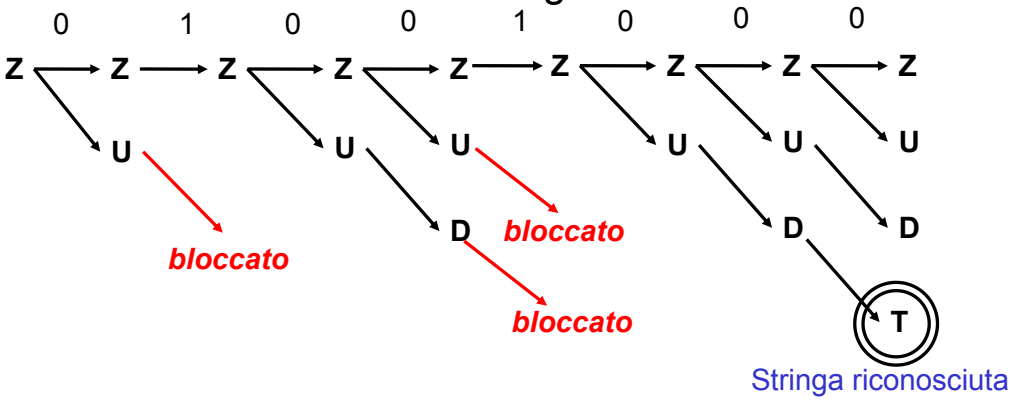
- Un NFA può essere in vari stati nello stesso momento,
- oppure, visto in un altro modo, può "scommettere" su quale sarà il prossimo stato
- Esempio: un automa che accetta tutte e solo le stringhe che finiscono in 000.



Automi a stati finiti non deterministici (NFA)



- Elaborazione della stringa 01001000



Stringa riconosciuta

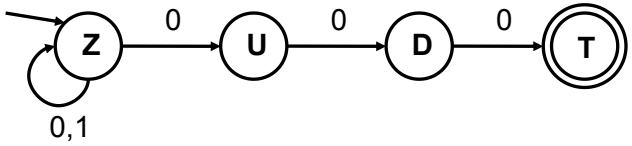
Definizione formale di NFA

Formalmente, un automa riconoscitore a stati finiti non deterministico è una quintupla

$$\langle \mathtt{A}, \mathtt{S}, \mathtt{S}_0, \mathtt{F}, \mathtt{sfn} \rangle$$

dove

- **A = alfabeto**
- **S = insieme degli stati**
- **S_0 = stato iniziale $\in$ S**
- **F = insieme degli *stati finali* $\subseteq$ S**
- **sfn: $S \times A \rightarrow \wp(S)$** *(state function)*



	0	1
Z	{Z,U}	{Z}
U	{D}	∅
D	{T}	∅
T	∅	∅

ACCETTAZIONE

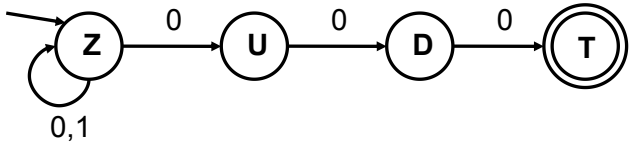
Un riconoscitore a stati finiti nondeterministico (NFA) accetta una stringa
$w = a_1 a_2 \ldots a_n$

se *esiste un cammino* nel diagramma di transizione che

1. Inizia nello stato iniziale
2. Finisce in uno stato finale (di accettazione)
3. Ha una sequenza di etichette $a_1\ a_2\ \ldots\ a_n$

Identica al caso deterministico

Ad esempio, l'automa sotto riconosce la stringa 01001000



Funzione di transizione estesa sfn*

La funzione di accettazione *sfn*, che opera su coppie (*stato, simbolo*),

può essere estesa ad una funzione *sfn**, che opera su coppie (*stato*, *stringa*)

$sfn^*(q,\varepsilon) = \{q\}$

$$sfn^*(q, sc) = \bigcup_{p \in sfn^*(q,s)} sfn(p, c)$$

($s \in A^*$ è una stringa, $c \in A$ è un simbolo)

Formalmente, il linguaggio accettato dall'automa è

$$L(R) = \{w : sfn^*(S_0, w) \cap F \neq \emptyset\}$$

RICONOSCITORI NON DETERMINISTICI

Un automa *riconoscitore non deterministico* dev'essere *intrinsecamente dotato della capacità di scegliere* in quale stato portarsi, quando ha più alternative.

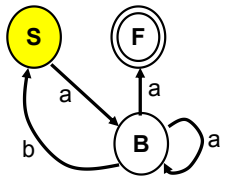
Nell'automa a lato:

- la frase `abaa` è riconosciuta dalla sequenza di transizioni `S → B → S → B → F`
- la frase `abaaa` è riconosciuta invece dalla sequenza `S → B → S → B → B → F`

Nel primo caso, dallo stato `B` con ingresso `a` l'automa deve "scegliere" di portarsi in `F`, mentre nel secondo caso deve "scegliere" di restare in `B` (la prima volta) e successivamente di portarsi in `F` (la seconda volta).

- la frase `aaba` **è invece sbagliata** e non viene riconosciuta (`S → B → B → S → B`)

Esempio:



	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

RICONOSCITORI NON DETERMINISTICI

Se il linguaggio supporta il non determinismo, *implementare il riconoscitore è semplice*

Grammatica:

```
S → a B
B → a B | b S | a
```

In Prolog,

- ogni **produzione** diventa una **regola**
- la macchina virtuale del linguaggio è in grado di "tentare una strada" e *tornare indietro* nel caso si riveli sbagliata, provando via via *tutte le possibili alternative.*

Il programma Prolog

```
regolaS([a|B]) :- regolaB(B).

regolaB([a|B]) :- regolaB(B).
regolaB([b|S]) :- regolaS(S).
regolaB([a]).
```

UN RICONOSCITORE NON-DETERMINISTICO IN PROLOG

Ora basta interrogare il sistema Prolog e vedere cosa risponde:

```
?- regolaS([a,a,b,a]).          % aaba è sbagliata
no
?- regolaS([a,a,b,a,a]).        % aabaa è corretta
yes
?- regolaS([a,b,a,a]).          % abaa è corretta
yes
```

Il programma Prolog

```
regolaS([a|B]) :- regolaB(B).

regolaB([a|B]) :- regolaB(B).          % regola 1
regolaB([b|S]) :- regolaS(S).          % regola 2
regolaB([a]).                          % regola 3
```

FUNZIONAMENTO DELLA MACCHINA VIRTUALE PROLOG

```
?- regolaS([a,a,b,a,a]).
1 1 Call: regolaS([a,a,b,a,a]) ?
2 2 Call: regolaB([a,b,a,a]) ?       % usa regola 1
3 3 Call: regolaB([b,a,a]) ?         % usa regola 1
4 4 Call: regolaS([a,a]) ?
5 5 Call: regolaB([a]) ?             % usa regola 1
6 6 Call: regolaB([]) ?              % TENTA, MA FALLISCE..
6 6 Fail: regolaB([]) ?              % ... E TORNA INDIETRO
5 5 Exit: regolaB([a]) ?             % usa regola 3 -> OK
4 4 Exit: regolaS([a,a]) ?
3 3 Exit: regolaB([b,a,a]) ?
2 2 Exit: regolaB([a,b,a,a]) ?
1 1 Exit: regolaS([a,a,b,a,a]) ?
yes
```

E se il linguaggio non supporta il non-determinismo ?

RICONOSCITORI NON DETERMINISTICI CON LINGUAGGI IMPERATIVI

Un riconoscitore non-deterministico

- **è meno efficiente di un riconoscitore deterministico**
- **deve disporre di strutture dati interne per «ricordare la strada fatta» e poterla *disfare* se necessario per *esplorarne un'altra***

Come può essere costruito se il linguaggio non supporta il non-determinismo ?

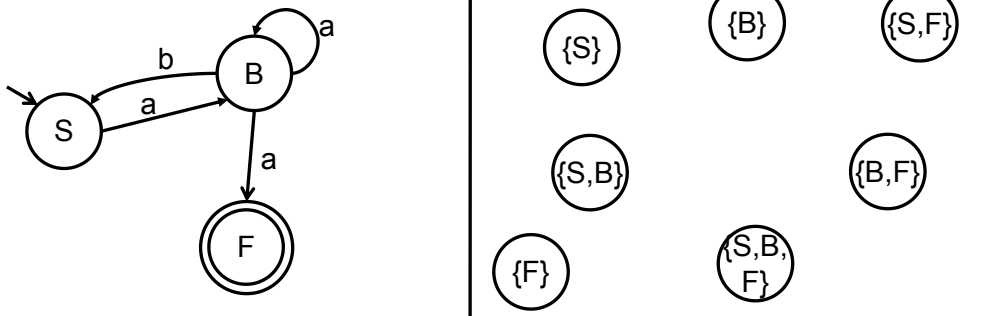
In un linguaggio imperativo, occorre *costruirsi a mano tutte le strutture per "ricordare la strada" e mettere in piedi il "motore" per gestirle*

… cioè ricostruirsi la stessa capacità che il motore Prolog *ha già innata.*

Da NFA e DFA

- Sorprendentemente, ogni NFA può essere convertito in un DFA che riconosce lo stesso linguaggio
- L'NFA "si trova contemporaneamente in più stati", quindi possiamo associare ad ogni possibile insieme di stati di un NFA, uno stato di un DFA:

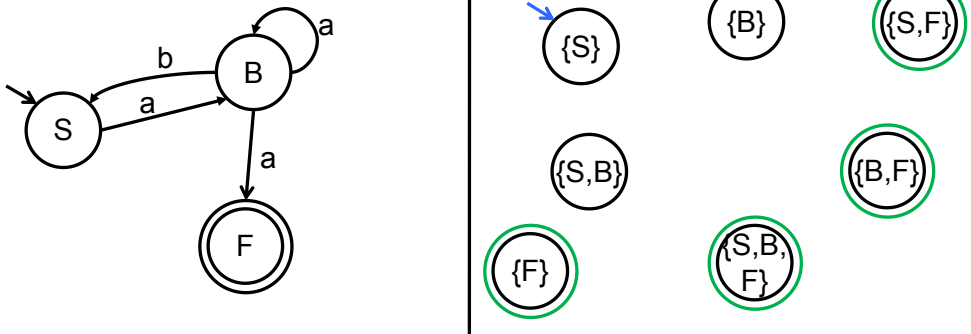
$NFA=\langle A, S_N, I_N, F_N, sfn_N\rangle$ $DFA=\langle A, S_D, I_D, F_D, sfn_D\rangle$



Da NFA e DFA

- Lo stato iniziale del DFA è lo stato che contiene solo lo stato iniziale del NFA: $I_D=\{I_N\}$
- Gli stati finali del DFA sono quelli che hanno intersezione non nulla con gli stati finali del NFA: $F_D = \{ X: X \subseteq S_N \text{ e } X \cap F_N \neq \emptyset\}$

$NFA=\langle A, S_N, I_N, F_N, sfn_N\rangle$ $DFA=\langle A, S_D, \{I_N\}, F_D, sfn_D\rangle$

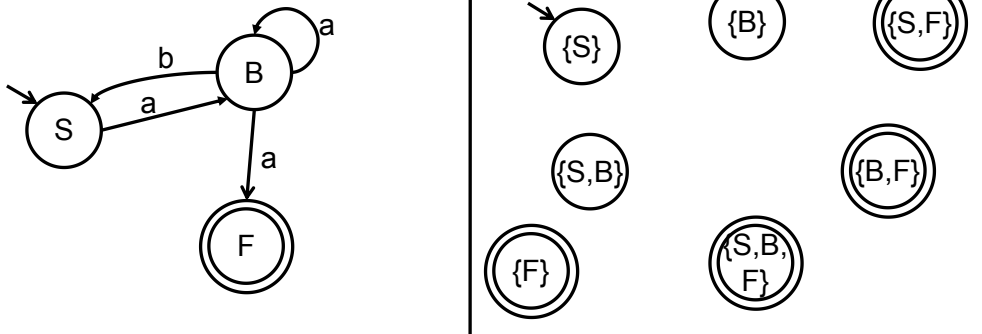


Da NFA e DFA

- La funzione di transizione sfn_D è costruita come segue:
- Per ogni stato $X \in S_D$ e carattere $\mathbf{a} \in \mathbf{A}$,

$$sfn_D(X,a) = \bigcup_{p \in X} sfn_N(p,a)$$

$NFA=\langle A, S_N, I_N, F_N, sfn_N\rangle$ $DFA=\langle A, S_D, I_D, F_D, sfn_D\rangle$



Da NFA e DFA

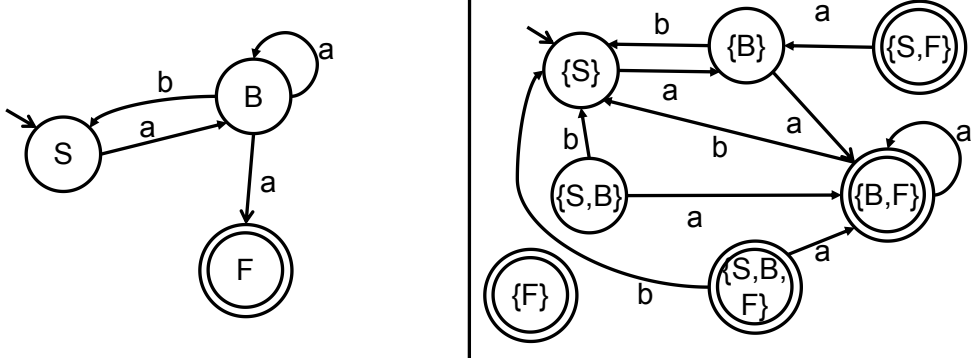
- La funzione di transizione sfn_D è costruita come segue:
- Per ogni stato $X \in S_D$ e carattere $\mathtt{a} \in \mathtt{A}$,

$$sfn_D(X,a) = \bigcup_{p \in X} sfn_N(p,a)$$

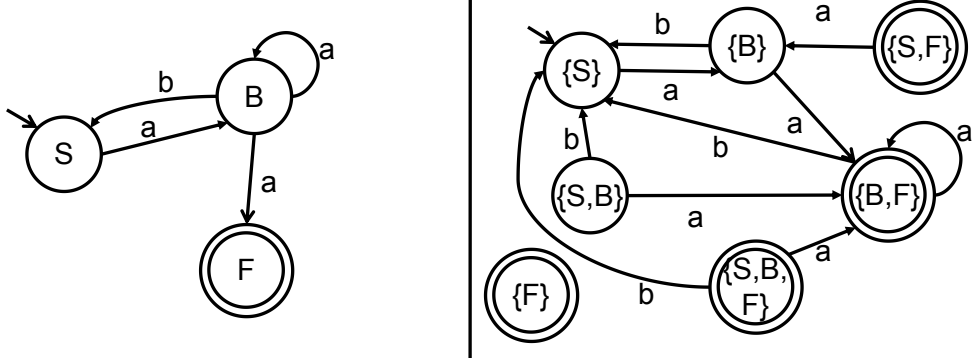
Da NFA e DFA

- L'automa DFA risultante ha quindi 2^n nodi (se l'automa NFA aveva n nodi), o 2^n-1 se si esclude il nodo che rappresenta l'insieme vuoto
- In realtà molti di questi nodi sono inutili, in quanto irraggiungibili dallo stato iniziale

$\mathtt{NFA} = \langle \mathtt{A}, \mathtt{S}_N, \mathtt{I}_N, \mathtt{F}_N, \mathtt{sfn}_N \rangle$ $\mathtt{DFA} = \langle \mathtt{A}, \mathtt{S}_D, \mathtt{I}_D, \mathtt{F}_D, \mathtt{sfn}_D \rangle$

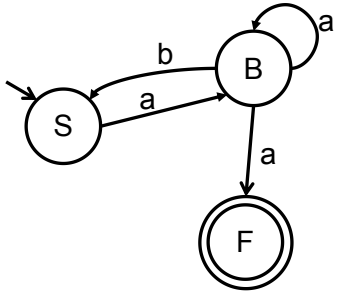


$\mathtt{NFA} = \langle \mathtt{A}, \mathtt{S}_N, \mathtt{I}_N, \mathtt{F}_N, \mathtt{sfn}_N \rangle$ $\mathtt{DFA} = \langle \mathtt{A}, \mathtt{S}_D, \mathtt{I}_D, \mathtt{F}_D, \mathtt{sfn}_D \rangle$



Da NFA e DFA

- Per costruire il DFA evitando stati irraggiungibili, si crea la tabella di transizione sfn_D in maniera incrementale:
- Si inserisce lo stato iniziale nella tabella della sfn_D, con la riga corrispondente
- Se nella tabella c'è una transizione verso uno stato D che non compare nell'insieme degli stati, si aggiunge D e la riga corrispondente; poi si itera

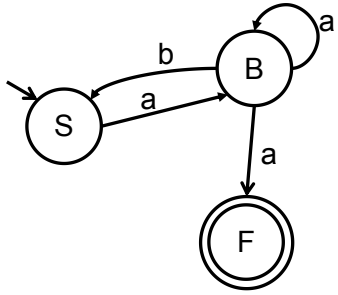


sfn_D

Stato	a	b
{S}	{B}	Ø

Da NFA e DFA

- Per costruire il DFA evitando stati irraggiungibili, si crea la tabella di transizione sfn_D in maniera incrementale:
- Si inserisce lo stato iniziale nella tabella della sfn_D, con la riga corrispondente
- Se nella tabella c'è una transizione verso uno stato D che non compare nell'insieme degli stati, si aggiunge D e la riga corrispondente; poi si itera



Da NFA a DFA

Teorema: Sia D un automa deterministico ottenuto con la costruzione precedente da un automa nondeterministico N. Allora L(D)=L(N)

Dim: Il linguaggio L(X) è dato dalla $sfn^*_X()$ a partire dallo stato iniziale I. Per induzione sulla lunghezza della stringa da riconoscere

- $sfn^*_D(\{I\},\varepsilon) = sfn^*_N(I,\varepsilon)$
- supponiamo che $sfn^*_D(\{I\},x)=sfn^*_N(I,x)$ e dimostriamo che $sfn^*_D(\{I\},xc) = sfn^*_N(I,xc)$.

$sfn^*_D(\{I\},xc) = sfn_D\,(sfn^*_D(\{I\},x),c)$ *definizione di* sfn^*_D

$sfn^*_D(\{I\},xc) = sfn_D\,(sfn^*_N(I,x),c)$ *ipotesi induttiva*

$sfn^*_D(\{I\},xc) = \bigcup_{p \in sfn^*_N(I,x)} sfn_N\,(p,c)$ *def di* sfn_D

$sfn^*_D(\{I\},xc) = sfn^*_N(\{I\},xc)$ *definizione di* sfn^*_N

$S \rightarrow Ba \mid Sa \mid b$

$B \rightarrow Bb \mid Cb \mid Cc$

$C \rightarrow Ca \mid b \mid Bc \mid Sc$

- Si disegni l'automa riconoscitore del linguaggio generato.

Esercizio (29 giugno 2016)

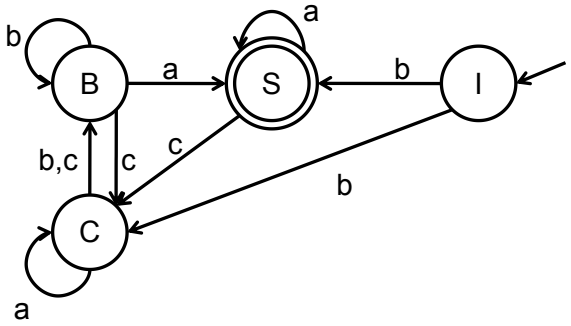
- Si consideri la seguente grammatica:

  $S \rightarrow Ba \mid Sa \mid b$

  $B \rightarrow Bb \mid Cb \mid Cc$

  $C \rightarrow Ca \mid b \mid Bc \mid Sc$
- Si disegni l'automa riconoscitore del linguaggio generato. L'automa è deterministico? Se non lo è, si disegni un automa deterministico equivalente, evidenziando gli stati iniziali e finali.

- L'automa è deterministico?
- Se non lo è, si disegni un automa deterministico equivalente



UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 3.5

Riconoscitori a Stati Finiti

Minimizzazione ed equivalenza di Automi a Stati Finiti

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

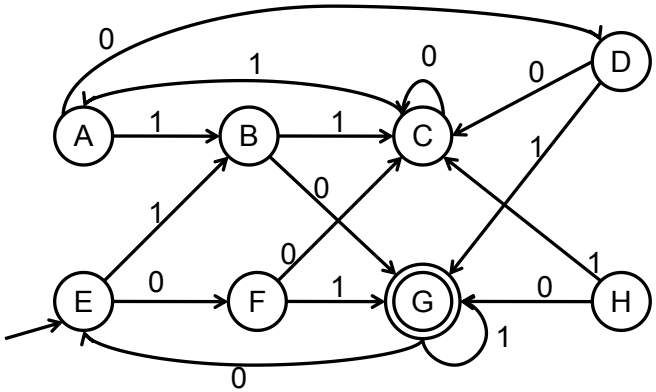
Prof. MARCO GAVANELLI

Stati equivalenti

- Dato un ASF = $\langle A, S, S_0, F, sfn \rangle$
- Due stati p e $q \in S$ sono equivalenti, scritto $p \equiv q$, se per ogni parola w
  - da p si riconosce la parola $\Leftrightarrow$ da q si riconosce la parola
- Formalmente, $p \equiv q$ se

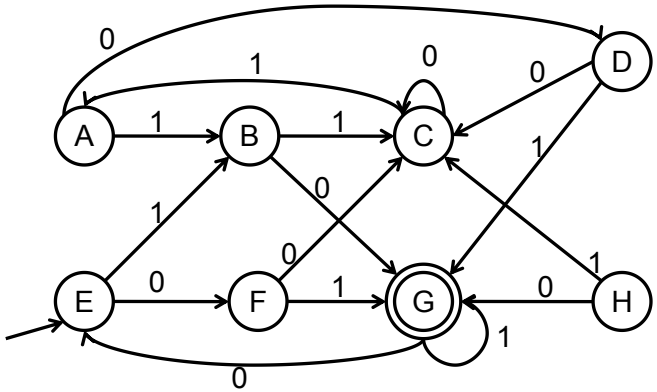
  $\forall w \in A^* \quad sfn^*(p,w) \in F \Leftrightarrow sfn^*(q,w) \in F$
- Viceversa, due stati p e q sono distinguibili , scritto $p \not\equiv q$ se esiste una parola w tale che $sfn^*(p,w) \in F$ ma $sfn^*(q,w) \notin F$ o viceversa

Esempio



- Gli stati A ed E sono equivalenti?

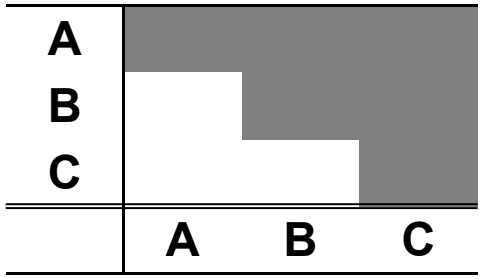
Esempio

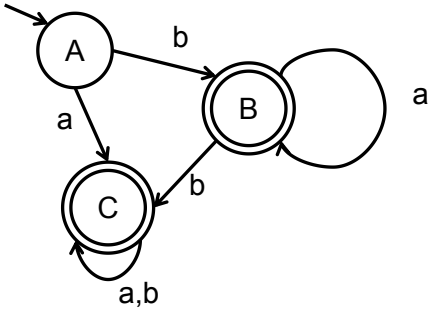


- Gli stati A ed F sono equivalenti?

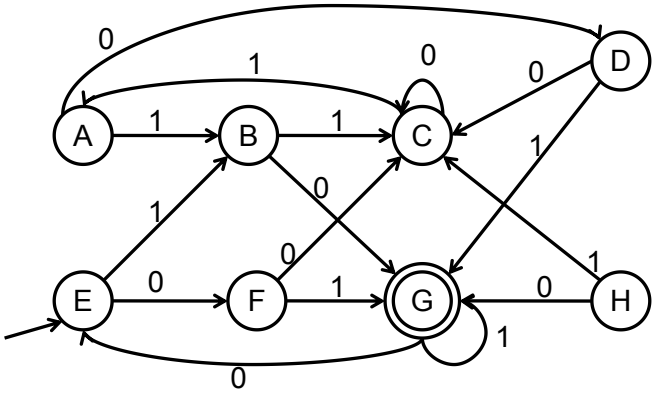
Algoritmo induttivo

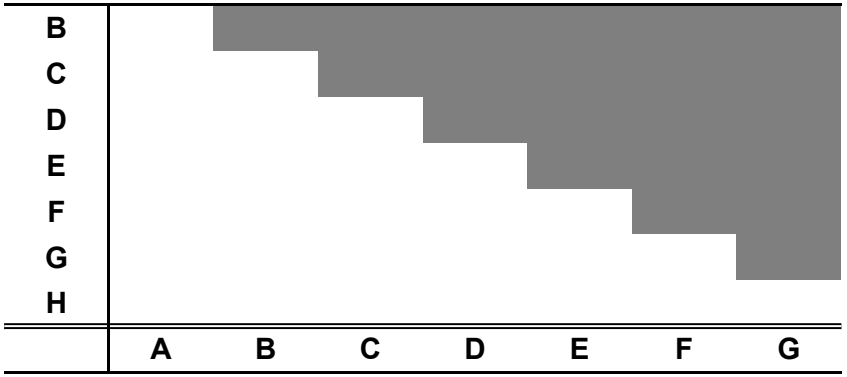
- Caso base: se $p \in F$ e $q \notin F$, allora $p \not\equiv q$
- Induzione: se $\exists a \in A$ $sfn(p,a) \not\equiv sfn(q,a)$, allora $p \not\equiv q$





Esempio





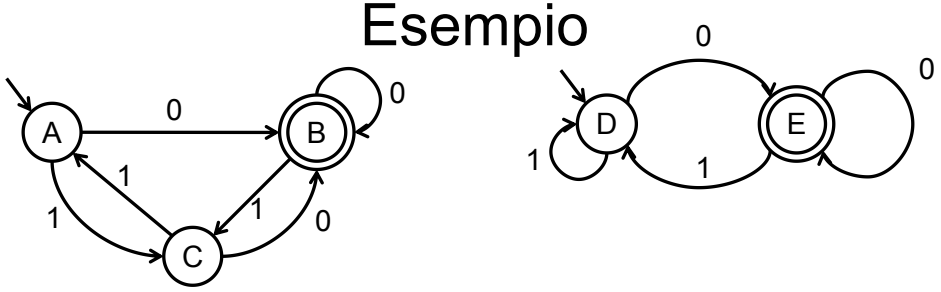
Nota

- Nella relazione di equivalenza fra stati, non è importante quale stato è iniziale (lo stato iniziale può essere equivalente ad uno stato che non è iniziale), mentre è fondamentale distinguere fra stati finali (di accettazione) e non

Equivalenza di FSA

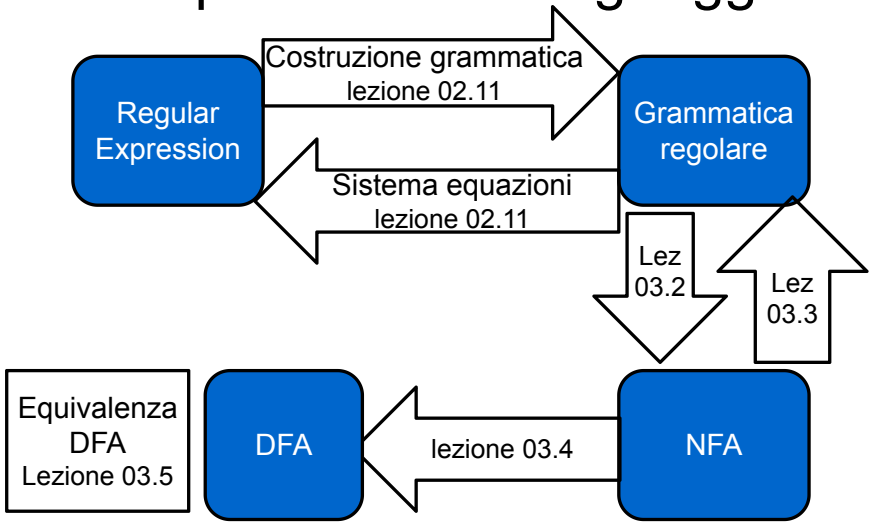
- Per verificare l'equivalenza di due automi a stati finiti, si possono unire i due automi e verificare se i due stati iniziali sono equivalenti
- Se lo sono, significa che tutte le parole riconosciute dal primo automa sono riconosciute anche dal secondo e viceversa
- Quindi riconoscono lo stesso linguaggio.

Esempio



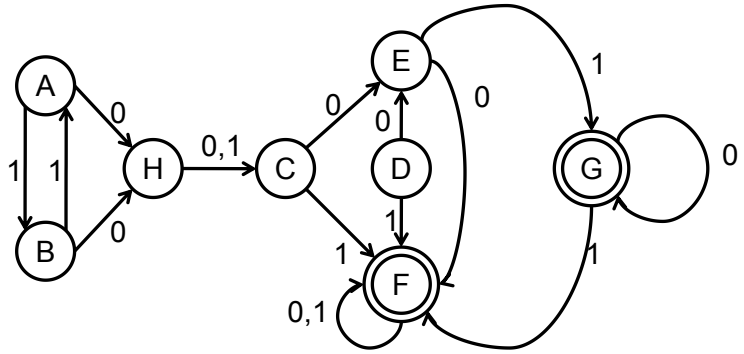
	A	B	C	D
B				
C				
D				
E				

Equivalenza di linguaggi



- Quindi con questo procedimento possiamo verificare anche se due grammatiche o due regular expression generano lo stesso linguaggio

Esercizio



- Trovare l'automa minimo

Da AUTOMI NON DETERMINISTICI ad AUTOMI DETERMINISTICI

Tabella nuovo automa:

	a	b
[S]	[B]	[E]
[B]	[B,F]	[S]
[B,F]	[B,F,E]	[S,E]
[S,E]	[B,E]	[E]
[B,E]	[B,F,E]	[S,E]
[B,F,E]	[B,F,E]	[S,E]
[E]	[E]	[E]

Rinominata:

	a	b
S0	S1	S6
S1	S2	S0
S2	S5	S3
S3	S4	S6
S4	S5	S3
S5	S5	S3
S6	S6	S6

Attenzione: gli **stati finali** (cui è idealmente associata uscita OK) sono **sempre distinti dagli stati non-finali** (cui è idealmente associata uscita NO)

Minimizzazione:

S1	S1/S2,S0/S6					
S2	X	X				
S3	S1/S4	S2/S4,S0/S6	X			
S4	S1/S5,S3/S6	S2/S5,S0/S3	X	S4/S5,S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6,S0/S6	X	S4/S6	S5/S6,S3/S6	X
	S0	S1	S2	S3	S4	S5

Da AUTOMI NON DETERMINISTICI ad AUTOMI DETERMINISTICI

Tabella nuovo automa:

	a	b
[S]	[B]	[E]
[B]	[B,F]	[S]
[B,F]	[B,F,E]	[S,E]
[S,E]	[B,E]	[E]
[B,E]	[B,F,E]	[S,E]
[B,F,E]	[B,F,E]	[S,E]
[E]	[E]	[E]

Rinominata:

	a	b
S0	S1	S6
S1	S2	S0
S2	S5	S3
S3	S4	S6
S4	S5	S3
S5	S5	S3
S6	S6	S6

Minimizzata:

	a	b
S0	S1	S6
S1	S2	S0
S2	S2	S0
S6	S6	S6

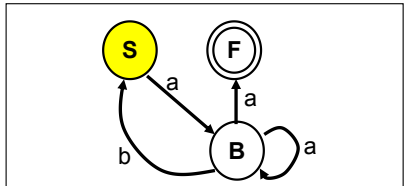
Minimizzazione:

S1	S1/S2,S0/S6					
S2	X	X				
S3	S1/S4	S2/S4,S0/S6	X			
S4	S1/S5,S3/S6	S2/S5,S0/S3	X	S4/S5,S3/S6		
S5	X	X		X	X	
S6	S1/S6	S2/S6,S0/S6	X	S4/S6	S5/S6,S3/S6	X
	S0	S1	S2	S3	S4	S5

Stati indistinguibili:
- **S2/S5 (finali)**
- **S0/S3, S1/S4**

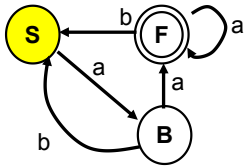
Da AUTOMI NON DETERMINISTICI ad AUTOMI DETERMINISTICI

Automa non deterministico:



	a	b
S	B	E
B	B/F	S
F	E	E
E	E	E

Automa deterministico minimo:



	a	b
S0	S1	S6
S1	S2	S0
S2	S2	S0
S6	S6	S6

	a	b
S	B	E
B	F	S
F	F	S
E	E	E

Stati rinominati

Il nuovo automa riconosce anch'esso le due frasi
`abaa` (sequenza di transizioni: `S` → `B` → `S` → `B` → `F`)
`abaaa` (sequenza di transizioni: `S` → `B` → `S` → `B` → `F` → `F`)

IMPLEMENTARE QUESTO AUTOMA

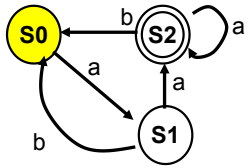
Una volta tradotto in grammatica, l'automa deterministico (minimo) può essere implementato sia in un linguaggio imperativo, sia, *più rapidamente,* in un linguaggio dichiarativo come il Prolog:

Il programma Prolog (base)

```
s0([a|S1]) :- s1(S1).

s1([a|S2]) :- s2(S2).
s1([b|S0]) :- s0(S0).

s2([]).
s2([a|S2]) :- s2(S2).
s2([b|S0]) :- s0(S0).
```



Grammatica corrispondente:

```
S0 → a S1
S1 → a S2 | b S0
S2 → a S2 | b S0 | ε
```

Qui S2 (finale) compare in altre produzioni, quindi il caso ε va previsto esplicitamente.

IMPLEMENTARE QUESTO AUTOMA

L'uso di un linguaggio che supporta il non-determinismo come il Prolog *non implica che si debba necessariamente pagare il prezzo del non-determinismo* anche quando non è necessario.

L'indexing di Prolog in questo caso riesce ad evitare di aprire punti di scelta

Il programma Prolog (base)

```
s0([a|S1]) :- s1(S1).

s1([a|S2]) :- s2(S2).
s1([b|S0]) :- s0(S0).

s2([]).
s2([a|S2]) :- s2(S2).
s2([b|S0]) :- s0(S0).
```

Il programma Prolog (variato)

```
s0([a|S1]) :- s1(S1).

s1([a|S2]) :- !, s2(S2).
s1([b|S0]) :- !, s0(S0).

s2([]).
s2([a|S2]) :- !, s2(S2).
s2([b|S0]) :- !, s0(S0).
```

Se ho un compilatore vecchio stile, il predicato CUT (`!`) indica alla macchina virtuale Prolog di *tagliare le alternative* perché non saranno necessarie.

DAL RICONOSCITORE AL GENERATORE

Un linguaggio dichiarativo come il Prolog ha anche un altro vantaggio: lo stesso programma può essere usato *sia da riconoscitore sia da generatore* (con alcune avvertenze).

Interrogando il sistema *con una stringa fissata*, riconosce:

```
?- s0([a,a,b,a]).          % aaba è sbagliata
no
?- s0([a,a,b,a,a]).        % aabaa è corretta
yes
?- s0([a,b,a,a]).          % abaa è corretta
yes
```

Interrogandolo *con una variabile*, genera via via le diverse frasi:

```
?- s0(X).
  X = [a,a] ? ;
  X = [a,a,a] ? ;
  X = [a,a,a,a] ?
  ...
```

Poiché le frasi possibili sono infinite, l'ordine con cui le genera *dipende da come il motore Prolog esplora le regole !*

DAL RICONOSCITORE AL GENERATORE

Il programma Prolog (base):

```
s0([a|S1]) :- s1(S1).

s1([a|S2]) :- s2(S2).
s1([b|S0]) :- s0(S0).

s2([]).
s2([a|S2]) :- s2(S2).
s2([b|S0]) :- s0(S0).
```

Per scelta, la macchina virtuale Prolog esplora le regole *nell'ordine testuale in cui sono scritte.*

1ª CONSEGUENZA: è fondamentale che *la regola che chiude la ricorsione sia la prima,* per garantire che la computazione termini!

Interrogandolo *con una variabile*, genera via via le diverse frasi:

```
?- s0(X).
  X = [a,a] ? ;
  X = [a,a,a] ? ;
  X = [a,a,a,a] ?
  ...
```

2ª CONSEGUENZA: se una regola dà luogo a *infinite alternative* (come qui), le regole successive *potrebbero non essere mai esplorate*
→ le frasi con `b` in realtà non appaiono !

DAL RICONOSCITORE AL GENERATORE

Il programma Prolog (base):

```
s0([a|S1]) :- s1(S1).

s1([a|S2]) :- s2(S2).
s1([b|S0]) :- s0(S0).

s2([]).
s2([a|S2]) :- s2(S2).
s2([b|S0]) :- s0(S0).
```

E` comunque semplice cambiare l'ordine in cui le soluzioni vengono presentate.
Predicato length predefinito:

```
length([],0).
length([_|L],N):-
        length(L,N1),
        N is N+1.
```

```
?- length(X,_),s0(X).
  X = [a, a] ;
X = [a, a, a] ;
X = [a, a, a, a] ;
X = [a, b, a, a] ;
X = [a, a, a, a, a] ;
X = [a, a, b, a, a] ;
X = [a, b, a, a, a] ;
X = [a, a, a, a, a, a] ;
X = [a, a, a, b, a, a] ;
X = [a, a, b, a, a, a] ;
X = [a, b, a, a, a, a] ;
X = [a, b, a, b, a, a] ;
X = [a, a, a, a, a, a, a] ;
X = [a, a, a, a, b, a, a] ;
...
```

ESPRESSIONI E LINGUAGGI REGOLARI

TEOREMA:
l'insieme dei linguaggi riconosciuti da un ASF coincide con l'insieme dei linguaggi regolari, ossia quelli descritti da *espressioni regolari.*

***Espressioni regolari* e *automi a stati finiti* sono *metodi descrittivi* appartenenti a due differenti categorie**

- **Gli *automi a stati* sono un metodo di descrizione *operazionale,*** in quanto **evidenziano i *passi computazionali* da compiere per riconoscere le *frasi* –** che quindi sono descritte tramite *le operazioni* necessarie a riconoscerle.
- **Le *espressioni regolari* sono invece un metodo di descrizione *denotazionale,*** in cui un'entità è specificata tramite operatori (funzioni) di tipo matematico.

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 3.6

Riconoscitori a Stati Finiti

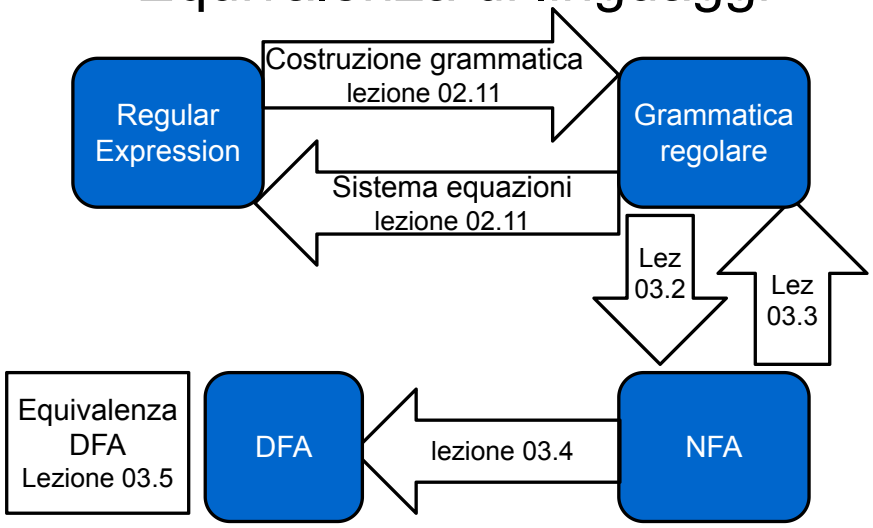
Da Espressione Regolare a Riconoscitore a Stati Finiti

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

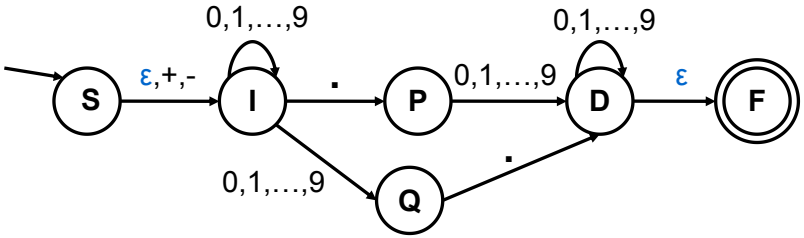
Equivalenza di linguaggi



- Quindi con questo procedimento possiamo verificare anche se due grammatiche o due regular expression generano lo stesso linguaggio

NFA con transizioni ε

- Un altro modo per descrivere NFA è aggiungere la possibilità di avere ε-mosse
- L'idea è che in presenza di una ε-mossa, l'automa può *scegliere* se effettuare la mossa oppure no
- Siccome c'è una scelta, si tratta di un automa non deterministico
  - Si può immaginare che l'automa passi "contemporaneamente" in più stati
  - oppure si può immaginare che indovini sempre la scelta giusta

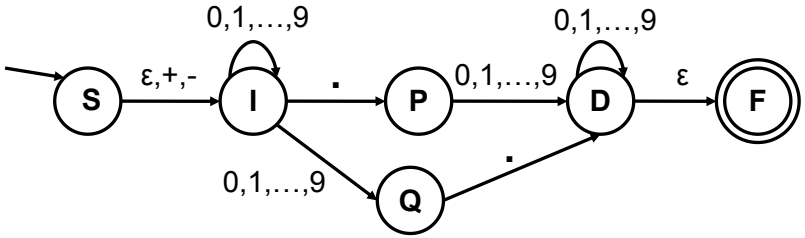


Esempio

Un ε-NFA che accetta numeri decimali consiste di:

1. Un segno + o -, opzionale
2. Una stringa di cifre decimali
3. un punto decimale
4. un'altra stringa di cifre decimali

Una delle stringhe (2) e (4) è opzionale



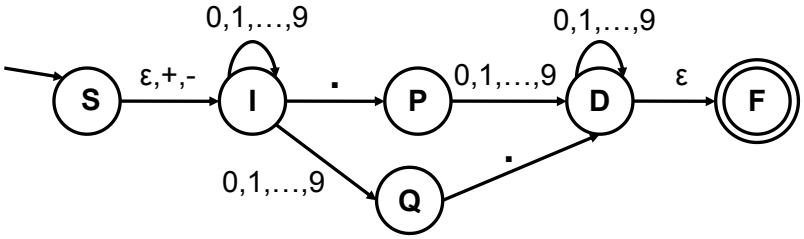
Definizione formale di NFA con ε-mosse

Formalmente, un automa riconoscitore a stati finiti non deterministico con ε-mosse è una quintupla

$$\langle A, S, S_0, F, sfn \rangle$$

dove

- **A = alfabeto**
- **S = insieme degli stati**
- **S_0 = stato iniziale $\in S$**
- **F = insieme degli *stati finali* $\subseteq S$**
- **$sfn: S \times (A \cup \{\varepsilon\}) \rightarrow \wp(S)$** *(state function)*



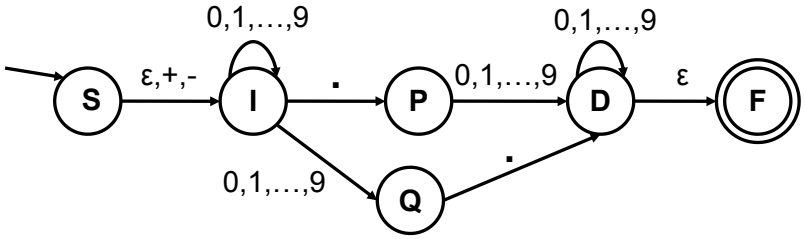
Esempio

- La tabella di transizione contiene quindi una colonna **ε**.
- Ovviamente la transizione

$$S \xrightarrow{\varepsilon} \{I\}$$

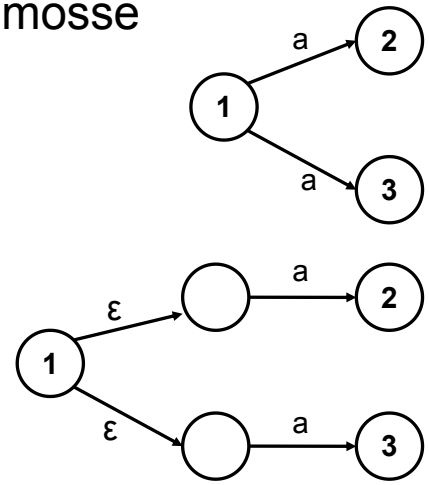
non significa ~~*"da S si va in I con input ε"*~~, bensì *"da S ci si può portare in I senza leggere alcun simbolo da input"*

	ε	+,-	.	0,…,9
S	{I}	{I}	∅	∅
I	∅	∅	{P}	{I,Q}
P	∅	∅	∅	{D}
Q	∅	∅	{D}	∅
D	{F}	∅	∅	{D}
F	∅	∅	∅	∅



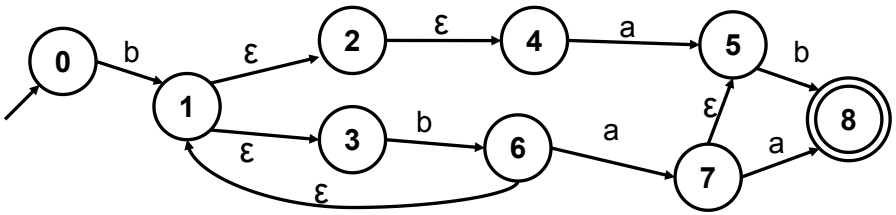
NFA e ε-NFA

- Qualunque NFA può sempre essere descritto da un automa in cui tutto il nondeterminismo è codificato nelle ε-mosse



da ε-NFA a FA

- Si definisce *ε-chiusura* di un nodo l'insieme degli stati che si possono raggiungere da quel nodo facendo solo ε-mosse
- Def. induttiva:
  - $q \in \varepsilon\text{-CLOS}(q)$
  - $p \in \varepsilon\text{-CLOS}(q)$ e $r \in sfn^*(q, \varepsilon) \rightarrow r \in \varepsilon\text{-CLOS}(q)$



da ε-NFA a FA

Dato un ε-NFA

$$\mathbf{E} = \langle \mathbf{A}, \mathbf{S}, \mathbf{S}_0, \mathbf{F}, \mathbf{sfn} \rangle$$

si costruisce un DFA che riconosce lo stesso linguaggio

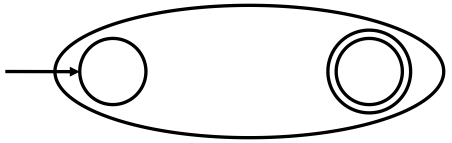
$$\mathbf{D} = \langle \mathbf{A}, \mathbf{S}_D, \mathbf{S}_{D0}, \mathbf{F}_D, \mathbf{sfn}_D \rangle$$

dove

- $\mathbf{S}_D = \{ X: X \subseteq \mathbf{S} \text{ e } X = \varepsilon\text{-CLOS}(X) \}$ *gli stati sono ε-closure*
- $\mathbf{S}_{D0} = \varepsilon\text{-CLOS}(\mathbf{S}_0)$ *lo stato iniziale è la ε-closure dello stato iniziale*
- $\mathbf{F}_D = \{ X: X \subseteq \mathbf{S}_D \text{ e } X \cap \mathbf{F} \neq \emptyset \}$ *gli stati finali sono quelli in cui compaiono gli stati finali dell'automa originario*
- $\mathbf{sfn}_D(\mathbf{X},\mathbf{a}) = \bigcup_{t \in \mathbf{X}} \{\varepsilon\text{-CLOS}(p) : p \in \text{sfn}(t,\mathbf{a})\}$ *dallo stato* **X** *con input* **a** *si arriva all'insieme degli stati a cui si arriverebbe con lo stesso input nell'automa originario*

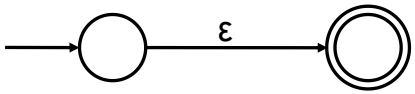
Da RE a ε-NFA

- Data una Regular Expression, si può costruire un automa ε-NFA componendo dei sotto-automi, in cui ciascun componente ha un nodo iniziale e uno finale

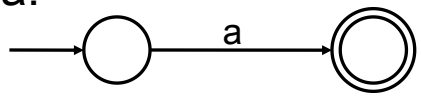


Da RE a ε-NFA

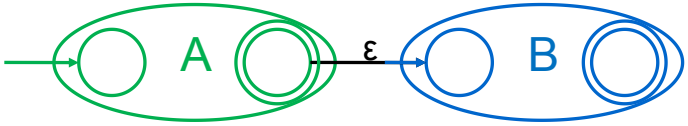
- Se la Regular Expression è ε:



- Se è a:

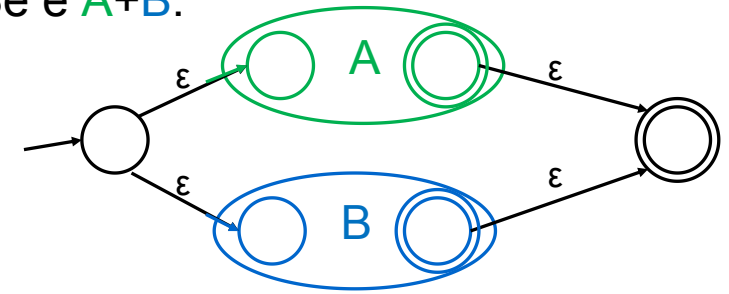


- Se è AB:

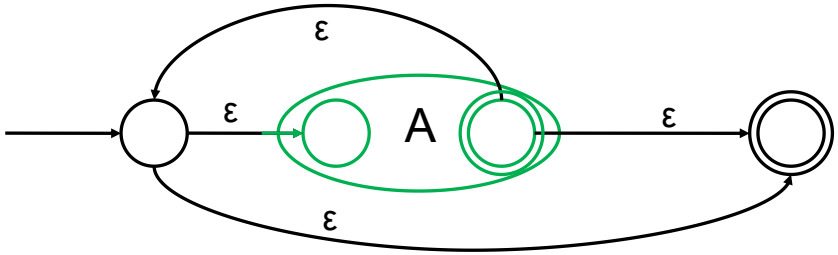


Da RE a ε-NFA

- Se è A+B:



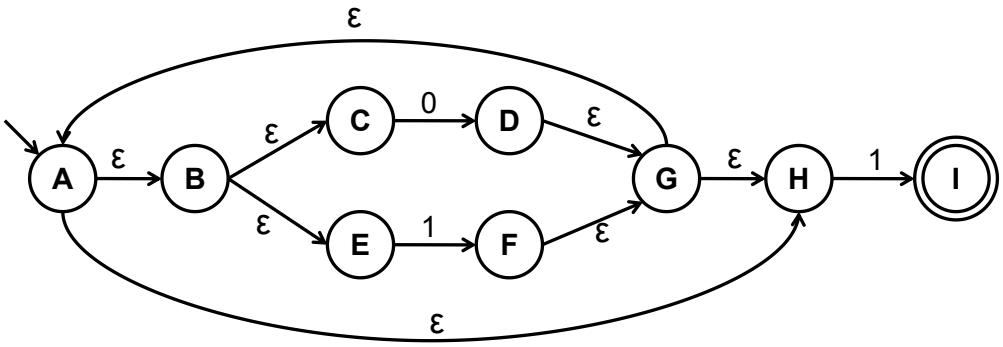
- Se è A*:



Esercizio

- Scrivere un ε-NFA che riconosce il linguaggio (0+1)*1

- Si scriva un ASF equivalente al seguente ε-NFA



Riassumendo

- Data un'espressione regolare, si può costruire un automa non deterministico basato su ε-mosse in maniera induttiva
- L'automa non deterministico può poi essere convertito in automa deterministico

UN ANTICO ESEMPIO

Fase 1 • **scrittura di un'equazione per ogni regola:**	Grammatica data: S → a B | a S B → d S | b
Fase 2 • **eventuali raccoglimenti a fattore comune per evidenziare suffissi:** ***qui non ce ne sono***	Equazioni: S = a B + a S B = d S + b
Fase 3 • **eliminare la ricorsione diretta** X = u X + δ **riscrivendola come** X = u* δ **(qui** δ = a B**)**	S = a* a B B = d S + b
Fase 4 • **sostituzione della 2ª equazione nella 1ª e sviluppo dei relativi calcoli**	S = a* a (d S + b) = = a* a d S + a* a b
Fase 5 • **nuova eliminazione della ricorsione introdotta al punto precedente: risultato finale.**	S = a* a d S + a* a b S = (a* a d)* a* a b

UN ANTICO ESEMPIO

Fase 1 • **scrittura di un'equazione per ogni regola:**	Grammatica data: S → a B | a S B → d S | b
Fase 2 • **se ora eliminiamo subito B, sostituendo la 2ª equazione nella 1ª e raccogliamo S:**	Equazioni: S = a B + a S B = d S + b
Fase 3 • **eliminando ora la ricorsione X = u X + δ riscrivendola come X = u* δ (qui δ = a b)**	S = a (d S + b) + a S = = (a d + a) S + a b
• **che costituisce già una espressione regolare (risultato finale)**	S = (a d + a)* a b

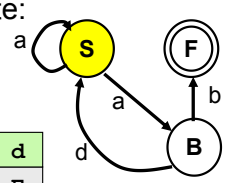
LA PRIMA ESPRESSIONE ottenuta: S = (a* a d)* a* a b
LA SECONDA ESPRESSIONE ottenuta: S = (a d + a)* a b
UNA TERZA ESPRESSIONE equivalente: S = a (d a + a)* b

ESPRESSIONI vs AUTOMI

Quali automi verrebbero originati dalla grammatica originale e dalle varie espressioni regolari ottenute?

La grammatica di partenza:
S → a B | a S
B → d S | b

L'automa non deterministico corrispondente:



	a	b	d
S	S/B	E	E
B	E	F	S
F	E	E	E
E	E	E	E

La prima espressione regolare:
S = (a* a d)* a* a b
La seconda espressione regolare:
S = (a d + a)* a b

Da Espressione Regolare a RSF
- ogni * → un ciclo (ε **-rule**) [autoanello nel caso più semplice]
- ogni + → percorsi alternativi

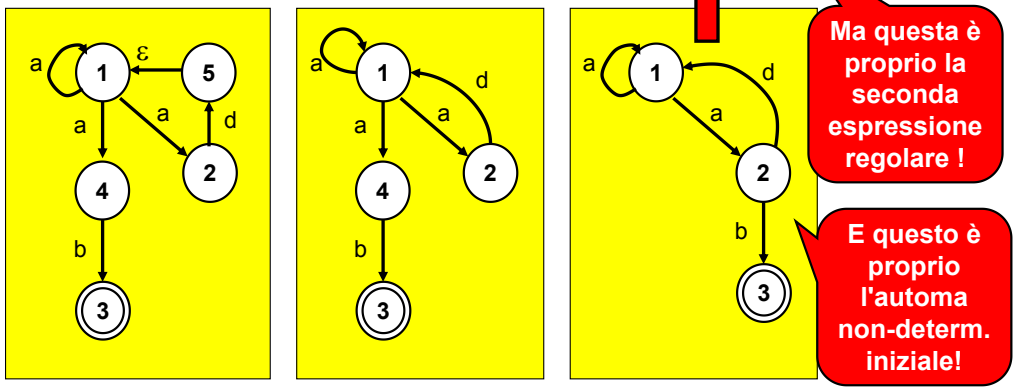
L'operatore * introduce ε **-archi**, facilmente eliminabili subito dopo con una semplice trasformazione.

ESPRESSIONI vs AUTOMI

Quali automi verrebbero originati dalla grammatica originale e dalle varie espressioni regolari ottenute?

La prima espressione regolare:
S = (a* a d)* a* a b

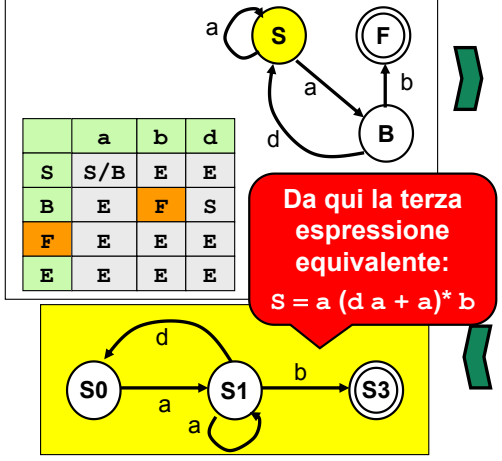
Una espressione equivalente:
S = (a + a d)* a b



ESPRESSIONI vs AUTOMI

Cosa succede rendendo *deterministico* e *minimo* l'automa (non-deterministico) ottenuto?

L'automa non deterministico:



	a	b	d
S	S/B	E	E
B	E	F	S
F	E	E	E
E	E	E	E

Automa deterministico (non minimo):

	a	b	d	
[S]	[S,B]	[E]	[E]	S0
[S,B]	[S,B,E]	[E,F]	[E,S]	S1
[S,B,E]	[S,B,E]	[E,F]	[E,S]	S2
[E,F]	[E]	[E]	[E]	S3
[E,S]	[S,B,E]	[E]	[E]	S4
[E]	[E]	[E]	[E]	E

Classi di equivalenza:
- **S1 / S2**
- **S0 / S4**

Automa minimo:

	a	b	d
S0	S1	E	E
S1	S1	S3	S0
S3	E	E	E
E	E	E	E