Definizioni su alfabeti e stringhe

Alfabeto: insieme finito di elementi detti **simboli** o caratteri. E*sempio* -> ∑ = {a, b, c}

- **Cardinalità:** la cardinalità di un alfabeto è il numero di simboli contenuti nell'alfabeto stesso. La cardinalità del dell'alfabeto ∑ si indica con |∑|. *Esempio* -> |{a, b, c}| = 3

Stringa: sequenza ordinata di simboli appartenenti all'alfabeto. *Esempio* -> "aab" è una stringa dell'alfabeto {a, b, c}

- **Stringa vuota:** la stringa vuota viene definita col simbolo **ε**.
- **Lunghezza**: la lunghezza di una stringa è il numero dei suoi caratteri, si denota con |x|. *Esempio* -> |abc| = 3.
- **Sottostringa**: y è sottostringa di x se esistono delle stringhe u e v tali che x=uvy.
- **Prefisso**: y è prefisso di x se esiste una stringa v tale che x=yv.
- **Suffisso**: y è un suffisso di x se esiste una stringa u tale che x=uy.
- **Stringa propria**: una sottostringa è detta propria se non coincide con la stringa vuota o con se stessa.

Operazioni sulle stringhe

Concatenamento (*non-commutativo*, *associativo*): il concatenamento di due stringhe è la stringa formata da tutti i simboli della prima stringa seguiti da tutti i simboli della seconda stringa. L'operazione di concatenazione si definisce tramite l'operatore "**.**"
Elemento neutro*:* ε
Esempio -> x = tele, y = visione
x.y = televisione
y.x = visionetele
x.ε = tele

Riflessione: la riflessione di una stringa è la stringa ottenuta scrivendo i suoi caratteri in ordine inverso.
Proprietà I: $(x^R)^R = x$
Proprietà II: la riflessione della concatenazione di due stringhe è la concatenazione inversa delle loro riflessioni. $(xy)^R = y^R x^R$
Elemento neutro*:* ε
Esempio -> x = tele
(x^R) = elet

Potenza: la potenza m-esima di una stringa x è il concatenamento di x con se stessa m volte.
Nota I: $x^0 = ε$
Esempio -> x = tele
x^2 = teletele

Precedenze tra operazioni sulle stringhe: Riflessione = Potenza > Concatenamento

Esempi finali:
$(ab)^R$ = ba
ab^2 = abb
$((ab)^3)^R)$ = $(ababab)^R$ = bababa

Definizioni sui linguaggi

Linguaggio: il linguaggio è un insieme di stringhe su un certo alfabeto. Le singole stringhe vengono anche dette **frasi**. *Esempio* -> {abba, baba, cabba} è un linguaggio sull'alfabeto {a, b, c}

- **Cardinalità**: la cardinalità di un linguaggio è il numero delle sue stringhe. La cardinalità del linguaggio L si indica con |L|. *Esempio* -> |{abba, bab}| = 2.
- **Linguaggio finito/infinito**: un linguaggio viene detto finito/infinita se ha cardinalità finita/infinita (c'è un numero finito/infinito di frasi).
- **Linguaggio vuoto:** denotato dal simbolo "Φ" è il linguaggio che non contiene stringhe. |Φ| = 0.

Operazioni sui linguaggi

Essendo i linguaggi degli insiemi, essi ereditano le operazioni classiche sugli insiemi.

Unione: $L_1 \cup L_2$ è l'insieme delle stringhe che appartengono a L_1 oppure a L_2.
Intersezione: $L_1 \cap L_2$ è l'insieme delle stringhe che appartengono sia a L_1 che a a L_2.
Differenza: $L_1 - L_2$ è l'insieme delle stringhe che appartengono sia a L_1 meno quelle che sono in L_2.
Inclusione: $L_1 \subseteq L_2$ (L_1 incluso in L_2) è vero se tutte le stringhe di L_1 appartengono anche a L_2.
Inclusione propria: $L_1 \subset L_2$ (L_1 incluso propriamente in L_2) è vero se tutte le stringhe di L_1 appartengono anche a L_2 ma in L_2 c'è almeno una stringa che non appartiene ad L_1.
Uguaglianza: $L_1 = L_2$ è vero se L_1 ed L_2 contengono esattamente le stesse stringhe.

Riflessione: la riflessione di un linguaggio L è l'insieme delle stringhe riflesse di L.
Esempio -> $(\{ab, ba, ca\})^R = \{ba, ab, ac\}$

Concatenazione: la concatenazione di due linguaggi L_1 ed L_2 è l'insieme ottenuto concatenando in tutti i modi possibili le stringhe di L_1 con quelle di L_2.
Nota I: $L\Phi = \Phi$
Nota II: $L\{\varepsilon\} = L$
Esempio -> {ab, abc}{ab, aa, cb} = {abab, abaa, abcb, abcab, abcaa, accb}

Potenza: la potenza m-esima di un linguaggio L è il concatenamento di L con se stesso m volte.
Nota I: $L^0 = \{\varepsilon\}$
Nota II: $\Phi^0 = \{\varepsilon\}$
Esempio -> $\{ab, abc\}^2 = \{abab, ababc, abcabc, abcab\}$ (tutti i modi possibili)

Chiusura di Kleene: è l'unione di tutte le potenze di L. L'operazione di chiusura di Kleene si definisce tramite l'operatore "*".
Proprietà I: $L \subseteq L^*$ (monotonicità)
Proprietà II: $(x \in L^*) \,\&\, (y \in L^*) \Rightarrow xy \in L^*$ (chiusura rispetto al concatenamento)
Proprietà III: $(L^*)^* = L$ (idempotenza)
Proprietà IV: $(L^*)^R = (L^R)^*$ (commutatività rispetto alla riflessione)
Esempio -> $(\{a,ab\})^* = \{\varepsilon, a, ab, aa, aab, aba, aaa, aaab\}$ $\{\varepsilon\}^* = \{\varepsilon\}$ $\Phi^* = \{\varepsilon\}$

Chiusura di Kleene non riflessiva: è l'unione di tutte le potenze positive di L. L'operazione di chiusura di Kleene si definisce tramite l'operatore "+". In sostanza, $L^* = L^+ \cup \{\varepsilon\}$

Complemento di un linguaggio: il complemento di un linguaggio L su un alfabeto ∑ rispetto ad un alfabeto Δ (notazione: $\neg L_\Delta$) è la differenza tra Δ* ed L. $\neg L_\Delta = \Delta^* - L$

I Linguaggi regolari

Espressione regolare: dicesi espressione regolare r su un alfabeto ∑ una stringa costruita sui seguenti elementi: ∑ ∪ {., |, *, Φ, (,)}
È in sostanza un modo più comodo di scrivere gli insiemi: si perde di complessità nella scrittura, pur mantenendo il linguaggio formale. Ecco alcuni esempi:

Espressione regolare	Insieme denotato
ε	{ε}
a (a ∈ ∑)	{a}
r | s	$L_r \cup L_s$
r.s (anche scritto rs)	$L_r L_s$
r*	L_r^*
(r)	L_r

- **Equivalenza**: due espressioni regolari sono detti equivalenti se denotano lo stesso insieme di stringhe.
- **Sottoespressioni**: r è una una sotto espressione di s se r occorre come parte di s. *Esempio* -> (ab)* è sottoespressione di (ab)* | bca

Il not e l'or
Non altrettanto semplici da definire sono le operazioni di not e quella di intersezione. Vediamo alcuni esempi. Dati:
r= a* b* = {a, aa, aaa, aaaa, ... , b, bb, bbb, bbb, ...}
s= $(a|b)^2$ = {aa, ab, ba, bb.. e combinazioni di queste}
Abbiamo che l'intersezione fra i due deve permettere sia le prime stringhe che le seconde. Quindi si ha che (aa)*(ab|ε)(bb)*
Invece, ¬s (dato che s permette la creazione di sole stringhe di valore pari) ci basta mettere una a od una b per rendere la stringa dispari ed automaticamente non appartenente al linguaggio, pertanto $(a|b)(a|b)^2$ è il risultato.

Liste con marche: è possibile desiderare inserire degli altri simboli in una lista, ad esempio desideriamo poter scrivere dei numeri binari separati da un punto e virgola. Allora definiamo st = 1(0|1)* | 0 [cioè un qualsiasi numero binario che inizi con uno] e poi possiamo dire che eseguiremo un st(; st)*.

Grammatiche context-free

Definizione

Una grammatica context-free è una quadrupla G = <V, ∑, P, S> dove:

- V è l'insieme dei simboli non terminali, *esempio:* { A, B, ... }
- ∑ è l'insieme dei simboli terminali, *esempio:* { a, b, ... }
- P è l'insieme delle regole di produzione, *esempio:* { A -> α, B ->β, ... } *dove α e β sono* ∈ *(V ∪ ∑)**
- S ∈ V è l'assioma di partenza (un non-terminale da cui partire)

Non sono escluse produzioni A -> ε.

Esempio:

< {S, T}, {a,b}, {S -> aT | bT, T -> a | b}, S > si può derivare come S ⇒ aT ⇒ aa.

Un **linguaggio** è **context-free** se esiste una grammatica context-free che lo genera.

Due grammatiche G e G' sono **debolmente equivalenti** se generano lo stesso linguaggio, cioè L(G) = L(G').

Trasformazioni di grammatiche

Vi sono due tipi di trasformazioni: ***pulire le grammatiche*** *e* ***renderle in forma normale***.

I simboli inutili

Un simbolo è inutile quando non concorre a formare il linguaggio generato.

- Vi sono non terminali **non definiti** (generano il linguaggio vuoto, ovvero che prima o poi "terminano" in una stringa e non creano un ciclo infinito, per così dire),
- Vi sono poi i simboli **non raggiungibili** (banalmente: partendo dall'assioma non riusciamo a raggiungerli/produrli)

L'algoritmo

Prima si calcolano i non terminali non definiti.

Si genera l'insieme DEF, il quale è composto dalle produzioni che "terminano", poi, mano a mano, si aggiungono le produzioni successive che sono in grado di "terminare".

S -> B		S -> AB
B -> aC | a	Chiaramente, partendo da D si va in E o F, e da E in D, così come da F.	B -> aC | a
C -> b | Ed	C'è un ciclo che "blocca" il tutto.	C -> b
D -> Ea | F	Dopo aver eliminato i non definiti si ha:	
E -> Dc	---------->	
F -> D | E		

Per quanto concerne i simboli non raggiungibili, basta partire dall'assioma, segnare tutti i simboli raggiungibili, quindi eliminare quelli non contenuti nell'insieme appena creato.

S -> AB	H è evidentemente un non terminale irraggiungibile, non c'è alcuna	S -> AB
B -> aC | a	produzione che abbia H nella sua parte destra.	B -> aC | a
C -> b | c	Dopo aver eliminato i non raggiungibili si ha:	C -> b | c
H -> ABC	---------->	

Le regole di copiatura

Una regola di copiatura, detta anche produzione unitaria, è una produzione del tipo A -> B (anche in più passi). Tali regole sono inutili ed è sufficiente copiare le parti destre delle produzioni copiate dentro alla produzione che usa la copia, quindi se { A -> B | c, B -> a | b } si ha che { A -> a | b | c }.

L'algoritmo

Si definisce l'insieme COPIA(A) = { ... } che definisce la serie di non terminali che si possono copiare in A.

Al primo passo, COPIA(A) = { A }, poi, iterativamente, COPIA(A) += COPIA(A) ∪ { C | B -> C & B ∈ COPIA(A) }, ovvero si aggiunge a COPIA(A) gli insiemi COPIA() (al passo precedente!) di tutti gli elementi contenuti in COPIA(A) stesso. Si procede finché gli elementi copia, continuando ad iterare, rimangono immutati.

E -> E + T | T	Si hanno (~ rappresenta il passo iterativo) :	E -> E + T | T * C | a | b | c
T -> T * C | C	COPIA(E) = { E } ~ { E, T } ~ { E, T, C }	T -> T * C | a | b | c
C -> a | b | c	COPIA(T) = { T } ~ {T, C}	C -> a | b | c
	COPIA(C) = { C } Pertanto si ha:	
	---------->	

Forma normale non annullabile

Un non-terminale si dice annullabile quando possiede una produzione (A -> ε) ∈ P.

Una grammatica si dice **in forma normale non annullabile** se:

- Nessun non terminale diverso dall'assioma è annullabile
- L'assioma è annullabile solo se la stringa vuota appartiene al linguaggio

L'algoritmo

Come primo passo si sostituiscono nelle parti destre delle produzioni, tutte le regole, considerando i non terminali annullabili come annullati (cioè ciò che potremmo produrre se annullassimo un non terminale annullabile). Si eliminano poi le regole del tipo A -> ε. Nel caso di assioma annullabile e che appare nella parte destra di altre produzioni, si pone S_0 -> S | ε.

A -> B | C | aBb | aB | Bb	B possiede una ε produzione. In A si aggiungono	A -> B | C | aBb | aB | Bb | ab | a | b
B -> aC | b | ε	tutte le produzioni che si avrebbero se B andasse	B -> aC | b |
C -> b	in ε. Si ha quindi:	C -> b
	---------->	

Grammatica pulita

Una grammatica è pulita se non ha terminali inutili e non ammette produzioni circolari ovvero che A $\Rightarrow^+$ A in qualsiasi numero di passi (si parla di una derivazione esattamente A -> A, non di una ricorsione generica).

Forma normale di Chomsky

Una grammatica è in forma normale di Chomsky se:

- Vi sono solo regole nella forma A -> BC oppure A -> a
- S -> ε esiste solo se il linguaggio contiene la stringa vuota

Forma normale di Greibach

Una grammatica è in forma normale di Greibach se:

- Il linguaggio che genera la grammatica non contiene la stringa vuota
- Tutte le regole sono di forma A -> aβ dove a ∈ ∑ e β ∈ V*(ovvero si ha un terminale seguito da non terminali)

Linguaggi infiniti

Ricorsione

Una grammatica è detta ricorsiva se un terminale produce se stesso in qualche passo ovvero $A \Rightarrow^+ \alpha A \beta$.

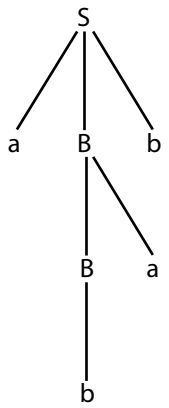
È detta **ricorsiva sinistra** se $\alpha = \varepsilon$ cioè $A \Rightarrow^+ A\beta$, **ricorsiva destra** se $\beta = \varepsilon$ cioè $A \Rightarrow^+ \alpha A$.

A è quindi detto terminale non ricorsivo.

Il linguaggio L(G) prodotto dalla grammatica G pulita è **infinito** solo se G ammette derivazioni ricorsive.

Alberi sintattici

Possiamo rappresentare le derivazioni tramite alberi.



Ad esempio data la grammatica G = <{ S }, { a, b }, {S -> aBb, B -> aB | Ba | b }, S> ed una derivazione S ⇒ aBb ⇒ aBab ⇒ abab si ha l'albero dell'immagine a sinistra.

Derivazioni canoniche

- Una derivazione canonica è **sinistra** se ad ogni passo viene sostituito il non terminale più a sinistra.
- Una derivazione canonica è **destra** se ad ogni passo viene sostituito il non terminale più a destra.

Alberi ambigui

Una frase è ambigua se ha due alberi sintattici distanti. Una grammatica è ambigua se almeno una frase del linguaggio generato è ambigua.

Grammatiche equivalenti

Una grammatica è **debolmente equivalente** se generano lo stesso linguaggio.
Una grammatica è **fortemente equivalente** se è debolmente equivalente e assegna ad ogni frase alberi sintattici isomorfi.

Grammatiche lineari e unilineari

Grammatiche lineari

Una grammatica è **lineare** se le sue regole hanno la forma: A -> uBv con u, v ∈ ∑* e B ∈ V.
Una grammatica è **unilineare destra** se le sue regole hanno la forma: A -> uB con u ∈ ∑* e B ∈ V.
Una grammatica è **unilineare sinistra** se le sue regole hanno la forma: A -> Bv con v ∈ ∑* e B ∈ V.

Proprietà tra le classi dei linguaggi

- S-UNI-LIN ⊆ UNI-LIN (strettamente unilineari incluse nelle unilineari)
- UNI-LIN ⊆ S-UNI-LIN (unlineari incluse nelle strettamente unilineari)
- perciò: UNI-LIN ≡ S-UNI-LIN
- inoltre: UNI-LIN ≡ S-UNI-LIN ⊆ REG

Dalla grammatica unilineare alla espressione regolare

È sempre possibile, per la regola sopracitata, trasformare una grammatica unilineare in una espressione regolare.
Prendiamo la grammatica e trasformiamo tutte le produzioni sostituendo la freccia con l'uguale. Questo ha un grande significato semantico, che è derivante dalla equivalenza sopracitata.
Se non c'è ricorsione, l'equazione rimane immutata, altrimenti si eliminano quindi le ricorsioni, utilizzando le due regole:

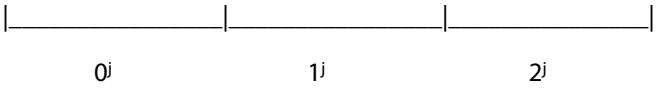
- $S = \alpha S \mid \beta \rightsquigarrow S = \alpha^*\beta$
- $S = S\alpha \mid \beta \rightsquigarrow S = \beta\alpha^*$

Esempio

Partenza	Uguaglianze	Raccolgo i coefficienti	Applico la regola per eliminare la ricorsione	Si è quindi applicato:
S -> aS | bS | A	S = aS | bS | A	S = (a|b)S | A	S = (a|b)*A	$S = \alpha S \mid \beta \rightsquigarrow S = \alpha^*\beta$,
A -> aB	A = aB	A = aB	A = aB	
B -> aC	B = aC	B = aC	B = aC	In questo caso:
C -> cC | c	C = cC | c	C = cC | c	C = cC | c	$\alpha = (a\|b)$
				$\beta = A$

Pumping lemma

Per ogni linguaggio context-free L esiste una costante caratteristica k (che dipende dall'L in questione) tale che ogni frase $z \in L$ di lunghezza maggiore o uguale a k (cioè $|z| \geq k$) si può scrivere come concatenazione di cinque sottostringhe **x u w v y**.
Tali sottostringhe hanno le proprietà: $|uwv| \leq k$ e $uv \neq \varepsilon$, tali che $xu^iwv^iy \in L$ per ogni $i \geq 0$.
Esempio
$\{ 0^n 1^n 2^n \mid n > 0 \}$ non è context free.
Infatti, prendiamo una stringa generica con $n = j > k$ (il numero prestabilito).



Allora, dato che $|uwv| \leq k$, e che abbiamo preso un $j > k$, abbiamo che la stringa $|uwv|$ sarà più corta di 0^j, 1^j, 2^j. Quindi, pompando con la i la u e la v, una delle tre sezioni sarebbe subito sbilanciata. Se prendessimo poi uwv in modo che un po' di u siano nella parte degli zeri, e un po' di v nella parte degli uni (ovvero "accavallata"), pompando u e v (ovvero 0 e 1) rimaremmo sbilanciati rispetto ai 2. Quindi non esiste una sottostringa $|uwv| \leq k$ e $uv \neq \varepsilon$, tali che $xu^iwv^iy \in L$ per ogni $i \geq 0$. Il pumping lemma non vale, quindi il linguaggio non è context-free.

Esiste poi la versione "più forte" per le grammatiche **unilineari**, ovvero:
Per ogni linguaggio context-free L esiste una costante caratteristica k (che dipende dall'L in questione) tale che ogni frase $z \in L$ di lunghezza maggiore o uguale a k (cioè $|z| \geq k$) si può scrivere come concatenazione di tre sottostringhe **x u w** .
Tali sottostringhe hanno le proprietà: $|uw| \leq k$ e $u \neq \varepsilon$, tali che $xu^iw \in L$ per ogni $i \geq 0$.

Esiste ovviamente la variante unilineare destra che utilizza le stringe **w v y**.

Esempio

$\{ a^n b^n \mid n > 0 \}$ non è generato da nessuna grammatica unilineare.

Infatti se prendiamo una stringa generica con $j = n > k$

|_____|_____|

a^j b^j

Dato che $|uw| \leq k$, e $j > k$, allora uw deve necessariamente "sforare" dagli zeri verso gli uni e viceversa. Quindi, pompando uno dei due, si rimarrebbe sbilanciati. Dimostrato che non è generato da una grammatica unilineare sx.

Classificazione di Chomsky

Chomsky ha introdotto una classificazione famosissima dei linguaggi, distinguendoli in tipo 3 (regolari), tipo 2 (context free), tipo 1 (contestuali), tipo 0 (ricorsivamente enumerabili).

S-UNILIN ≡ UNILIN ≡ REG (tipo 3) ⊆ Lineari ⊆ Context-free (tipo 2) ⊆ Contestuali (tipo 1) ⊆ Ricorsivamente enumerabili (tipo 0).

Automi

Per cercare di capire se una stringa rispetta o meno una grammatica è necessario implementare un algoritmo di riconoscimento. Tale algoritmo, detto **automa** è una macchina decisionale che per ogni stringa x ci dice se essa è accetta o meno dalla grammatica.

Automi a stati finiti

Automa finiti deterministici

Il tipo più semplice di automi è quello detto "a stati finiti" che possono essere deterministici e non deterministici.
Un **automa finito deterministico** M è una quintupla $<Q, \Sigma, \delta, q_0, F>$ tale che:
- Q è l'insieme degli stati (finito e non vuoto)
- Σ è l'alfabeto di ingresso
- $\delta : Q \times \Sigma \rightarrow Q$ è una funzione di transizione parziale, che porta da uno stato ad un altro
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali.

Le caratteristiche principali di questo automa sono il fatto che non ha memoria e che la testina sul nastro di input può solo leggere e avanzare di una posizione per volta verso destra.

Una stringa x è **riconosciuta** dall'automa M se $\delta^* (q_0,x) \in F$ cioè se esistono un certo numero di delta che ci portano ad uno stato finale leggendo x. Un linguaggio L(M) **riconosciuto** è l'insieme delle stringhe riconosciute da M:
$L(M) = \{ x \in \mid \delta^* (q_0,x) \in F \}$.

Quando in uno stato non è definita nessuna mossa per un carattere in input, è sottinteso che l'automa leggendo tale carattere transisca in uno stato di errore detto **pozzo**.

Automa finiti deterministici minimi

Dato un automa finito deterministico M che riconosce un linguaggio L(M) è possibile costruire un automa detto "minimo" ovvero col il minimo numero di stati, che, ovviamente, continua a riconoscere il linguaggio L(M).

Uno stato q è detto **accessibile** se è raggiungibile dallo stato iniziale e **post-accessibile** se da q si può raggiungere uno stato finale. Uno stato non è accessibile oppure non è post-accessibile è detto **inutile**. Vi sono poi gli stati detti **ridondanti** ovvero che potrebbero essere fusi con altri. Al termine della costruzione dell'automa minimo vi saranno solo più stati indispensabili, anche detti **indistinguibili**.

Definizione
Sia $M = <Q, \Sigma, \delta, q_0, F>$ un automa finito deterministico senza stati inutili, con $p, q \in Q$ e $p \neq q$.
1) Una certa stringa x *distingue* tra p e q se uno e uno solo tra gli stati $\delta(p,x)$ e $\delta(q,x)$ è finale.
2) p e q sono *k-indistinguibili* $p \equiv^k q$ se nessuna stringa x la cui lunghezza è minore di k distingue p da q.
3) p e q sono *indistinguibili* $p \equiv q$ se sono indistinguibili per tutti i $k \geq 0$.

L'idea è quella di creare delle classi di equivalenza (inizialmente due, stati finali e stati non finali) e poi, iterativamente, "accorpare" le classi cercando di vedere esaminando tutte le δ se la transizione ci porta ad uno stato (k-1)esimo equivalente.

L'algoritmo

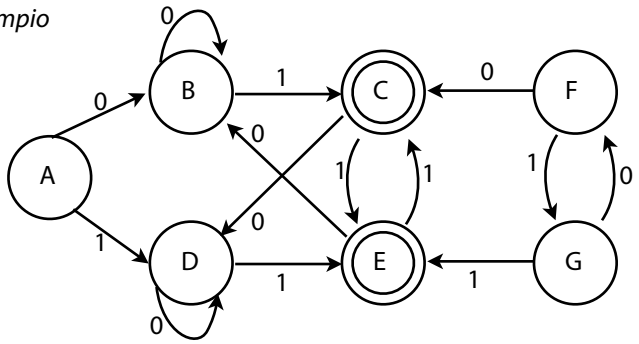
Sfruttando le tre regole sopracitate possiamo ottenere un semplice algoritmo di minimizzazione di un automa.

1) Cancella tutti gli stati inutili
2) Costruisci le classi d'equivalenza a livello 0, cioè dividendo gli stati finali dai non finali.
3) Costruisci il livello 1, prendi due stati e prova tutte le funzioni delta possibili (cioè con tutti i simboli $\in \Sigma$) e verifica che per ogni delta i due stati finiscano in altri due stati 0 equivalenti.
4) ...
5) Costruisci il livello k, prendi due stati e prova tutte le funzioni delta possibili (cioè con tutti i simboli $\in \Sigma$) e verifica che per ogni delta i due stati finiscano in altri due stati k equivalenti.

Quando le classi ottenute dal livello (k+1)-esimo sono uguali a quelle del livello k-esimo, allora abbiamo tutti stati indistinguibili.

6) Unire gli stati che sono nelle stesse classi di equivalenza.

Esempio



- F e G non sono raggiungibili dallo stato iniziale, quindi vengono eliminati automaticamente.
- Livello k = 0. {A, B, D} {C, E}. (divido finali da non finali)
- Livello k = 1.

A $\equiv^1$ B ?
$\delta(A, 0) \equiv^0 \delta(B, 0)$? --> B $\equiv^0$ B --> Sì.
$\delta(A, 1) \equiv^0 \delta(B, 1)$? --> D $\equiv^0$ C --> No.
Sì & No = No. A e B non sono $\equiv^1$ equivalenti.
Classi al momento: {A {B

A $\equiv^1$ D ?
$\delta(A, 0) \equiv^0 \delta(D, 0)$? --> B $\equiv^0$ D --> Sì.
$\delta(A, 1) \equiv^0 \delta(D, 1)$? --> D $\equiv^0$ E --> No.
Sì & No = No. A e D non sono $\equiv^1$ equivalenti.
Classi al momento: {A {B {D

B $\equiv^1$ D ?
$\delta(B, 0) \equiv^0 \delta(D, 0)$? --> B $\equiv^0$ D --> Sì.
$\delta(B, 1) \equiv^0 \delta(D, 1)$? --> C $\equiv^0$ E --> Sì.
Sì & Sì = Sì. B e D sono $\equiv^1$ equivalenti.
Classi al momento: {A {B,D}

C $\equiv^1$ E ?
$\delta(C, 0) \equiv^0 \delta(E, 0)$? --> D $\equiv^0$ B --> Sì.
$\delta(C, 1) \equiv^0 \delta(E, 1)$? --> E $\equiv^0$ C --> Sì.
Sì & Sì = Sì. C e E sono $\equiv^1$ equivalenti.
Classi finali: {A} {B,D} {C,E}

Classi a fine passo: {A} {B,D} {C,E}

- Livello k = 2.

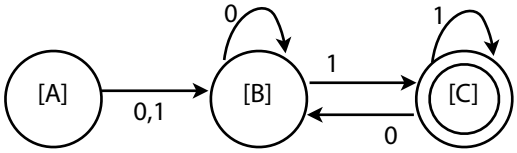
B $\equiv^2$ D ?
$\delta(B, 0) \equiv^1 \delta(D, 0)$? --> B $\equiv^1$ D --> Sì.
$\delta(B, 1) \equiv^1 \delta(D, 1)$? --> C $\equiv^1$ E --> Sì.
Sì & Sì = Sì. B e D sono $\equiv^2$ equivalenti.
Classi al momento: {A} {B,D}

C $\equiv^2$ E ?
$\delta(C, 0) \equiv^1 \delta(E, 0)$? --> D $\equiv^1$ B --> Sì.
$\delta(C, 1) \equiv^1 \delta(E, 1)$? --> E $\equiv^1$ C --> Sì.
Sì & Sì = Sì. C e E sono $\equiv^2$ equivalenti.
Classi finali: {A} {B,D} {C,E}

Il livello 2 è uguale al livello 1, quindi k+1 è uguale a k. Abbiamo finito.

Le classi ottenute sono: {A} {B,D} {C,E}

Rinominiamole rispettivamente come [A] [B] [C] e riscriviamo l'automa.



Si ricorda che una classe di equivalenza che contiene uno stato finale diventa uno stato finale nell'automa minimo.

Automa finiti non deterministici

Un automa finito non deterministico N è una quintupla <Q, ∑, δ, q_0, F> dove però δ : Q x ∑ -> Q è una funzione di transizione parziale, che definisce un insieme di possibili stati successori, e non uno solo. In sostanza da uno stato possiamo avere più "frecce" con lo stesso label. Ovvero data una lettura in input 'a' potremmo avere più stati successori. Questo crea un evidente non determinismo.

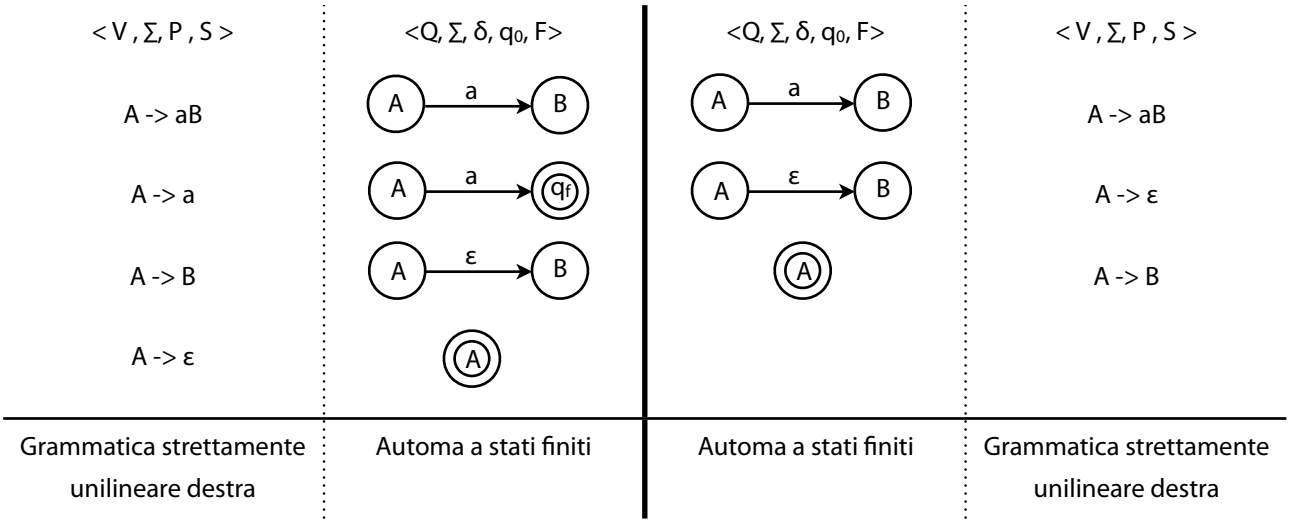
Un automa finito con **mosse spontanee** è una quintupla <Q, ∑, δ, q_0, F> dove però δ non è necessariamente una funzione con due parametri, ovvero potrebbe bastare uno stato per passare ad un altro. In sostanza esistono delle mosse "speciali", spontanee, dette **epsilon mosse** le quali senza "preavviso" potrebbero far mutare uno stato in un altro.

Automi finiti, grammatiche unilineari ed espressioni regolari

Si può mostrare che:
- I linguaggi generati dalle grammatiche unilineari destre sono riconosciuti dagli automi finiti e viceversa
- I tre modelli di automi finiti sono equivalenti (è possibile passare da uno all'altro)
- I linguaggi regolari sono riconosciuti dagli automi finiti
- Le grammatiche unilineari destre e le grammatiche unilineari sinistre generano la stessa classe di linguaggi.

Relazioni tra grammatiche e automi

Vediamo come trasformare una grammatica unilineare destra in un automa a stati finiti.
Si ricorda che Q = V ∪ {q_f} che è uno stato finale, mentre lo stato iniziale q_0 è l'assioma S.



Eliminazione della non determinazione di un automa

L'operazione consta di due parti, la prima è l'eliminazione delle epsilon mosse la seconda invece è l'eliminazione delle indeterminazioni.

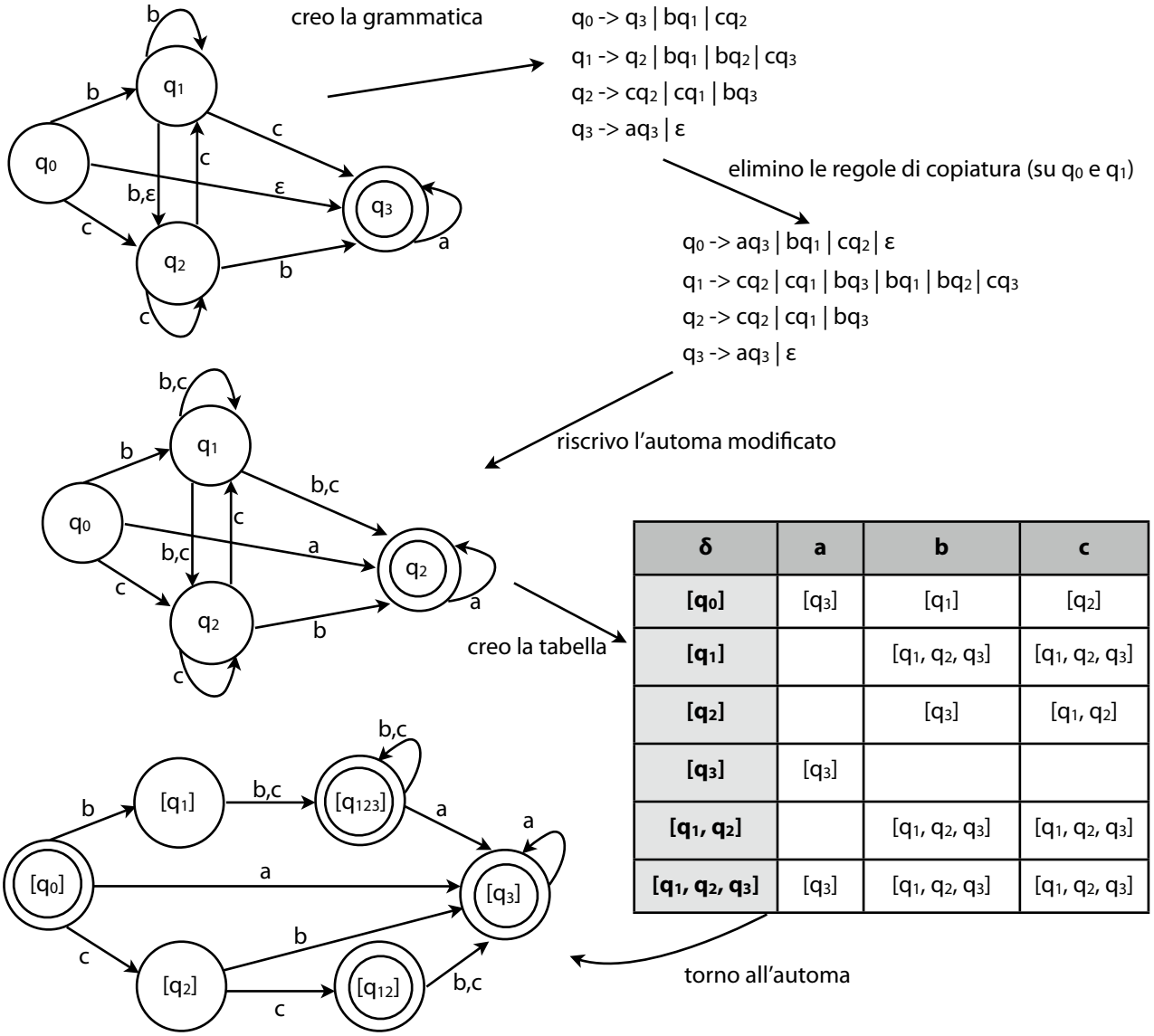
Eliminare le mosse spontanee (epsilon mosse)

Un semplice metodo per eliminare le mosse spontanee è quello di trasformare l'automa in grammatica context-free, eliminare le regole di copiatura, e ritrasformare la grammatica in automa.

Eliminare il non determinismo

Non è altrettanto semplice eliminare il non determinismo! Creiamo una tabella, inserendo sulla colonna più a destra gli stati (mano a mano!). Il primo stato sarà ovviamente quello iniziale. Sulla prima riga invece inseriamo tutti i simboli $\in \Sigma$. All'interno della tabella inseriamo quindi tutti gli stati, che, partendo dallo stato sulla colonna sinistra, percorrendo tutte le "strade" fornite dai simboli di Σ, riusciamo a raggiungere. Poi aggiungiamo quegli stati nella prima colonna a sinistra e ripetiamo finché la tabella non smette di produrre nuovi stati. A questo punto rinominiamo li stati e otteniamo un nuovo automa, finalmente, deterministico.

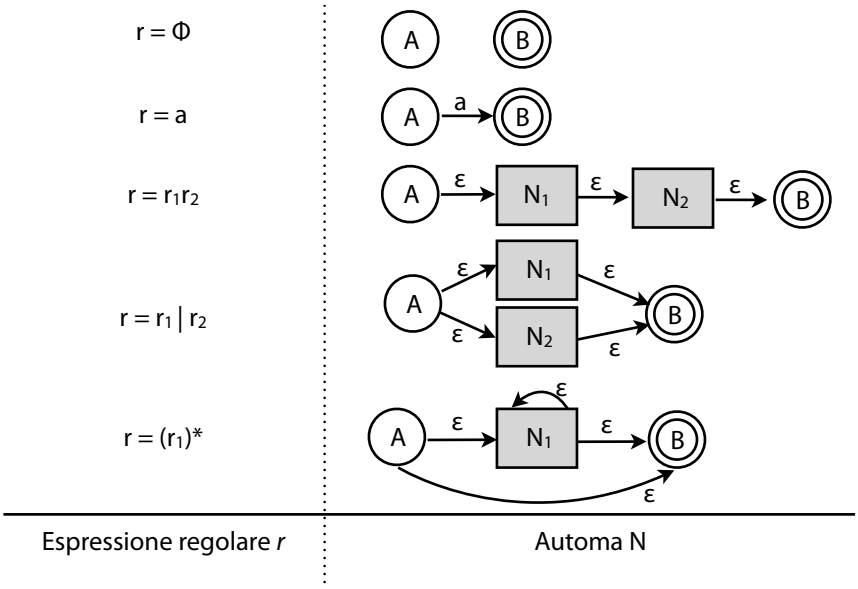
Esempio



δ	a	b	c
[q0]	[q3]	[q1]	[q2]
[q1]		[q1, q2, q3]	[q1, q2, q3]
[q2]		[q3]	[q1, q2]
[q3]	[q3]		
[q1, q2]		[q1, q2, q3]	[q1, q2, q3]
[q1, q2, q3]	[q3]	[q1, q2, q3]	[q1, q2, q3]

Relazioni tra le famiglie dei linguaggi (relazioni tra espressioni regolari e automi)

Abbiamo quindi capito che possiamo passare da automi finiti con mosse spontanee ad automi finiti ad automi finiti deterministici, e da ognuno di essi possiamo trovare una grammatica unilaterale destra. E per quanto concerne le espressioni regolari?



Data un'espressione regolare r, esiste un automa N con un solo stato finale da cui non sono definite transizioni, tale che L(N) = L(*r*).

Si precisa che N_1 ed N_2 sono automi che sono in grado di riconoscere i singoli r_1 / r_2

Data un'espressione regolare r, esiste un automa N con un solo stato finale da cui non sono definite transizioni, tale che L(N) = L(*r*).

Abbiamo quindi capito che possiamo passare dalla grammatica unilineare destra all'espressione regolare, e dall'espressione regolare all'automa. Possiamo inoltre passare dall'automa alla grammatica unilineare destra.

Chiusure delle classi dei linguaggi regolari

I linguaggi regolari sono chiusi per riflessione, complementazione e intersezione. Quindi, tali operazioni sono effettuabili anche sugli automi.

Riflessione: è sufficiente scambiare tutte le frecce, ricordando però di inserire un nuovo stato iniziale con epsilon mosse verso i vecchi finali, se gli stati finali sono maggiori di 1.

Complementare: è l'automa che è in grado di leggere il linguaggio complementare.

Si crea il pozzo (con un cappio su se stesso di ∑), quindi si invertono tutti gli stati: quelli iniziali diventano non - iniziali, e quelli non - iniziali diventano iniziali (pozzo compreso!). Così possiamo riconoscere tutte le stringhe esistenti, ma certamente non quelle che riconosceva il linguaggio precedente.

Intersezione: Si intersecano tutti i vari insiemi della quintupla, $Q = Q_1 \times Q_2$, $\Sigma_1 \cap \Sigma_2$, $q_0 = \langle q_1,q_2 \rangle$, $F = F_1 \times F_2$

e per quanto concerne le δ, si ha $\delta(\langle q,p \rangle, a) = \langle \delta_1(q,a), \delta_2(p,a) \rangle$ se sia δ_1 che δ_2 sono definite, altrimenti non si crea del tutto la δ (ovvero non c'è l'arco).

Trasformare una grammatica unilineare destra in unilineare sinistra

Date le proprietà sopraelencate, è chiaro come passare da una grammatica **unilineare destra a una sinistra**:

1 - Costruire l'automa riconoscitore di L.

2 - Costruire l'automa riconoscitore di L^R.

3 - Trovare a partire dall'automa la grammatica G^R unilineare destra che genera L^R.

4 - Scrivere la grammatica G ottenuta da G^R rovesciando i membri destri delle produzioni.

Ripercorrendo i punti al contrario passiamo da una **unilineare sinistra ad una destra**.

Automi push-down: gli automi a pila

Questo tipo di automi hanno memoria, e si sviluppano con una sorta di stack. Si ha un nastro di input, con la stringa, e poi una memoria. Sono però necessari intrinsecamente degli stati, per poter definire "dato questo stato, se trovo sulla pila quello, allora.."

Gli automi a pila

Un **automa a pila** è una settupla $M = < Q, \Sigma, \Gamma, \delta, q_0, Z_0, F >$ dove:
- Q è l'insieme degli stati dell'unità di controllo
- Σ è l'alfabeto di ingresso
- Γ è l'alfabeto della pila
- δ è la mossa, $\delta(q, a, Z)$ con q come stato, a sulla testina e Z nello stack. (maggiori dettagli in seguito)
- $q_0 \in Q$ è lo stato iniziale
- $Z_0 \in \Gamma$ è il simbolo iniziale della pila
- $F \subseteq Q$ è l'insieme degli stati finali

Le transizioni: un po' più complesse
Le transizioni δ risultano un po' più complesse rispetto all'automa deterministico senza memoria. Ve ne sono di due tipi:
- **mossa con lettura**: $\delta(q, a, Z) = \{(p_1, \gamma_1), ..., p_k, \gamma_k)\}$. Si è nello stato $q \in Q$, si ha 'a' $\in \Sigma$ sulla testina in input e $Z \in \Gamma$ sulla pila. Date queste condizioni, ,viene non deterministicamente cambiato lo stato da q a p_i, vengono effettuate una pop (di Z, in questo caso) ed una push di γ_i. Avanza inoltre la testina sull' input.
- **mossa spontanea:** $\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), ..., p_k, \gamma_k)\}$. Si è nello stato $q \in Q$ e si ha $Z \in \Gamma$ sulla pila. Date queste condizioni, viene non deterministicamente cambiato lo stato da q a p_i, effettuate una pop ed una push di γ_i. La testina di input NON avanza.

Entrambe le mosse sono non deterministiche poiché non si sa quale delle scelte venga presa in considerazione.
Non sono ammesse transizioni a pila vuota.

La configurazione iniziale
Inizialmente la nostra pila appare così: (q_0, x, Z_0).

Riconoscere un linguaggio
- Un linguaggio è **riconosciuto per stato finale** se la lettura di una stringa x porta l'automa M in una configurazione in cui lo stato è definito "finale" cioè appartenente a F.
- Un linguaggio è **riconosciuto per pila vuota** se la lettura di una stringa x porta l'automa M in una configurazione in cui la memoria è vuota.

La transizione
La transizione tra una configurazione ed un altra si può indicare, in simboli (sapendo che q è lo stato di controllo, y è la parte della stringa ancora da esaminare e η è il contenuto della pila):
$(q, ay, \eta Z) \vdash (p, y, \eta\gamma)$ se $(p, \gamma) \in \delta (q, a, Z)$ (leggo a, pop di Z, push di γ e cambio di transizione in p)
$(q, ay, \eta Z) \vdash (p, ay, \eta\gamma)$ se $(p, \gamma) \in \delta (q, \varepsilon, Z)$ (non leggo nulla, pop di Z, push di γ e cambio di transizione in p)

Esempio

Automa a stack per: $N(M) = \{xx^R \mid x \in \{r,v\}^*\}$

$Q = \{q_0,q_1\}$ $\Sigma = \{v, r\}$ $\Gamma = \{Z_0, R, V\}$ $F = \Phi$

Stato	Input	Stack	Transizioni
q_0	r	Z_0	$\{ (q_0, Z_0R) \}$
q_0	v	Z_0	$\{ (q_0, Z_0V) \}$
q_0	r	R	$\{ (q_0, RR), (q_1, \varepsilon) \}$
q_0	v	R	$\{ (q_0, RV) \}$
q_0	ε	Z_0	$\{ (q_0, \varepsilon) \}$

Stato	Input	Stack	Transizioni
q_0	r	V	$\{ (q_0, VR) \}$
q_0	v	V	$\{ (q_0, VV), (q_1, \varepsilon) \}$
q_1	r	R	$\{ (q_1, \varepsilon) \}$
q_1	v	V	$\{ (q_1, RV) \}$
q_1	ε	Z_0	$\{ (q_1, \varepsilon) \}$

Avviene un cambiamento di stato se iniziamo a "tornare indietro", ovvero se abbiamo sia sulla testina d'input che sullo stack lo stesso simbolo. Abbiamo comunque una non determinazione, poiché potremmo semplicemente voler accumulare delle r o delle v.

$(q_0, rvrrvr, Z_0) \vdash (q_0, vrrvr, Z_0R) \vdash (q_0, rrvr, Z_0RV) \vdash (q_0, rvr, Z_0RVR) \vdash (q_1, vr, Z_0RV) \vdash (q_1, r, Z_0R) \vdash (q_1, \varepsilon, Z_0) \vdash (q_1, \varepsilon, \varepsilon)$

Al quinto caso avremmo avuto una non determinazione, ma noi, esseri senzienti, sappiamo che siamo a metà stringa, quindi conviene andare in q_1.

Gli automi a pila deterministici

Gli automi a pila deterministici non hanno "problemi di determinazione" ovvero da uno stato possono transitare solo in un altro stato. In sostanza, $|\delta(q, a, Z)| \leq 1$ e $|\delta(q, \varepsilon, Z)| \leq 1$.
Se per una coppia q, Z è definita una mossa spontanea, non è definita nessuna mossa con lettura.

Proviamo ora a risolvere il problema di prima... ci servirebbe una sorta di "carattere di centro stringa" così da sapere che siamo giunti a metà. Quindi...

Automa a stack per: $N(M) = \{xbx^R \mid x \in \{r,v\}^*\}$

$Q = \{q_0,q_1\}$ $\Sigma = \{r, b, v\}$ $\Gamma = \{Z_0, R, V\}$ $F = \Phi$

Stato	Input	Stack	Transizioni
q_0	r	Z_0	$\{ (q_0, Z_0R) \}$
q_0	v	Z_0	$\{ (q_0, Z_0V) \}$
q_0	r	R	$\{ (q_0, RR) \}$
q_0	v	R	$\{ (q_0, RV) \}$
q_0	b	Z_0	$\{ (q_1, \varepsilon) \}$
q_1	ε	Z_0	$\{ (q_1, \varepsilon) \}$

Stato	Input	Stack	Transizioni
q_0	r	V	$\{ (q_0, VR) \}$
q_0	v	V	$\{ (q_0, VV)\}$
q_0	b	R	$\{ (q_1, R) \}$
q_0	b	V	$\{ (q_1, V) \}$
q_1	v	V	$\{ (q_1, \varepsilon) \}$
q_1	r	R	$\{ (q_1, \varepsilon) \}$

Grammatiche libere ed automi a pila

Teoremi

- Data una grammatica context-free G, esiste un automa push down M tale che L(M) = L(G).
- Dato un automa push down M, esiste una grammatica context-free G tale che L(G) = L(M).

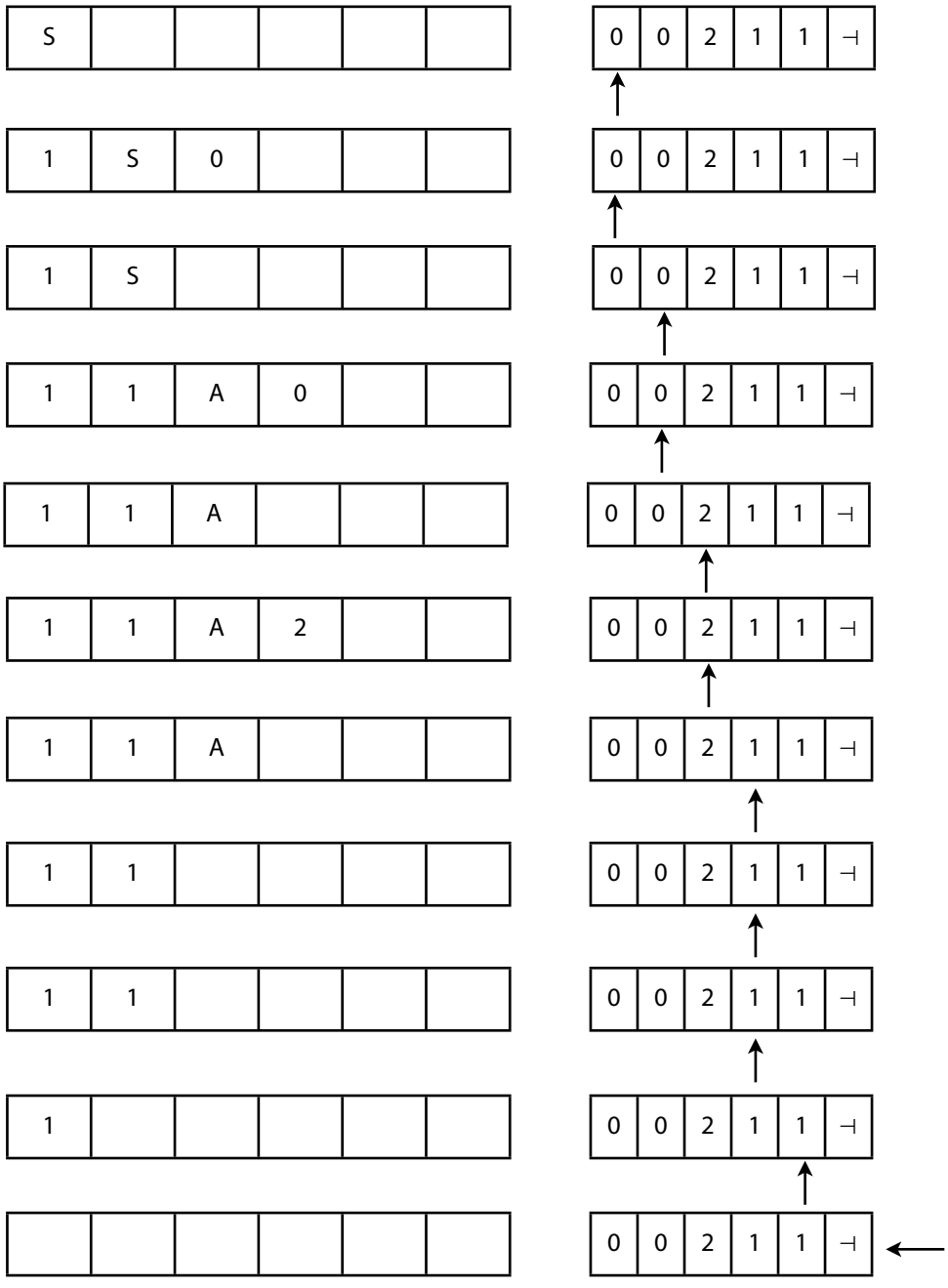
Nota: d'ora in poi supponiamo che la stringa da riconoscere abbia il carattere di fine stringa ⊣.
IDEA! Utilizziamo lo stack per simulare una derivazione leftmost!

Grammatica context-free: **Derivazione:**

S -> 0S1 | 0A1 S ⇒ 0S1 ⇒ 00A11 ⇒ 002A11⇒ 00211

A -> 2A | ε

Abbiamo quindi: 00211⊣. Impiliamo sullo stack le produzioni al contrario.



Costruire l'automa partendo dalla grammatica

Come abbiamo fatto nell'esempio sopra? Ci sono alcune regole:

1) Inserire S sullo stack definendo una mossa spontanea $\delta(q, \varepsilon, Z_0) = \{ (q,S) \}$.
2) Per ogni produzione del tipo A -> YX_1 se A è il simbolo in cima allo stack si inserisce la stringa X_1Y (cioè la produzione scritta al contrario) al posto della variabile A, senza avanzare nella testina di input. Si ha quindi una produzione $\delta(q, \varepsilon, A) = \{ (q, X_1Y) \}$.
3) Per ogni simbolo terminale a, se a è il simbolo in input e il simbolo sul top dello stack si avanza con la testina di input e si effettua una pop. Si ha quindi la produzione $\delta(q, a, a) = \{ (q,\varepsilon) \}$.

Quindi nell'esempio precedente:

Grammatica context-free:

S -> 0S1 | 0A1

A -> 2A | ε

Automa:

$\delta(q, \varepsilon, Z_0) = \{ (q, S) \}$

$\delta(q, \varepsilon, S) = \{ (q, 1A0), (q, 1S1) \}$

$\delta(q, \varepsilon, A) = \{ (q, A2), (q, \varepsilon) \}$

$\delta(q, 0, 0) = \{ (q, \varepsilon) \}$

$\delta(q, 1, 1) = \{ (q, \varepsilon) \}$

$\delta(q, 2, 2) = \{ (q, \varepsilon) \}$

Quindi abbiamo la settupla:

$< \{q\}, \{0,1,2\}, \{Z_0, A, S, 0, 1, 2\}, \delta, q, Z_0, \Phi >$

L'automa generato:

- ha un solo stato
- accetta per stack vuoto
- non è deterministico

Vi sono transizioni differenti che possono portare all'accettazione della stringa 00211⊣.

Proprietà di chiusura dei linguaggi liberi

	regolari	context-free	context-free deterministici
Unione	$R_1 \cup R_2 \in Reg$	$L_1 \cup L_2 \in C\text{-}F$	$D_1 \cup D_2 \notin C\text{-}F_{Det}$ $D \cup R \in C\text{-}F_{Det}$
Concatenazione	$R_1.R_2 \in Reg$	$L_1.L_2 \in C\text{-}F$	$D_1.D_2 \notin C\text{-}F_{Det}$ $D.R \in C\text{-}F_{Det}$
Chiusura di Kleene	$R^* \in Reg$	$L^* \in C\text{-}F$	$D^* \notin C\text{-}F_{Det}$
Riflessione	$R^R \in Reg$	$L^R \in C\text{-}F$	$D^R \notin C\text{-}F_{Det}$
Complemento	$\Sigma^*\text{-}R \in Reg$	$\Sigma^*\text{-}L \notin C\text{-}F$	$\Sigma^*\text{-}D \in C\text{-}F_{Det}$
Intersezione	$R_1 \cap R_2 \in Reg$	$L_1 \cap L_2 \notin C\text{-}F$ $L \cap R \in C\text{-}F$	$D \cap R \in C\text{-}F_{Det}$

Possibili domande di teoria

1) Definizione di automa finito deterministico
2) Definizione di automa finito non deterministico
3) Cosa significa che p è equivalente a q? (due stati)
4) Definire la classificazione tra context-free, espressioni regolari, etc (quale linguaggio include quale)
5) Enuncia il pumping lemma
6) Definizione di grammatica context-free
7) Definizione di grammatica context-free lineare, unilineare destra, sinistra, ricorsiva, etc

Possibili esercizi

1) Definire un automa push down che riconosca questo linguaggio: $\{a^i b^j \mid 0 < i < j\}$
2) Minimizzare gli stati di un automa
3) Trasformare una grammatica unilineare destra in unilineare sinistra e viceversa
4) Costruire un automa partendo da una grammatica context - free
5) Trovare una espressione regolare partendo da una grammatica context - free
6) Eliminare le regole di copiatura
7) Eliminare epsilon produzioni/mosse

Riassunto da: Enrico Mensa

Esercizi sulle algebre ed i linguaggi

Dimostrare
$(L^*)^* = L^*$ --> dobbiamo quindi dimostrare $(L^*)^* \subseteq L^*$ & $L^* \subseteq (L^*)^*$
La seconda parte è piuttosto banale, poiché per definizione $L \subseteq L^*$.
Per la prima parte, prendiamo un $x \in (L^*)^*$ e dimostriamo che $x \in L^*$.
Se $x = \varepsilon$, $x \in L^*$, se $\neq \varepsilon$ allora è scrivibile come $x_1....x_n$ ognuna appartenente ad L^*.
Allora dato che $x_1 \cdot ... \cdot x_n = L^* \cdot .... \cdot L^*$ (n volte) $= L^*$, $x \in L^*$.

Scrivere una espressione tale che
Ogni occorrenza di a sia seguita da almeno due occorrenze di b (sull'alfabeto $\Sigma = \{a,b\}$)
Soluzione: $(abb^+)^* \mid (\varepsilon bb^+)^*$

Contenga almeno una occorrenza di aa ed una di bb nell'ordine (sempre su Σ)
Soluzione: $(a|b)^*(aa)(a|b)^*(bb)(a|b)^*$