

Prof. Marco Cesati

SITO: <https://al20.sprg.uniroma2.it>

GESTIONE PROVE D'ESAME: <http://gocu.sprg.uniroma2.it>

E-MAIL: al@sprg.uniroma2.it

RICEVIMENTO: martedì, dalle 9:30 alle 11:00

STANZA A3-05, TERZO PIANO DELL'EDIFICIO INGEGNERIA DELL'INFORMAZIONE

TESTO: Introduction to the Theory of Computation, 3rd ed. (Michael Sipser)

(ESISTE ANCHE UNA VERSIONE IN ITALIANO)

CORREZIONI AL TESTO: <http://math.mit.edu/~sipser/book.html>

03/03/2020

Problema:

Data una qualunque stringa di bit, vogliamo stabilire se rappresenta un multiplo di 3 attraverso una macchina che può memorizzare solo 2 bit.

→ 2 bit implicano $2^2 = 4$ possibili stati differenti; STIAMO CONSIDERANDO LA MACCHINA COME UN AUTOMA A STATI FINITI.

- PARTO DA 0, CHE È UN MULTIPLO DI 3 (= 3K)
- SE A 3K AGGIUNGO UNO 0, HO ANCORA UN MULTIPLO DI 3 (RADOPPIO 3K)
E così via..

A questo meccanismo c'è un limite: come trattare una stringa vuota?
Si utilizza un quarto stato (altrimenti ne bastano 3), da utilizzarsi come stato d'ingresso.

13/03/2020

ALFABETO (Σ) = insieme finito di elementi chiamati "simboli".

STRINGA SOPRA Σ = sequenza finita di caratteri dentro un alfabeto.

Σ^* = insieme di tutte le stringhe possibili costruibili sopra l'alfabeto Σ .
È chiaramente un insieme infinito.

LUNGHEZZA DI UNA STRINGA = numero di caratteri che costituiscono la stringa.

STRINGA NULLA = stringa senza alcun simbolo = " " = ϵ

Ha lunghezza pari a zero.

$$\forall \Sigma \quad \epsilon \in \Sigma^*$$

SOTTOSTRINGA DI UNA STRINGA s = stringa i cui caratteri compaiono nel giusto ordine e consecutivamente in s .

CONCATENAZIONE DI DUE STRINGHE : produce una terza stringa che contiene tutti i caratteri della prima stringa nel giusto ordine e consecutivi e tutti i caratteri della seconda stringa nel giusto ordine e consecutivi.

ORDINE LESSICOGRAFICO = ordine alfabetico.

Esempio nella lingua italiana:

alfa < beta

STRING ORDER = ordine secondo cui una stringa più corta precede sempre una più lunga. A parità di lunghezza si segue l'ordine lessicografico.

Esempio nella lingua italiana:

alfa > bit

LINGUAGGIO SOPRA Σ = Sottoinsieme di stringhe sopra Σ ($L \subseteq \Sigma^*$)

Esempio:

$$\Sigma = \{0, 1\}$$

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$$

$$L = \{w \in \Sigma^* \mid w \text{ decodifica in binario un numero divisibile per } 3\}$$

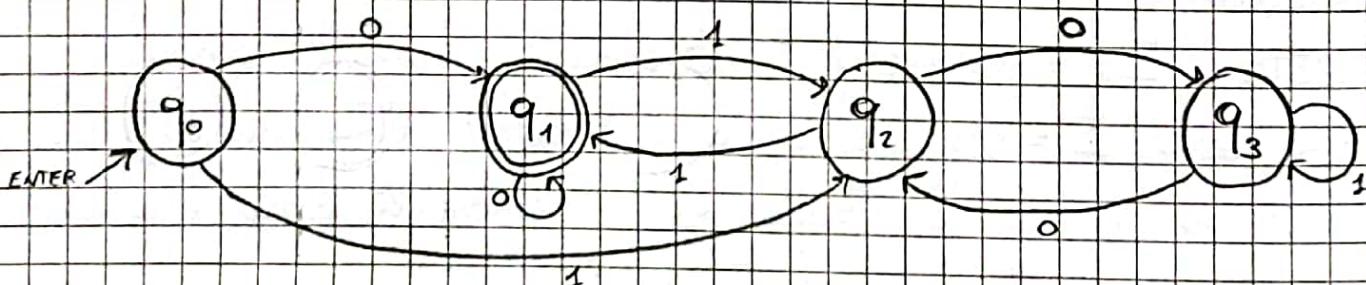
$$L = \{0, 11, 110, 1001\}$$

$$\epsilon \notin L$$

$$1 \notin L$$

Esiste una relazione tra i linguaggi e i problemi. In questo caso abbiamo a che fare con un problema decisionale: dato un numero, decidere se è divisibile per 3 oppure no.

Abbiamo introdotto questo problema nella scorsa lezione e possiamo affermare che è risolvibile con una macchina con memoria RAM pari a 2 bit (che identificano $2^2 = 4$ stati):



STADIO q_0 : stringa vuota

STADIO q_1 : $n = 3k$

STADIO q_2 : $n = 3k+1$

STADIO q_3 : $n = 3k+2$

Se alla fine della scansione la macchina si trova nello stato q_2 , allora la stringa rappresenta un multiplo di 3 e appartiene al linguaggio. Negli altri casi no.

AUTOMA A STATI FINITI (DFA) = macchina M ^{DETERMINISTICO} ^{astratta} costituita da 5 componenti:

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

Q = stati della macchina (q_0, q_1, q_2, \dots)

Σ = alfabeto, cioè l'insieme dei simboli che la macchina può leggere e interpretare

$\delta: Q \times \Sigma \rightarrow Q$
 $(q, \sigma) \rightarrow q'$

È una funzione che, dati uno stato e un simbolo, restituisce un nuovo stato.

È la cosiddetta funzione di TRANSIZIONE dell'automa.

$q_0 \in Q$ = start state, lo stato in cui si trova la macchina prima di leggere il primo simbolo dato in input.

$F \subseteq Q$ = insieme degli stati di accettazione. Se dopo la lettura di tutti i simboli dati in input, la macchina si trova in uno di questi stati, riconosce la stringa come appartenente al linguaggio (la "accetta").

Nell'esempio precedente:

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{0, 1\}$$

STATO INIZIALE = q_0

$$F = \{q_1\}$$

	0	1
q_0	q_1	q_2
q_1	q_1	q_2
q_2	q_3	q_1
q_3	q_2	q_3

Una macchina riconosce sempre e soltanto un solo linguaggio.

CASI PARTICOLARI:

$\rightarrow F = \emptyset \Rightarrow L(M) = \emptyset$ \Rightarrow il linguaggio riconosciuto dalla macchina non contiene alcuna stringa

$\rightarrow F = Q \Rightarrow L(M) = \Sigma^*$ \Rightarrow il linguaggio riconosciuto dalla macchina contiene tutte le stringhe appartenenti a Σ^*

Dove $L(M) = \{ w \in \Sigma^* \mid M \text{ "ACCETTA"} w \}$

NB: Se lo stato iniziale è uno stato di accettazione, la stringa vuota appartiene al linguaggio.

Definizione (accettazione di un automa):

Dato un automa a stati finiti $M = (Q, \Sigma, \delta, q_0, F)$, M accetta una certa stringa $w \in \Sigma^*$ se esiste una sequenza di stati $r_0, r_1, \dots, r_n \in Q$ tali che:

$$1) r_0 = q_0$$

$$2) \forall i = 0, \dots, n-1 \quad \delta(r_i, w_{i+1}) = r_{i+1}, \text{ dove } w_i \text{ è l'}i\text{-esimo simbolo di } w$$

$$3) r_n \in F$$

Definizione:

Un linguaggio A è detto REGOLARE se un certo DFA lo riconosce ($L(M) = A$).

Come si fa a determinare se un certo linguaggio è regolare?

Costruire un DFA che lo riconosce.

Come si fa a determinare se un certo linguaggio NON è regolare?

Bisognerebbe dimostrare che non esiste alcun automa a stati finiti che riconosca il linguaggio (si tratta di dimostrazioni complicate che vedremo dopo).

Operazioni sui linguaggi regolari:

Siano A, B due linguaggi regolari.

- $A \cup B = \{w \mid w \in A \vee w \in B\}$
- $A \circ B = \text{concatenazione} = \{xy \mid x \in A \wedge y \in B\}$
- $A^* = \{x_1, x_2, \dots, x_K \mid K \geq 0 \wedge x_i \in A \quad \forall i=1, \dots, K\} = \text{operatore di KLEENE}$
 $\epsilon \in A^* \quad \forall A$; un esempio è proprio Σ^*
- ALTRÒ ESEMPIO: $A = \{01, 10\} \Rightarrow A^* = \{\epsilon, 01, 10, 0110, 1001, \dots\}$
- NB: $A^* \neq \Sigma^*$; in questo caso, $01 \notin A^*$
- $A \cap B = \{w \mid w \in A \wedge w \in B\}$

Teorema:

Se A, B sono linguaggi regolari $\Rightarrow A \cup B$ è un linguaggio regolare.

Dim:

A regolare $\Rightarrow \exists$ DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ t.c. $L(M_1) = A$

B regolare $\Rightarrow \exists$ DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ t.c. $L(M_2) = B$

Dovrò dimostrare che ~~esiste~~ $\exists M = (Q, \Sigma, \delta, q_0, F)$ che, data una stringa w , termini in uno stato di accettazione \searrow se almeno una delle due macchine M_1, M_2 termina in uno stato di accettazione.

Il problema è che non è possibile simulare M_1 e M_2 in momenti diversi poiché, una volta che sono stati letti tutti i simboli di w , non esiste alcun meccanismo che permetta di "riavviare il master". Perciò è necessario simulare M_1 e M_2 in parallelo. Vediamo come:

$$\rightarrow Q = Q_1 \times Q_2$$

$\rightarrow \Sigma$ è un linguaggio comune ad A e B , quindi non darà problemi.

$$\rightarrow \delta((r_1, r_2), \sigma) = (\delta_1(r_1, \sigma), \delta_2(r_2, \sigma))$$

$$\rightarrow q_0 = (q_1, q_2)$$

$$\rightarrow F = (F_1 \times Q_2) \cup (Q_1 \times F_2) \rightarrow \text{QUALUNQUE STATO DI ACCETTAZIONE DI } M_1 \text{ CON QUALUNQUE STATO DI } M_2 + \text{QUALUNQUE STATO DI } M_1 \text{ CON QUALUNQUE STATO DI ACCETTAZIONE DI } M_2$$

A questo punto resta solo da provare che la DFA M rispetti la definizione di accettazione (dimostrazione lasciata per esercizio).

Teorema:

Se A, B sono linguaggi regolari $\Rightarrow A \cap B$ è un linguaggio regolare.

Dim:

A regolare $\Rightarrow \exists$ DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ t.c. $L(M_1) = A$

B regolare $\Rightarrow \exists$ DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ t.c. $L(M_2) = B$

Dovrò dimostrare che $\exists M = (Q, \Sigma, \delta, q_0, F)$ che, data una stringa w , termini in uno stato di accettazione se e solo se entrambe le macchine M_1, M_2 terminano in uno stato di accettazione.

$$\rightarrow Q = Q_1 \times Q_2$$

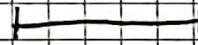
$\rightarrow \Sigma$ è un linguaggio comune ad A e B , quindi non dà problemi.

$$\rightarrow \delta((x_1, x_2), \sigma) = (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma))$$

$$\rightarrow q_0 = (q_1, q_2)$$

$$\rightarrow F = F_1 \times F_2$$

A questo punto resta solo da provare che la DFA M rispetti la definizione di accettazione (dimostrazione lasciata per esercizio). ■



Adesso vogliamo dimostrare che, se A, B sono linguaggi regolari $\Rightarrow A \circ B$ è un linguaggio regolare. Tuttavia c'è un problema. Vediamo con un esempio:

$$\Sigma = \{0, 1\}$$

$$A = \{01, 011\}$$

$$B = \{10, 101\}$$

$$w = \underbrace{01101}_{x \in A, y \in B} \in A \circ B$$

$x \in A, y \in B \rightarrow$ Se prendessi questa suddivisione, non si direbbe che la stringa appartenga al linguaggio $A \circ B$!

$x \in A, y \in B \rightarrow$ Per questo motivo, la macchina M deve accettare la stringa ~~anche~~ nel linguaggio $A \circ B$

Quello che vogliamo fare è accettare la stringa se esiste un qualunque modo di scomporla in " x concatenato y " tale che $x \in A \wedge y \in B$. Dobbiamo provarli tutti. Ma è possibile fare questa cosa?

14/03/2020

AUTOMA A STATI FINITI NON DETERMINISTICO (NFA) = macchina M_a
struttura costituita da 5 componenti:

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

La differenza sostanziale da una DFA sta nella definizione della funzione δ :

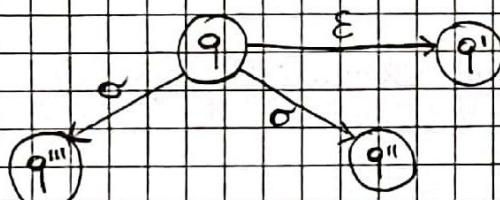
$$\delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q) = 2^Q \rightarrow \text{INSIEME DELLE PARTI DI } Q$$
$$(q, \sigma) \xrightarrow[\sigma \in \Sigma \vee \sigma = \epsilon]{} \{q', q'', q''', \dots\} \subseteq Q$$

Def (computazione accettante di un automa non deterministico):

È una sequenza di stati $r_0, r_1, \dots, r_n \in Q$ t.c.:

- 1) $r_0 = q_0$
- 2) $\forall i=0, \dots, n-1 \quad r_{i+1} \in \delta(r_i, y_{i+1})$ SIMBOLO SUCCESSIVO DELLA STRINGA CHE DEVE ESSERE LETTO DA PARTE DELL'AUTOMA $y_{i+1} = \epsilon \vee y_{i+1} = w_j$
- 3) $r_n \in F$

La cosa interessante è che a ogni iterazione non viene necessariamente consumato un carattere della stringa data in input (caso $y_{i+1} = \epsilon$).



L'automa può decidere, partendo dallo stato q , se:

→ Non leggere alcun carattere e passare automaticamente allo stato q'

→ Leggere il carattere σ e passare allo stato q''

→ Leggere lo stesso carattere σ e passare allo stato q'''

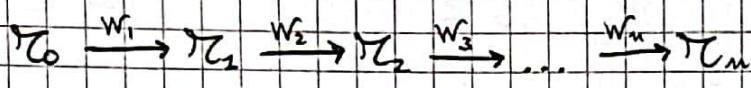
In realtà vedremo che può fare le tre cose in parallelo.

L'automa accetta una stringa w se una qualunque delle possibilità offerte da questa formulazione porta a consumare tutto l'input "in uno stato di accettazione".

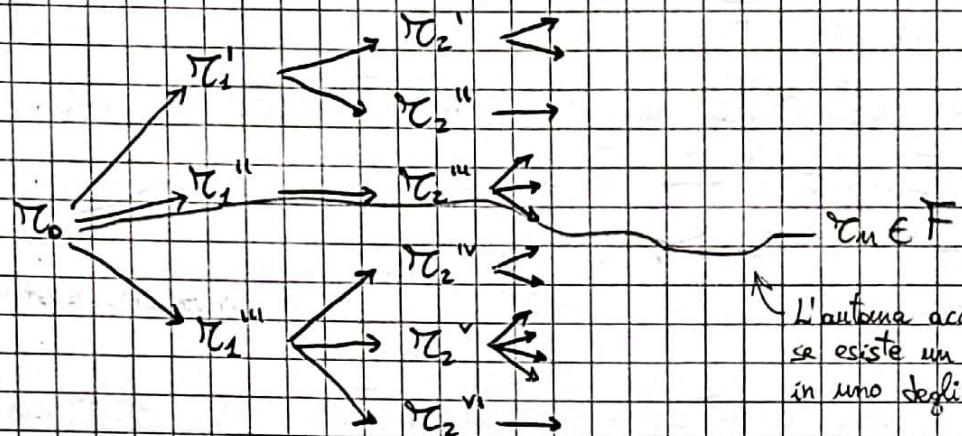
Vediamo questa proprietà graficamente:

CONFRONTO TRA DFA E NFA:

DFA



NFA



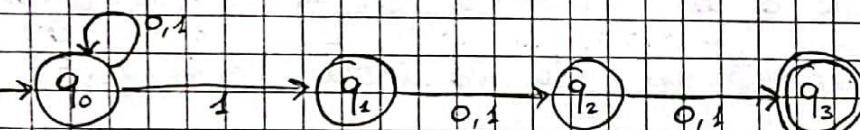
Esercizio:

Esibire un NFA che accetta ogni stringa binaria che ha un 1 nella terza posizione dalla fine.

ESEMPI: 000 ∈ L

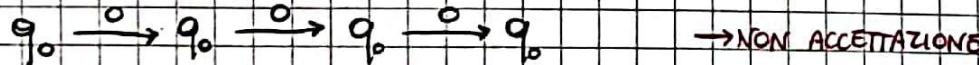
01000 ∈ L

10000100 ∈ L

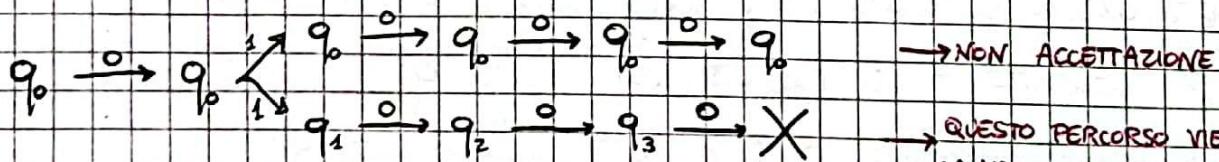


Vediamo cosa succede con i tre esempi appena elencati:

1) $w = 000$



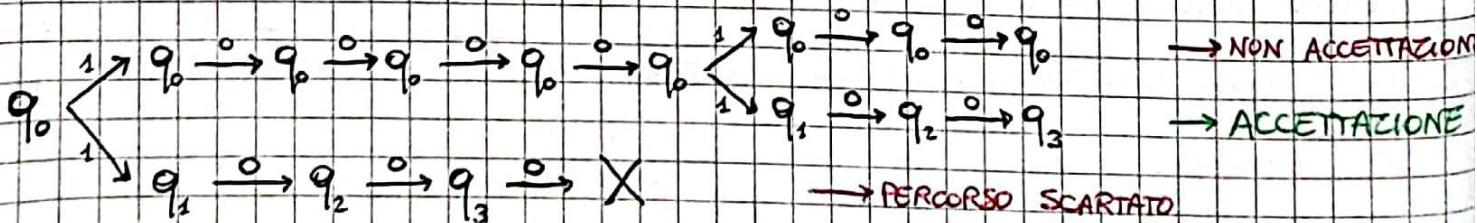
2) $w = 01000$



QUESTO FENOMENO È AMMESSO PERCHÉ $\emptyset \in P(Q)$

QUESTO PERCORSO VIENE BUTTATO PERCHÉ LA MACCHINA, NELLO STATO q_3 , LEGGE IL CARATTERE 0 MA NELL'SCHEMA DELL'AUTOMA NON CI È UNA FRECCIA CHE ESCE DA q_3 CON INPUT 0

3) $w = 10000100$



Teorema:

Ogni automa a stati finiti NON deterministico N ha un automa a stati finiti deterministico M equivalente, ovvero il linguaggio riconosciuto da N coincide col linguaggio riconosciuto da M ($L(N) = L(M)$).

Idea della dim:

M simula "in parallelo" ogni percorso (COMPUTATION PATH) di N .
 $N = (Q, \Sigma, \delta, q_0, F) \Rightarrow M = (Q', \Sigma, \delta', q'_0, F)$

→ ESPOENZIALMENTE PIÙ GRANDE DI Q

CASO A:

Non ci sono transizioni marcate con la stringa nulla ϵ in N .

- $Q' = 2^Q = \mathcal{P}(Q)$

- Σ È IDENTICO PER $N \in M$

- $\delta': Q \times \Sigma \rightarrow Q'$
 $(R, \sigma) \rightarrow R'$ con $R, R' \subseteq Q$
cioè $R R' \subseteq 2^Q$

$$\delta'(R, \sigma) = \bigcup_{r \in R} \delta(r, \sigma) = \{q \in Q \mid q \in \delta(r, \sigma) \text{ per qualche } r \in R\}$$

effettivamente
appartiene anche a Q'

→ In altre parole: se (R, σ) , cioè lo stato da cui sto partendo nella DFA M , è un sottoinsieme di stati di Q , allora arriverò a un nuovo sottoinsieme che corrisponderà a tutti gli stati che sono raggiungibili tramite σ dagli stati di Q tramite la funzione di transizione di N (δ)

- $q'_0 = \{q_0\}$

- $F' = \{R \in Q' \mid R \cap F \neq \emptyset\}$ → In altre parole: la DFA M accetta se alla fine arrivo a un sottoinsieme di stati di N che include uno stato di accettazione

CASO B:

Ci sono delle transizioni marcate con la stringa nulla ϵ in N .

→ Le frecce che descrivono queste transizioni non sono direttamente rappresentabili nella macchina deterministica. Dobbiamo trovare un modo per costruire una computazione equivalente.

IDEA: definire una FUNZIONE DI CHIUSURA, la quale permette di rappresentare, a partire da un certo sottoinsieme di Q , tutti gli stati raggiungibili con una transizione marcata da una stringa nulla.

Sia $R \subseteq Q$ ($R \subseteq Q'$).

$E(R) := \{q \in Q \mid q \text{ può essere raggiunto da } R \text{ utilizzando zero, una o tante frecce marcate da } \epsilon\}$ (E-ARROW) ?

$R \subseteq E(R)$ perché è un insieme raggiungibile da se stesso con zero E-arrow.

Ora vediamo come sono definiti i parametri della DFA M in questo caso:

- $Q' = 2^Q$
- Σ È IDENTICO PER $N \in M$
- $\delta'(R, \sigma) = \bigcup_{r \in R} E(\delta(r, \sigma))$ dove $R \subseteq Q$
- $q_0' = E(\{q_0\})$
- $F' = \{R \in Q' \mid R \cap F \neq \emptyset\}$

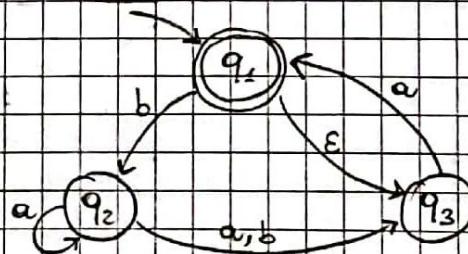
Alessio resta da provare che, con questa costruzione della DFA M , si ottiene che $L(N) = L(M)$ (dimostrazione lasciata per esercizio). ■

Corollario:

Un linguaggio A è regolare $\Leftrightarrow \exists$ NFA M t.c. $L(M) = A$.

Esercizio:

Convertire il seguente NFA in DFA.



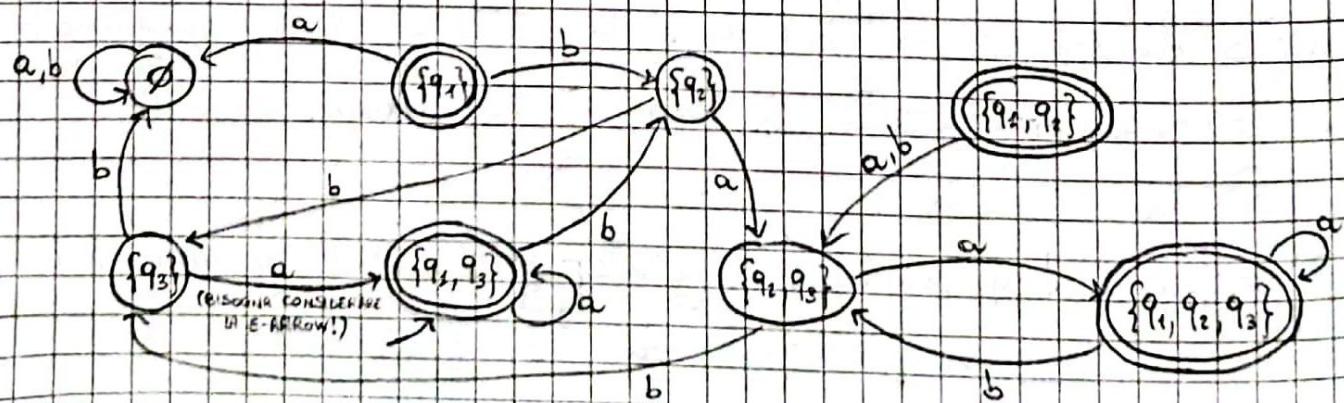
$$Q = \{q_1, q_2, q_3\} \Rightarrow Q' = 2^Q$$

$$F = \{q_1\} \Rightarrow F' = \{\{q_1\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_2, q_3\}\}$$

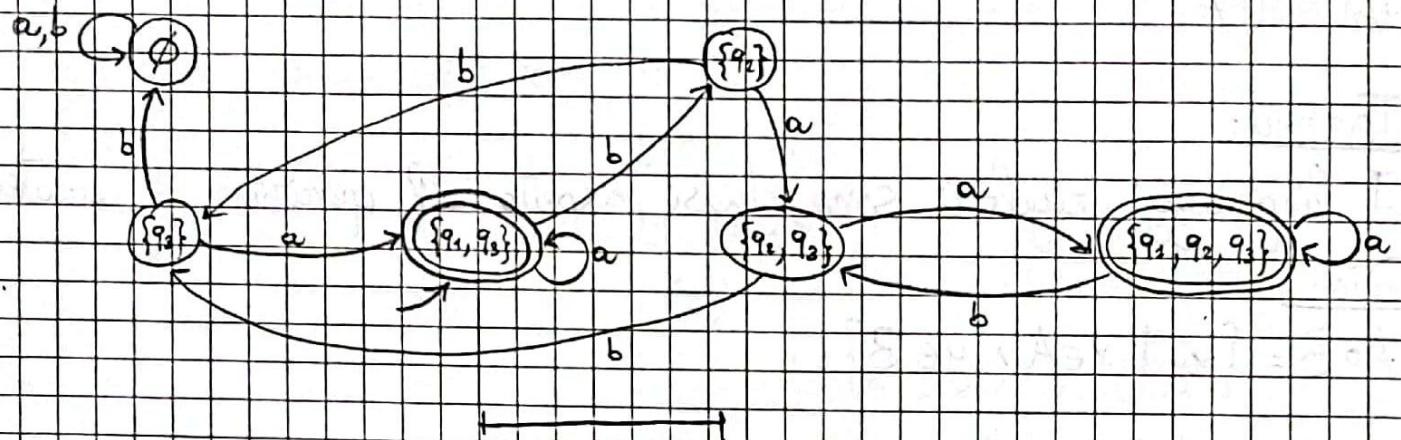
$$E(\{q_1\}) = \{q_2, q_3\}, \quad E(\{q_3\}) = \{q_3\},$$

$$E(\{q_1, q_2\}) = \{q_1, q_2, q_3\} \dots$$

$$q_0 = q_1 \Rightarrow q_0' = E(\{q_1\}) = \{q_1, q_2, q_3\}$$



A questo punto è possibile eliminare gli stati inaccessibili che, cioè, non hanno alcuna freccia entrante.

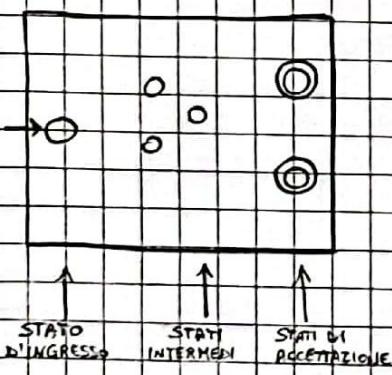


Consideriamo nuovamente il seguente teorema:

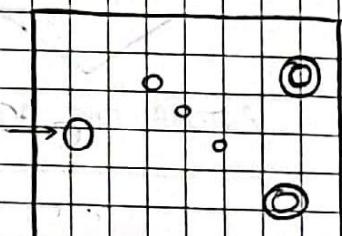
Se A, B sono linguaggi regolari: $\Rightarrow A \cup B$ è un linguaggio regolare.

Si può sfruttare l'equivalenza tra DFA e NFA e si può dimostrare più
il teorema più semplicemente con l'utilizzo di un NFA:

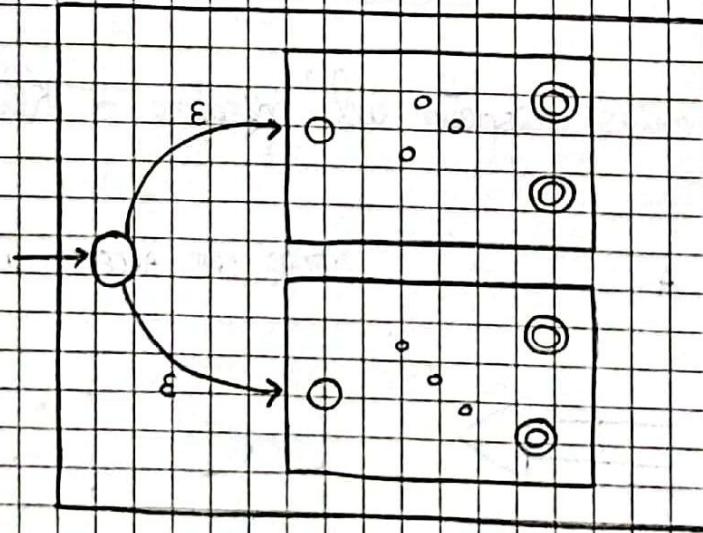
AUTOMA CHE ACCETTA
IL LINGUAGGIO A



AUTOMA CHE ACCETTA
IL LINGUAGGIO B



AUTOMA CHE ACCETTA IL LINGUAGGIO $A \cup B$



D'altra parte, però, non è per niente semplice dimostrare la chiusura dei linguaggi regolari ~~tecnicamente~~ rispetto all'intersezione attraverso un NFA.

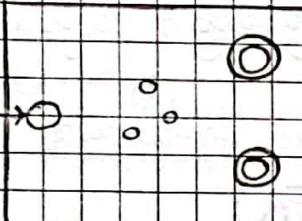
Teorema:

I linguaggi regolari sono chiusi rispetto all'operazione di concatenazione.

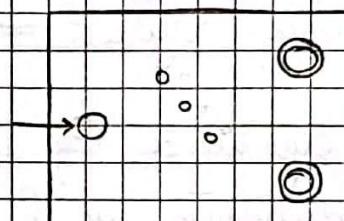
Dim:

$$A \circ B = \{xy \mid x \in A \wedge y \in B\}$$

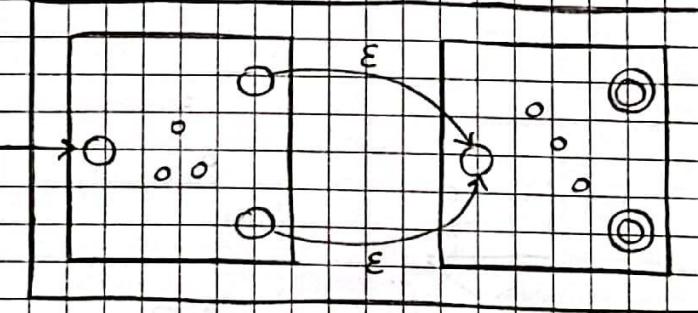
AUTOMA CHE ACCETTA
IL LINGUAGGIO A



AUTOMA CHE ACCETTA
IL LINGUAGGIO B



AUTOMA CHE ACCETTA IL LINGUAGGIO $A \circ B$

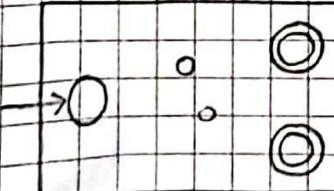


Teorema:

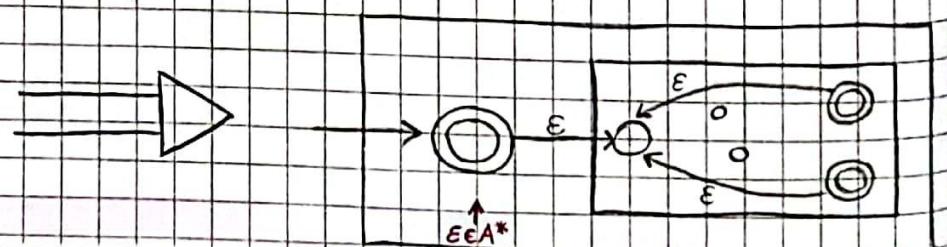
I linguaggi regolari sono chiusi rispetto all'operazione di Kleene (*).

Dim:

AUTOMA CHE ACCETTA IL LINGUAGGIO A



AUTOMA CHE ACCETTA IL LINGUAGGIO A^*



15/03/2020

Def (espressioni regolari - REX):

Sia Σ un alfabeto e siano R_1, R_2 espressioni regolari (REX). Allora anche le seguenti espressioni sono regolari:

- 1) $a \quad \forall a \in \Sigma \quad \rightsquigarrow L(a) = \{a\}$
- 2) $\epsilon \quad \rightsquigarrow L(\epsilon) = \{\epsilon\}$
- 3) $\emptyset \quad \rightsquigarrow L(\emptyset) = \emptyset$
- 4) $(R_1 \cup R_2) \quad \rightsquigarrow L((R_1 \cup R_2)) = L(R_1) \cup L(R_2)$
- 5) $(R_1 \circ R_2) \quad \rightsquigarrow L((R_1 \circ R_2)) = L(R_1) \circ L(R_2)$
- 6) $(R_1)^* \quad \rightsquigarrow L((R_1)^*) = (L(R_1))^*$

ORDINE DI PRECEDENZA DEGLI OPERATORI:

- 1) *
- 2) \circ
- 3) \cup

ESEMPIO: $R_1^* \cup R_2 \equiv ((R_1)^* \cup R_2)$

NB: $R_1 \cup R_2$ equivalente a $R_1 + R_2$

$R_1 \circ R_2$ equivalente a $R_1 R_2$

$R R^*$ equivalente a R^+

$\underbrace{R \circ R \circ \dots \circ R}_K$ equivalente a R^K

Osservazione:

$R \epsilon \equiv R$ $R \emptyset \equiv \emptyset$

$R \cup \epsilon$ ha come linguaggio associato $L(R) \cup \{\epsilon\}$

$R \cup \emptyset$ ha come linguaggio associato $L(R) \cup \emptyset = L(R)$

$\Rightarrow R \cup \emptyset \equiv R$

$\emptyset^* = \{\epsilon\} \rightarrow$ SI POSSONO CONCATENARE AL PIÙ ZERO ELEMENTI ALL'INTERNO DELL'INSIEME VUOTO; LA CONCATENAZIONE DI ZERO ELEMENTI PRODUCE LA STRINGA VUOTA

Teorema:

Un linguaggio L è regolare $\Leftrightarrow \exists$ una REX R t.c. $L(R) = L$.

Dim:

\Leftarrow) R è una REX $\Rightarrow L(R)$ è regolare

Possiamo utilizzare una costruzione dovuta a McNaughton - Yamada, che si basa sul principio dell'INDUZIONE.

PASSO BASE ($|R| = 1$):

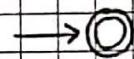
1) $R = a \Rightarrow L(a) = \{a\}$

$\Rightarrow \{a\}$ è regolare ✓

$[|R| = \text{lunghezza di } R]$

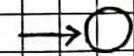


2) $R = \varepsilon \Rightarrow L(\varepsilon) = \{\varepsilon\}$



$\Rightarrow \{\varepsilon\}$ è regolare ✓

3) $R = \emptyset \Rightarrow L(\emptyset) = \emptyset$



$\Rightarrow \emptyset$ è regolare ✓

PASSO INDUTTIVO ($|R| = n > 1$):

4) $R = (R_1 \cup R_2)$ $|R_1| < n, |R_2| < n$

$\Rightarrow L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$

$L(R_1), L(R_2)$ sono linguaggi regolari per ipotesi induttiva

\Rightarrow per il teorema della chiusura dei linguaggi regolari rispetto all'unione, (che abbiamo già dimostrato), $L(R_1) \cup L(R_2)$ è regolare.

5) $R = (R_1 \circ R_2)$

IL DISCORSO È DEL TUTTO ANALOGO

6) $R = (R_1)^*$

□

Esercizio 1:

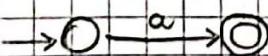
Sia $(ab \cup a)^*$ un'espressione regolare.

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

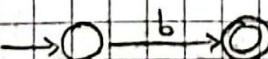
CATENAZIONE DI ZERO O PIÙ VOLTE DELLA STRINGA "ab" E/O DELLA STRINGA "a"
ESEMPIO: "aababaaa"

Dimostrare che $L((ab \cup a)^*)$ è regolare.

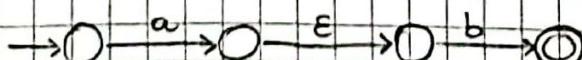
a:



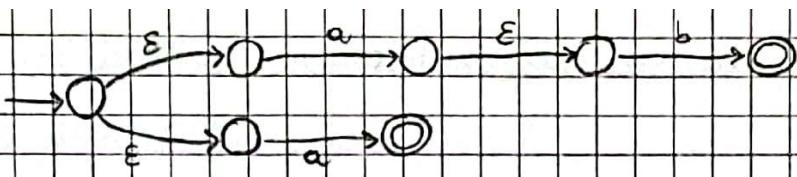
b:



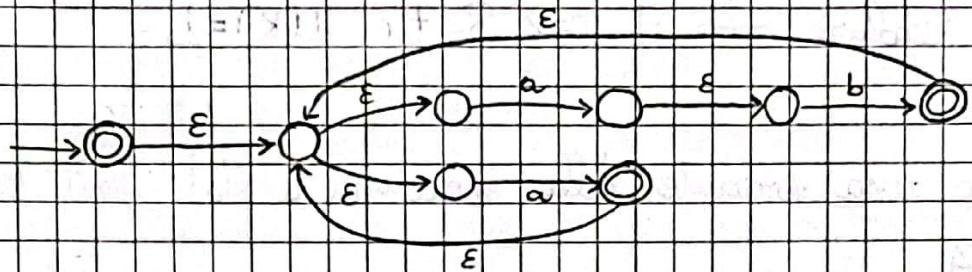
ab:



$ab \cup a:$



$(ab \cup a)^*$:



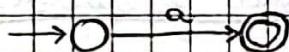
$\Rightarrow L((ab \cup a)^*)$ è regolare

Esercizio 2:

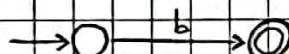
Ricostruire l'automa dell'espressione regolare $R = (a \cup b)^* aba$.

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

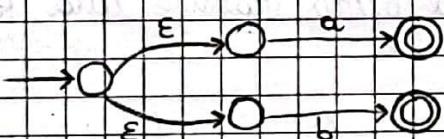
a:



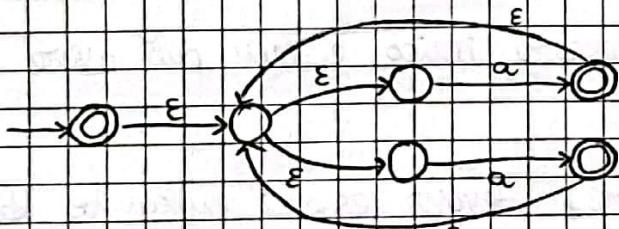
b:



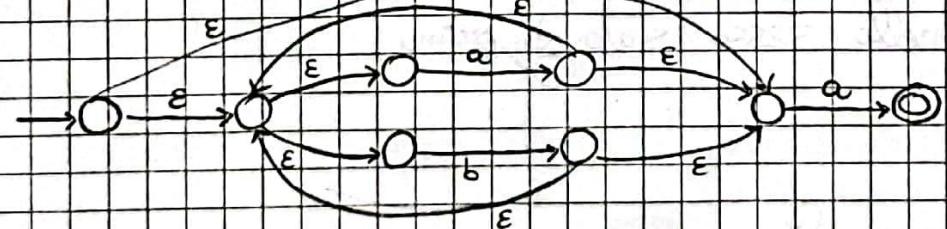
$a \cup b:$



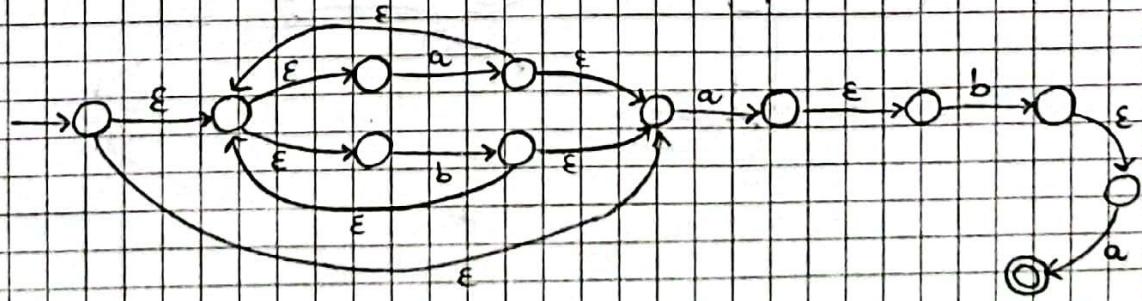
$(a \cup b)^*:$



$(a \cup b)^* a:$



$(a \cup b)^* aba:$



Adesso dimostriamo la seconda implicazione dell'ultimo Teorema che abbiamo enunciato.

$$\Rightarrow L \text{ è regolare} \stackrel{?}{\Rightarrow} \exists \text{ REX } R \text{ t.c. } L(R) = L$$

IDEA:

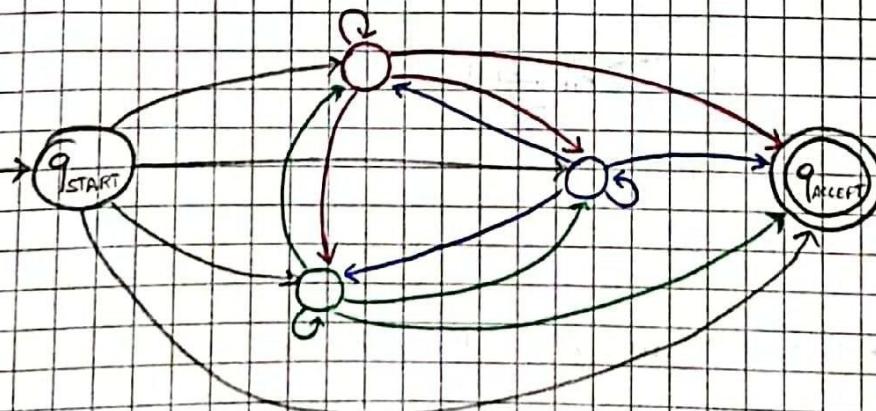
- Definiamo una variante della macchina a stati finiti non deterministica: il GNFA.
- Mostriamo come da ogni GNFA sia possibile derivare una REX equivalente.
- Mostriamo come ogni GNFA possa essere costruito a partire da un DFA.
- Siccome ogni linguaggio regolare è riconosciuto da un DFA, otteniamo che da ogni linguaggio regolare è possibile costruire una REX equivalente.

Def (GNFA - automa a stati finiti non deterministico generalizzato):

È una macchina astratta che deriva dal NFA ma con delle differenze sostanziali:

- 1) È possibile transitare da uno stato all'altro tramite una REX (quindi con più simboli alla volta).
- 2) Lo stato di ingresso non può avere alcuna freccia entrante.
- 3) Lo stato di accettazione deve essere unico e non può avere alcuna freccia uscente.
- 4) Per ogni coppia di stati intermedi devono esserci entrambe le transizioni.
- 5) Ogni stato intermedio deve avere un SELF-LOOP, ovvero una transizione che ritorna nello stesso stato di prima.

Esempio:



Se una delle transizioni del GNFA è inutile, si può marcare con \emptyset che, di fatto, è una REX.

Diamo adesso una DEFINIZIONE FORMALE del GNFA:

È una macchina astratta M costituita da 5 componenti:

$M = (Q, \Sigma, \delta, q_{\text{START}}, q_{\text{ACCEPT}})$, dove la funzione di transizione δ è definita nel seguente modo:

$$\delta: (Q \setminus \{q_{\text{ACCEPT}}\}) \times (\Sigma \setminus \{q_{\text{START}}\}) \rightarrow Q$$

Dove $R := \{\text{REX } R \text{ sopra l'alfabeto } \Sigma\}$

Def (computazione accettante di un GNFA):

Sia M un GNFA e sia $w \in \Sigma^*$ una stringa tale che

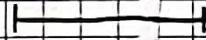
$$w = w_1 w_2 \dots w_k, \quad w_i \in \Sigma^* \quad \forall i = 1, \dots, k$$

Una computazione accettante di M è una sequenza di stati $r_0, \dots, r_k \in Q$ t.c.:

$$1) r_0 = q_{\text{START}}$$

$$2) w_i \in L(\delta(r_{i-1}, r_i)) \quad \forall i = 1, \dots, k \rightarrow w_i \text{ DEVE APPARTENERE AL LINGUAGGIO GENERATO DALL'ESPRESSIONE REGOLARE CHE È ASSOCIATA ALLA FRECCIA CHE FA TRANSIRE DALLO STATO } r_{i-1} \text{ ALLO STATO } r_i;$$

$$3) r_k = q_{\text{ACCEPT}}$$



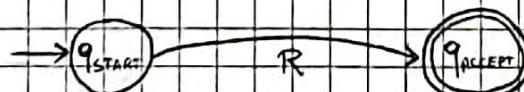
Ora vogliamo mostrare come, a partire da un GFA M sia possibile definire una REX R equivalente, cioè tale che $L(M) = L(R)$.

Teorema:

Per ogni GFA $M \exists \exists$ REX R tale che $L(M) = L(R)$.

Osservazione:

Vale anche il viceversa: da ogni REX R si può costruire una GNFA M equivalente.



Dim Teorema:

La dimostrazione si effettua per induzione.

PASSO BASE ($|Q| = 2$): $\rightarrow q_{\text{START}} \neq q_{\text{ACCEPT}}$, dato che hanno proprietà differenti

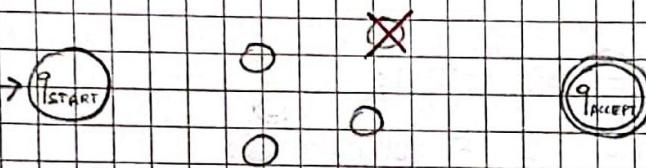
$$M = \begin{array}{c} \textcircled{1} \\ \xrightarrow{R} \textcircled{2} \end{array}$$

\Rightarrow LA REX EQUIVALENTE È PROPRIO
L' "ETICHETTA" DELLA FRECCIA

$$\rightarrow R \in R \quad \checkmark$$

PASSO INDUTTIVO ($|Q| \leq n$):

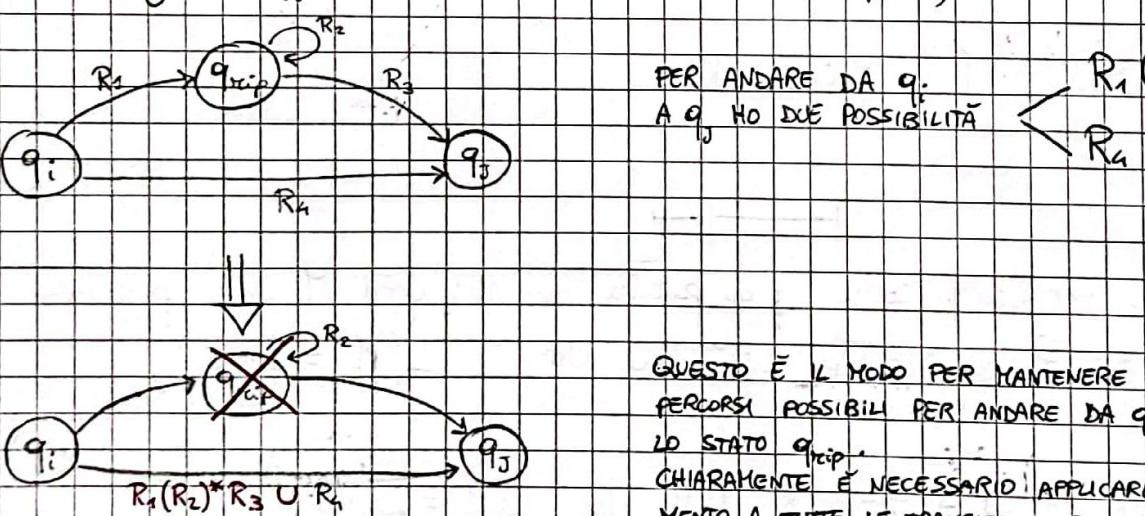
Consideriamo una macchina con $n+1$ stati:



Eliminiamo uno stato e riadattiamo le etichette delle transizioni rimanenti in modo da ottenere una GNFA equivalente; proseguiamo induttivamente in questo modo fino a tornare nel passo base. Abbiamo così trovato una REX R equivalente alla macchina da cui siamo partiti.

Ma come si fa a riadattare le etichette delle transizioni dopo aver eliminato uno stato dal GNFA?

Consideriamo il seguente scenario (dove vengono mostrate solo le transizioni che vengono effettivamente coinvolte nella modifica):

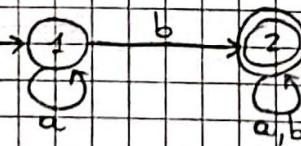


QUESTO È IL MODO PER MANTENERE LA SCELTA TRA I DUE PERCORSI POSSIBILI PER ANDARE DA q_i A q_j ELIMINANDO LO STATO q_{rip} .

CHIARAMENTE È NECESSARIO APPLICARE LO STESSO PROCEDIMENTO A TUTTE LE TRANSIZIONI PRESENTI NEL GNFA.

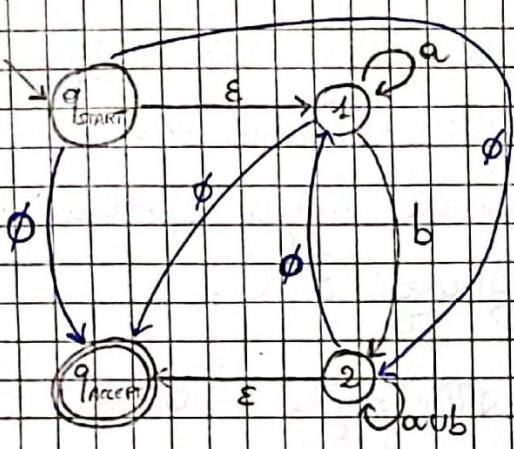
Esercizio:

Dato il seguente DFA:

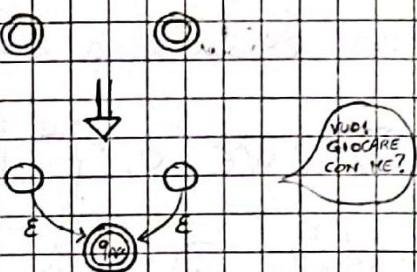


Trovare la REX equivalente.

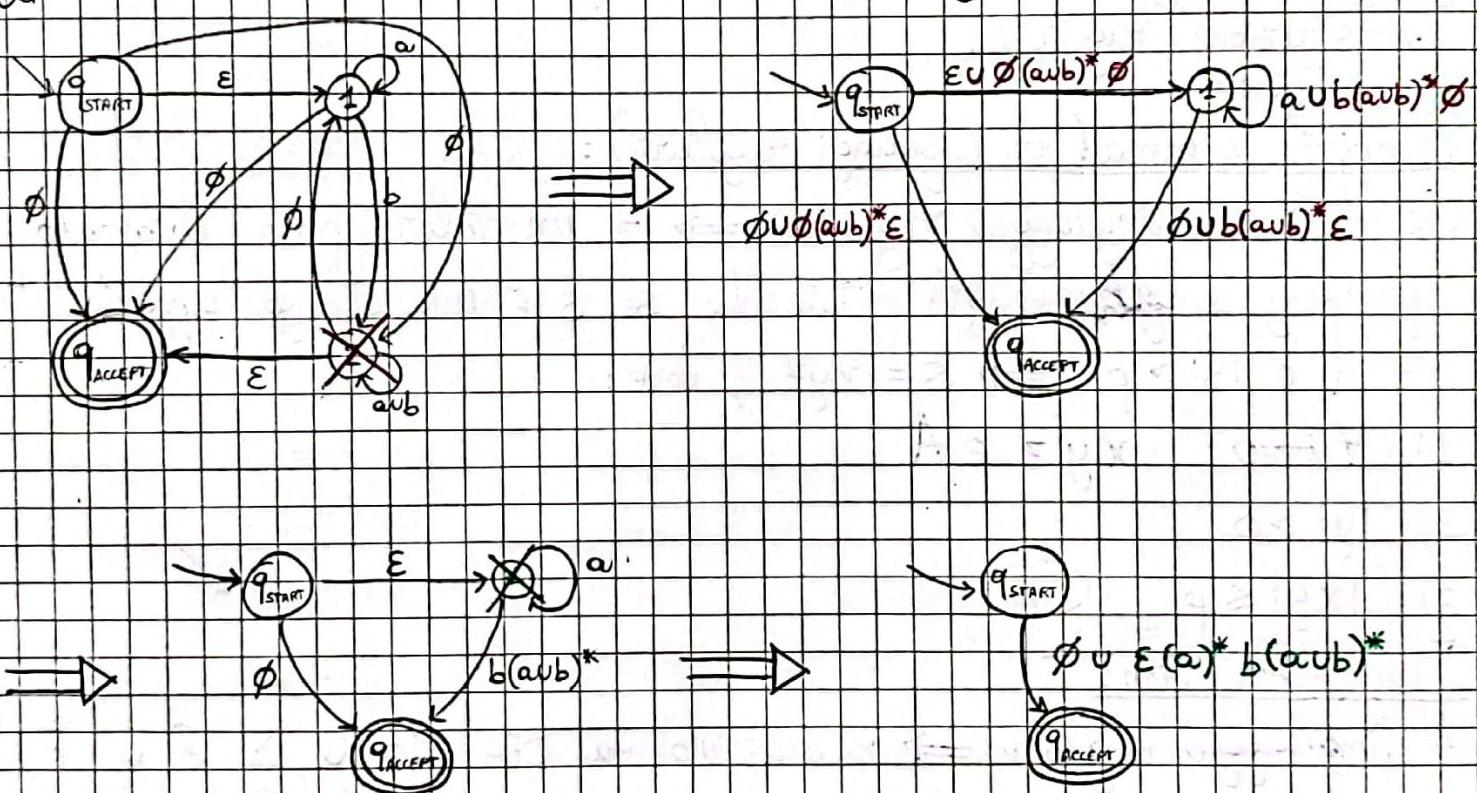
Per prima cosa costruiamo un GNFA equivalente al DFA dato dall'esercizio:



NB: Se avessimo avuto più di uno stato di accettazione nel DFA, li avremmo trattati nel seguente modo:



Per passare alla REX equivalente, dalla GNFA che ~~no~~ abbiamo appena ottenuto dobbiamo rimuovere intuitivamente uno stato intermedio per volta e aggiornare le etichette delle transizioni di conseguenza:



⇒ La REX equivalente è: $a^*b(a^*b)^*$

17/03/2020

Come facciamo a dimostrare che un linguaggio NON è regolare?

Facciamo qualche esempio:

$$B = \{0^n 1^n \mid n \geq 0\} \subseteq \{0, 1\}^*$$

$$C = \{w \in \{0, 1\}^* \mid w \text{ ha lo stesso numero di } 0 \text{ e di } 1\}$$

B e C non sembrano regolari: per essere riconosciuti, la macchina do-

trebbe saper contare il numero di 0 e di 1 all'interno della stringa w data in input. Ma, poiché l'automa non sa contare oltre il suo numero di stati e poiché la stringa può essere arbitrariamente lunga, sembra che non esista una macchina che riconosca i linguaggi B, C.

$$D = \{w \in \{0,1\}^* \mid w \text{ ha lo stesso numero di sottostringhe di } 01 \text{ e di } 10\}$$

Anche D sembra non regolare per lo stesso motivo di prima; e invece è regolare!

Perciò non ci si può affidare a ipotesi informali per stabilire se un linguaggio è regolare o meno, bensì bisogna adottare delle tecniche di dimostrazione rigorose.

Pumping Lemma (per linguaggi regolari):

Se A è un linguaggio regolare $\Rightarrow \exists$ un valore p > 0 chiamato "pumping length" tale che, se s è una stringa appartenente ad A e $|s| \geq p \Rightarrow s = xyz$, dove:

$$1) \forall i \geq 0 \quad xy^i z \in A$$

$$2) |y| > 0$$

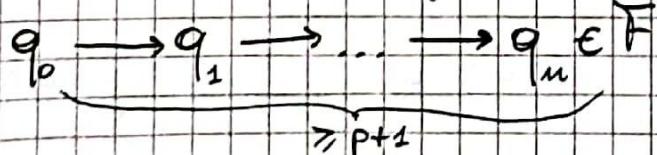
$$3) |xy| \leq p$$

Idea della dim:

A linguaggio regolare \Rightarrow riconosciuto da DFA $M = (Q, \Sigma, \delta, q_0, F)$

Supponiamo che $|Q| = p$; M è capace di contare solo fino a p.

Supponiamo che $s \in A$, $|s| \geq p$; allora una sequenza di stati che M può assumere sarà fatta così:



Poiché abbiamo supposto che M ha solo p stati diversi, per il PRINCIPIO DELLA PICCIONAIA, almeno uno stato deve essere percorso più di una volta. Denominiamo tale stato con q.

M

$$S = xyz$$

È facile intuire come anche
 xz , $xyyz$, $xyyyz$, ...
vengano accettate da M .

Dim. formale:

A linguaggio regolare \Rightarrow riconosciuto da DFA $M = (Q, \Sigma, \delta, q_0, F)$

Supponiamo che $|Q| = p$; se A , $|S| \geq p \Rightarrow S = s_1 s_2 \dots s_m$, $m \geq p$

Sequenza di stati che M può assumere:

q è lo stato che si ripete per il principio della piccionaia

$$\tau_{i+1} = \delta(\tau_i, s_{i+1})$$

È possibile impostare $S = xyz$, con:

$$x := s_1 \dots s_j$$

$$y := s_{j+1} \dots s_k$$

$$z := s_{k+1} \dots s_m$$

Dobbiamo provare che, in questo modo, le 3 condizioni del lemma sono vere:

1) $\forall i \geq 0 \quad xy^i z \in A$

• $i=0 \quad M(xy) = M(s_1 \dots s_j s_{k+1} \dots s_m)$

\hookrightarrow CON QUESTO INPUT, M ATTRAVERSA I SEGUENTI STATI:

$$\tau_0 \rightarrow \tau_1 \rightarrow \dots \rightarrow \tau_j \rightarrow \tau_{k+1} \rightarrow \dots \rightarrow \tau_m \xrightarrow{e_F}$$

Dove essere verificata la condizione per cui $\tau_{i+1} = \delta(\tau_i, s_{i+1})$

INFATTI: $\tau_{m+1} = \delta(\tau_k, s_{k+1}) \xrightarrow{\tau_j = \tau_k} \tau_{k+2} = \delta(\tau_j, s_{k+2}) \quad \checkmark$

$$\Rightarrow xz \in A$$

• $i \geq 1 \quad xy^i z \in A$ per esattamente per la stessa ragione di prima

2) $|y| > 0$ OMO PERCHÉ $j \neq k$ (quindi, per andare da τ_j a τ_k , bisogna consumare dei caratteri, che sono proprio quelli che costituiscono y)

3) $|xy| = |s_1 \dots s_k| = K \leq p$ PER COSTRUZIONE

Vedi schema all'inizio della dimostrazione

Pumping Lemma (per linguaggi non regolari):

Sia A un linguaggio per cui vale la seguente proprietà:

$\forall p > 0 \exists \bar{s} \in A, |\bar{s}| \geq p$ tale che \forall suffissione $\bar{s} = xyz$ con $|xy| \leq p$,
 $|y| > 0 \exists i \geq 0$ tale che $xy^i z \notin A$

$\Rightarrow A$ NON È REGOLARE.

Esercizio 1:

$$B = \{0^m 1^n \mid m \geq n\}$$

Dimostrare che B non è regolare.

Per assurdo, supponiamo che B sia regolare $\Rightarrow \exists$ pumping length ($p > 0$)

Consideriamo la stringa $\bar{s} = 0^p 1^p$. È evidente che $|\bar{s}| \geq p$

$$\bar{s} = \underbrace{000\dots 0}_p \underbrace{111\dots 1}_p \rightarrow \text{DOVREMO SCRIVERLA COME } xyz$$

CASO 1: LA SOTTOSTRINGA y È ESCLUSIVAMENTE COMPOSTA DA 0

$$i=2 \rightarrow \underbrace{x}_{00\dots 0} \underbrace{y}_{0\dots 0} \underbrace{y}_{0\dots 0} \underbrace{z}_{1\dots 1}$$

$|y| > 0 \Rightarrow$ il numero di 0 è maggiore di p , mentre il numero di 1 resta
uguale a $p \Rightarrow xy^2 z \notin B$

CASO 2: LA SOTTOSTRINGA y È ESCLUSIVAMENTE COMPOSTA DA 1

$$i=2 \rightarrow \underbrace{x}_{00\dots 0} \underbrace{y}_{1\dots 1} \underbrace{y}_{1\dots 1} \underbrace{z}_{1\dots 1}$$

$|y| > 0 \Rightarrow$ il numero di 1 è maggiore di p , mentre il numero di 0 resta
uguale a $p \Rightarrow xy^2 z \notin B$

CASO 3: LA SOTTOSTRINGA y È COMPOSTA SIA DA 0 CHE DA 1

$$i=2 \rightarrow \underbrace{x}_{00\dots 0} \underbrace{y}_{0\dots 1} \underbrace{y}_{0\dots 1} \underbrace{z}_{1\dots 1}$$

È evidente che $xy^2 z$ non è della forma $0^m 1^n \Rightarrow xy^2 z \notin B$

ASSURDO $\Rightarrow B$ NON È REGOLARE.

Osservazione:

In questa dimostrazione (come spesso accade) non abbiamo sfruttato la condizione $|xy| \leq p$. Se l'avessimo utilizzata, avremmo semplificato l'esercizio.

Infatti, poiché abbiamo considerato la stringa $\bar{s} = 0^p 1^p$, i cui primi p caratteri sono tutti 0, avremmo potuto dire che la sottostringa xy deve essere composta soltanto da 0; ciò avrebbe reso inutile l'analisi dei casi 2 e 3.

Esercizio 2:

$$C = \{w \in \{0,1\}^* \mid w \text{ ha lo stesso numero di } 0 \text{ e di } 1\}$$

Dimostrare che C non è regolare.

Per assurdo, supponiamo che C sia regolare $\Rightarrow \exists$ pumping length ($p > 0$)

Consideriamo la stringa $\bar{s} = 0^p 1^p$. È evidente che $|\bar{s}| \geq p$

$$\bar{s} = \underbrace{000\dots 0}_p \underbrace{111\dots 1}_p \rightarrow \text{Dovremmo scriverla come } xyz$$

Stavolta sfruttiamo la condizione $|xy| \leq p$, per cui la sottostringa xy deve essere composta unicamente da 0.

$$i=2 \rightarrow \begin{matrix} x & y & y & z \\ \curvearrowright & \curvearrowright & \curvearrowright & \\ 00\dots 0 & 0\dots 0 & 0\dots 11 \end{matrix}$$

$|y| > 0 \Rightarrow$ il numero di 0 è maggiore di p , mentre il numero di 1 resta uguale a $p \Rightarrow xy^2z \notin C$

Dato che questa suddivisione di \bar{s} era l'unica possibile, possiamo già concludere che C NON è regolare.

Osservazioni:

1) La condizione $|xy| \leq p$ è cruciale. Altrimenti avrei potuto considerare la suddivisione $x = \epsilon$, $y = 0^p 1^p$, $z = \epsilon$, che avrebbe fatto sì che $xy^i z \in C \quad \forall i \geq 0$.

2) Anche la scelta della stringa di riferimento è cruciale. Se avessimo scelto $\tilde{s} = (01)^p = 0101\dots 01$ ($|\tilde{s}| \geq p$), allora la suddivisione $x = \epsilon$, $y = 01$, $z = (01)^{p-1}$ avrebbe rispettato tutte le ipotesi del

Pumping Lemma con $xy^iz \in C \forall i \geq 0$.

Per cui, tramite \tilde{S} , non saremmo stati in grado di dimostrare che C non è un linguaggio regolare.

3) Non è strettamente necessario sfruttare il Pumping Lemma per dimostrare che un linguaggio non è regolare (in questo caso C). Qui basta combinare 3 affermazioni per giungere alla conclusione:

(I) $B = \{0^m 1^m \mid m \geq 0\}$ non è regolare

(II) I linguaggi regolari sono chiusi rispetto a \cap

(III) Le REX producono linguaggi regolari

È facile convincersi che $B = C \cap L(0^* 1^*)$.

Per assurdo, supponiamo che C sia regolare.

Poiché $0^* 1^*$ è una REX, per la (III) $L(0^* 1^*)$ è regolare.

Poiché C e $L(0^* 1^*)$ sarebbero entrambi regolari, per la (II) anche B è regolare; ma ciò contraddice la (I) \Rightarrow ASSURDO: C non può essere un linguaggio regolare.

Esercizio 3:

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

Dimostrare che F non è regolare.

Per assurdo, supponiamo che F sia regolare $\Rightarrow \exists$ pumping length ($p > 0$)

Consideriamo la stringa $\overline{s} = 0^p 1 0^p 1$. È evidente che $|s| \geq p$

$$\overline{s} = \underbrace{00\dots 0}_p \underbrace{1 00\dots 0}_p 1$$

\rightarrow DOVREMO SCRIVERLA COME xyz

Sfruttiamo la condizione $|xy| \leq p$, per cui la sottostringa xy deve essere composta unicamente da 0.

$$i=2 \rightarrow \underline{x} \underline{y} \underline{y} \underline{z}$$

$$00\dots 0 \ 0\dots 0 \ 0\dots 00\dots 01 \rightarrow 0^q 1 0^p 1$$

$|y| > 0 \Rightarrow$ il numero di 0 nella prima parte della stringa è maggiore del numero di 0 nella seconda parte della stringa ($q > p$) $\Rightarrow xy^2 z \notin F$

Dato che questa suddivisione di \bar{s} era l'unica possibile, possiamo già concludere che F NON è regolare.

Teorema:

\exists linguaggi NON regolari costruiti sopra un alfabeto unario.

Dim:

$D = \{1^m^2 \mid m > 0\}$ non è regolare. Dimostriamolo per assurdo.

Supponiamo che D sia regolare $\Rightarrow \exists$ pumping length ($p > 0$)

Consideriamo la stringa $\bar{s} = 1^p$. È evidente che $|\bar{s}| \geq p$, poiché $|\bar{s}| = p^2$.

Se scriviamo \bar{s} come xyz , abbiamo quindi che $|xyz| = p^2$.

$$i=2 \rightarrow p^2 < |xzyz| \leq p^2 + p < p^2 + p + p + 1 = (p+1)^2$$

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow$
 $|y| > 0 \quad |xy| \leq p \Rightarrow |y| < p \quad > 0$

$\Rightarrow |xy^2z|$ è compresa tra due quadrati perfetti consecutivi strettamente, per cui non può essere a sua volta un quadrato perfetto \Rightarrow

$\Rightarrow xy^2z \notin D \Rightarrow D$ NON è regolare. ■

Esercizio 4:

$E = \{0^i 1^j \mid i > j\}$

Dimostrare che E non è regolare.

Per assurdo, supponiamo che E sia regolare $\Rightarrow \exists$ pumping length (pro)

Consideriamo la stringa $\bar{s} = 0^{p+1} 1^p$. È evidente che $|\bar{s}| \geq p$

$$\bar{s} = \underbrace{000\dots 0}_{p+1} \underbrace{11\dots 1}_p \quad \rightarrow \text{DOVREMO SCRIVERLA COSE } xyz$$

$$|xy| \leq p \Rightarrow y \in 0^*$$

$$i=0 \rightarrow x \ z$$

$|y| > 0 \Rightarrow$ la stringa xz ha meno 0 di quanti ne ha \bar{s} ma, allo stesso tempo, conserva tutti gli 1 \Rightarrow non è vero che xz contiene più 0 che 1 $\Rightarrow xz \notin E$

Dato che questa suddivisione di \bar{s} era l'unica possibile, possiamo già concludere che E NON è regolare.

NB: A volte è molto complicato dimostrare che un dato linguaggio non è regolare mediante il Pumping Lemma.

Esempio:

$A = \{w \in \{0,1\}^* \mid w \text{ ha un numero differente di } 0 \text{ e di } 1\}$

è un linguaggio non regolare. Tuttavia, se prendiamo una qualunque stringa $s \in A$, $|s| \geq p$, difficilmente riusciremo a concludere che, per ogni suffissione xyz , ponendo y , si ottenga una stringa che non appartenga ad A , avendo che abbia lo stesso numero di 0 e di 1. Perciò, occorre una tecnica diversa per dimostrare la non regolarità di A .

20/03/2020

Definizione:

Il complemento di un linguaggio $L \subseteq \Sigma^*$ è:

$$L^c := \Sigma^* \setminus L = \{w \in \Sigma^* \mid w \notin L\}$$

Teorema:

Se L è un linguaggio regolare $\Rightarrow L^c$ è un linguaggio regolare.

Dim:

L regolare $\Rightarrow \exists$ DFA $M = (Q, \Sigma, \delta, q_0, F)$ t.c. $L(M) = L$

Sia $M' = (Q, \Sigma, \delta, q_0, Q \setminus F) \Rightarrow L(M') = L^c \Rightarrow L^c$ REGOLARE \blacksquare

Consideriamo l'ultimo esempio che abbiamo visto:

$A = \{w \in \{0,1\}^* \mid w \text{ ha un numero differente di } 0 \text{ e di } 1\}$

$A^c = \{w \in \{0,1\}^* \mid w \text{ ha lo stesso numero di } 0 \text{ e di } 1\}$

Già sappiamo che A^c NON è regolare $\Rightarrow A$ NON è regolare.

Altro esempio:

• A, B regolari $\Rightarrow A \cup B$ regolare, A^c regolare

$$\Rightarrow (A^c \cup B^c)^c = A \cap B \text{ è regolare}$$

→ modo più facile per dimostrare la chiusura dei linguaggi regolari rispetto all'intersezione

Definizione:

Sia $\sigma \in \Sigma^*$, $\sigma = \sigma_1 \sigma_2 \dots \sigma_n$ una stringa.

La stringa inversa è: $\sigma^R := \sigma_n \sigma_{n-1} \dots \sigma_2 \sigma_1$

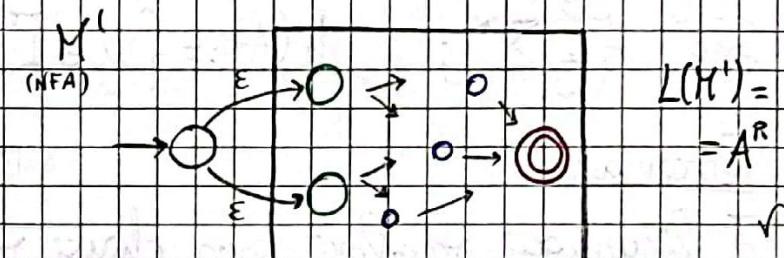
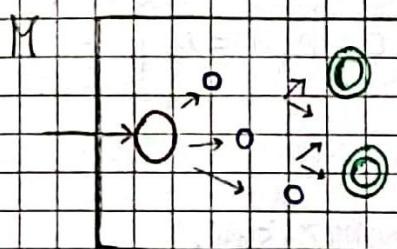
Sia $A \subseteq \Sigma^*$ un linguaggio. $A^R := \{w \in \Sigma^* \mid w^R \in A\}$.

Teorema:

Se A è un linguaggio regolare $\Rightarrow A^R$ è un linguaggio regolare.

Dim:

A regolare $\Rightarrow \exists$ DFA M t.c. $L(M) = A$



OPPURE:

Possiamo dimostrare che le espressioni regolari sono chiuse rispetto al ricvesciamento.

PASSO BASE ($|R| = 1$):

$$E^R = E \quad \emptyset^R = \emptyset \quad \{a\}^R = \{a\} \quad \checkmark$$

PASSO INDUTTIVO ($|R| \leq n$):

- $R = R_1 \cup R_2$ regolare $\Rightarrow R^R = R_1^R \cup R_2^R$ intuitivamente regolare
- $R = R_1 \circ R_2$ regolare $\Rightarrow R^R = R_2^R \circ R_1^R$ intuitivamente regolare
- $R = (R_1)^*$ regolare $\Rightarrow R^R = (R_1^R)^*$ intuitivamente regolare

Sfruttando la corrispondenza tra linguaggi regolari ed espressioni regolari, possiamo concludere che: A regolare $\Rightarrow A^R$ regolare ■

Definizione:

Un omomorfismo su un alfabeto Σ è una funzione $h: \Sigma \rightarrow \Gamma^*$, dove Γ è un alfabeto.

Esempio:

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{a, \dots, z\}$$

$$h: \Sigma \rightarrow \Gamma^*$$

0 \rightarrow "zero"
1 \rightarrow "one"

Possiamo estendere il concetto di omomorfismo su un alfabeto a omomorfismo su un insieme di stringhe.

Sia $w \in \Sigma^*$; $h(w) := h(w_1) h(w_2) \dots h(w_m) \in \Gamma^*$

Possiamo estendere ulteriormente tale concetto a omomorfismo su un linguaggio.

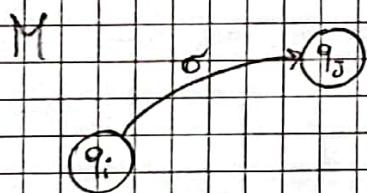
Sia $A \subseteq \Sigma^*$; $h(A) := \{u \in \Gamma^* \mid \exists w \in \Sigma^* \text{ t.c. } h(w) = u\}$

Teorema:

I linguaggi regolari sono chiusi rispetto agli omomorfismi.

Dim:

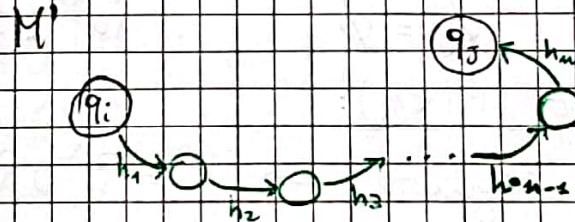
A regolare $\Rightarrow \exists$ DFA M t.c. $L(M) = A$



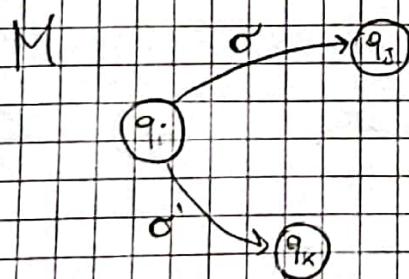
$$\sigma \in \Sigma$$

$$h: \Sigma \rightarrow \Gamma^*$$

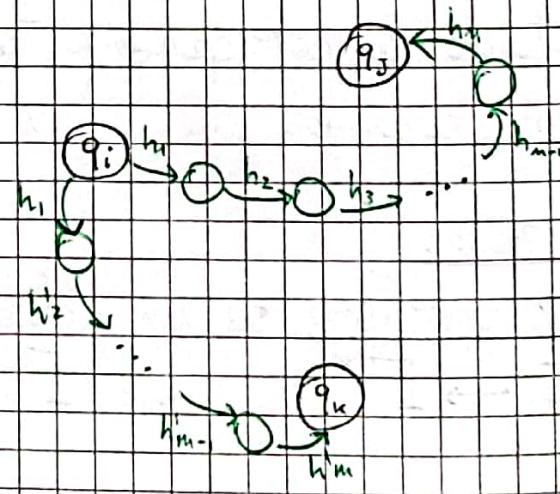
$$h(\sigma) = h_1 h_2 \dots h_m$$



Notiamo che M' può essere una NFA. Per esempio:



M'
NFA



Può succedere che:

$$h(\sigma) = h_1 h_2 \dots h_m \Rightarrow M' \text{ è un NFA}$$

$$h(\sigma') = h'_1 h'_2 \dots h'_{m'}$$

A questo punto resta solo da definire formalmente i parametri della macchina M per completare la dimostrazione (operazione lasciata per esercizio).

OPPURE:

Possiamo dimostrare che le espressioni regolari sono chiuse rispetto all'omomorfismo.

Ma prima osserviamo che tale operazione si può estendere anche alle espressioni regolari.

Siano Σ, Γ due alfabeti.

$$\Sigma^* := \Sigma \cup \{ () * \circ \cup \}$$

$$\Gamma^* := \Gamma \cup \{ () * \circ \cup \}$$

L'omomorfismo sulle REX è definito come:

$$\begin{array}{ccc} h: \Sigma^* & \longrightarrow & \Gamma^* \\ \sigma & \mapsto & h(\sigma) \quad \text{se } \sigma \in \Sigma \\ \sigma & \mapsto & \sigma \quad \text{se } \sigma \notin \Sigma \end{array}$$

Ora possiamo osservare che:

$$h(A) = h(L(R)) = L(h(R)) \Rightarrow h(A) \text{ è regolare}$$

Resta da dimostrare che $h(L(R)) = L(h(R))$. Facciamolo per induzione:

PASSO BASE ($|R| = 1$):

$$h(\epsilon) = \epsilon \qquad h(\emptyset) = \emptyset$$

$$\sigma \in \Sigma \mid h(\sigma) = u \in \Gamma^* \Rightarrow h(L(\sigma)) = h(\{\sigma\}) = \{u\} = L(u) = L(h(\sigma))$$

PASSO INDUTTIVO ($|R| \leq n$):

$$\bullet R = R_1 \cup R_2 \xrightarrow{\text{regolare}} h(R) = h(R_1) \cup h(R_2)$$

$$\bullet R = R_1 \circ R_2 \xrightarrow{\text{regolare}} h(R) = h(R_1) \circ h(R_2)$$

$$\bullet R = (R_1)^* \xrightarrow{\text{regolare}} h(R) = (h(R_1))^*$$

E quindi induttivamente ci si può ricontroare al caso base.

21/03/2020

Un sistema formale per caratterizzare i linguaggi non regolari è quello delle GRAMMATICHE LIBERE DAL CONTESTO (CFG).

Definizione:

Una CFG è una tupla (V, Σ, R, S) , dove:

V = VARIABILI: costituiscono un insieme di simboli che appaiono nelle regole della grammatica e che possono essere sostituiti da altre stringhe.

Σ = TERMINALI: costituiscono un insieme di simboli che non possono essere sostituiti.

R = REGOLE DI PRODUZIONE: sono le regole di sostituzione definite nella grammatica.

S = VARIABILE DI PARTENZA: appartiene a V .

Esempio:

$$G = (V, \Sigma, R, S)$$

$$V = \{A, B\}$$

$$\Sigma = \{0, 1\}$$

$$S = A$$

$$R = \{A \rightarrow 0A1, \quad A \rightarrow B, \quad B \rightarrow \epsilon\}$$

$$A \rightarrow B, \quad B \rightarrow \epsilon$$

$$A \rightarrow 0A1$$

$$= A \rightarrow 0A1B$$

C'è una scelta sul come trasformare $A \Rightarrow$ è una grammatica non deterministica

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\epsilon 11 = 0011$$

↳ Passo di derivazione

NB: Una CFG genera direttamente un linguaggio non regolare.

Nel nostro esempio, viene generato $B = \{0^n 1^m \mid n \geq 0\}$.

In generale, si dice che una stringa $\sigma \in (V \cup \Sigma)^*$ genera una stringa $\sigma' \in (V \cup \Sigma)^*$ ($\sigma \Rightarrow^* \sigma'$) se \exists stringhe u_1, u_2, \dots, u_k tali che:

$$\cdot u_1 = \sigma$$

$$\cdot u_k = \sigma'$$

$\cdot u_i \Rightarrow u_{i+1}$, cioè: applicando una regola di produzione a u_i , si ottiene u_{i+1} .

A questo punto siamo in grado di stabilire formalmente come è fatto un linguaggio generato da una CFG:

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}, \text{ dove } G \text{ è una CFG}$$

Definizione:

Qualunque linguaggio generato da una CFG è un LINGUAGGIO LIBERO

DAL CONTESTO (CFL).

Corollario:

\exists CFL che NON sono regolari.

Teorema:

Ogni linguaggio regolare è un CFL.

Dim:

L regolare $\Rightarrow \exists$ DFA $M = (Q, \Sigma, \delta, q_0, F)$ t.c. $L(M) = L$.

Sia $G = (V, \Sigma, R, S)$ una CFG

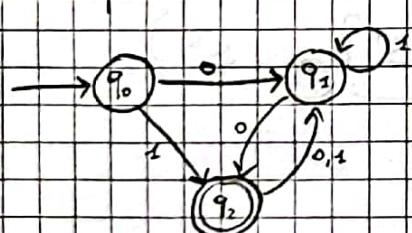
$V = \{R_i \mid q_i \in Q\}$ \rightarrow UN SIMBOLO DI V PER OGNI STATO DI M

$$\begin{array}{l} q_i \xrightarrow{a} q_j \\ \delta(q_i, a) = q_j \end{array} \quad R = \{R_i \rightarrow aR_j \mid \delta(q_i, a) = q_j\} \cup \{R_i \rightarrow \epsilon \mid q_i \in F\}$$

$$S = R_0$$

In questo modo possiamo concludere informalmente che $L = L(M) = L(G)$ ■

Esempio:



$$G = (V, \Sigma, R, S)$$

$$V = \{R_0, R_1, R_2\}$$

$$S = R_0$$

$$\begin{aligned} R = & \{R_0 \rightarrow 0R_1 \mid 1R_2, \quad R_1 \rightarrow 0R_2 \mid 1R_1, \\ & R_2 \rightarrow 0R_1 \mid 1R_1 \mid \epsilon\} \end{aligned}$$

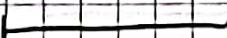
ESEMPIO DI STRINGA ACCETTATA DALL'AUTOMA: 0010

$$R_0 \Rightarrow 0R_1 \Rightarrow 00R_2 \Rightarrow 001R_1 \Rightarrow 0010R_1 \Rightarrow 0010 \in \Sigma^*$$

ESEMPIO DI STRINGA NON ACCETTATA DALL'AUTOMA: 0011

$$R_0 \Rightarrow 0R_1 \Rightarrow 00R_2 \Rightarrow 001R_1 \Rightarrow 0011R_1 \Rightarrow ???$$

Non ~~può~~ ^{si può} produrre la stringa 0011 $\Rightarrow 0011 \notin L(G)$



Facciamo ora un altro esempio di CFG:

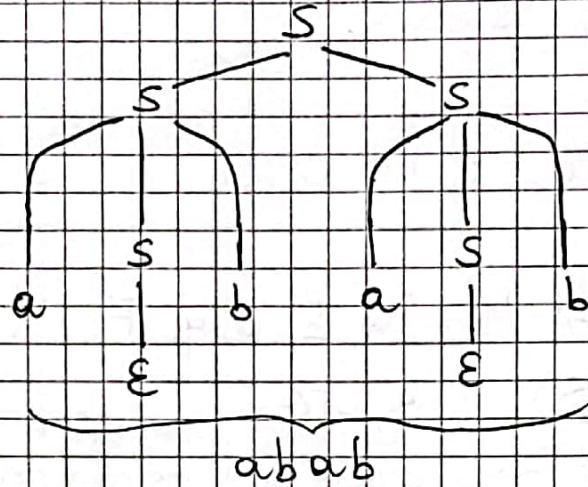
$$G_2 = (S, \{a, b\}, R, s)$$

$$L(G_2) = \{\epsilon, ab, aabb, abab, \dots\}$$

$$R = \{S \rightarrow asb \mid ss \mid \epsilon\}$$

E' possibile vedere la sequenza di tutti i passi di derivazione che portano da s a una stringa $w \in \Sigma^*$ come una struttura ad albero. Per esempio:

TALE STRUTTURA
SI CHIAMA "ALBERO SINTATTICO"
(parse tree)



Teorema:

Se A, B sono CFL $\Rightarrow A \cup B$ è un CFL

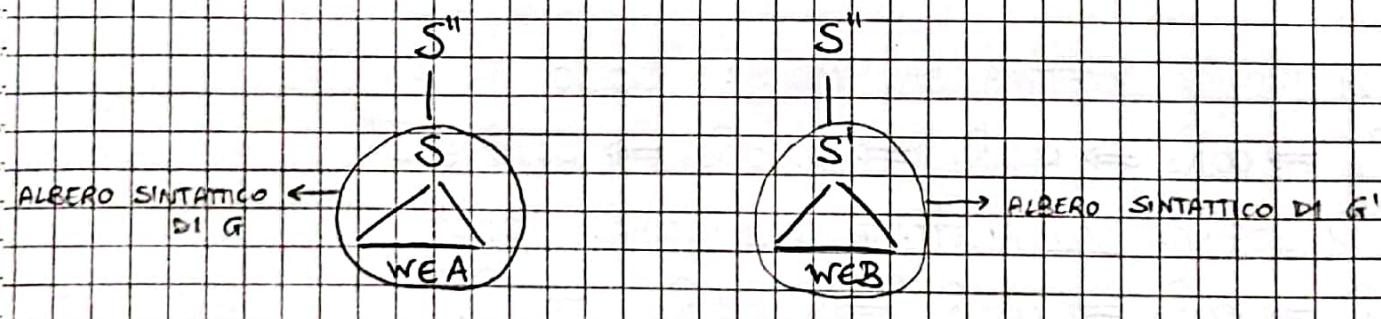
Dim:

A è un CFL $\Rightarrow \exists G = (V, \Sigma, R, S)$ t.c. $L(G) = A$

B è un CFL $\Rightarrow \exists G' = (V', \Sigma', R', S')$ t.c. $L(G') = B$

$$G'' = (\underbrace{V \cup V' \cup \{S''\}}_{V''}, \underbrace{\Sigma \cup \Sigma'}_{\Sigma''}, \underbrace{R \cup R' \cup \{S'' \rightarrow S|S'\}}_{R''}, S'')$$

Consideriamo i due possibili alberi sintattici di G'' :



Da qui è facile intuire che $L(G'') = A \cup B$

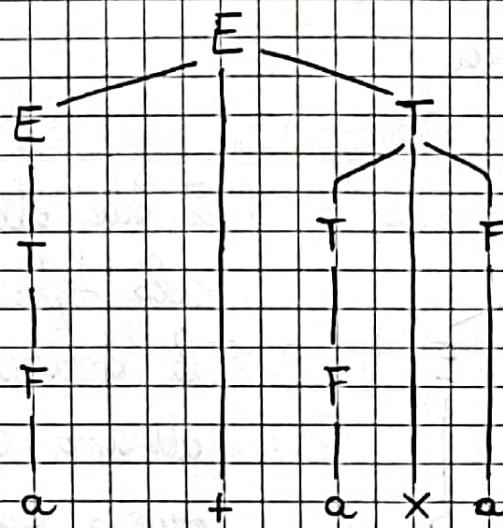
Verifichiamo un altro esempio di CFG:

$$(G_3 = (\{E, T, F\}, \{a, +, \times, (,), \}, R, E})$$

$$R = \{E \rightarrow E + T \mid T, \quad T \rightarrow T \times F \mid F, \quad F \rightarrow (E) \mid a\}$$

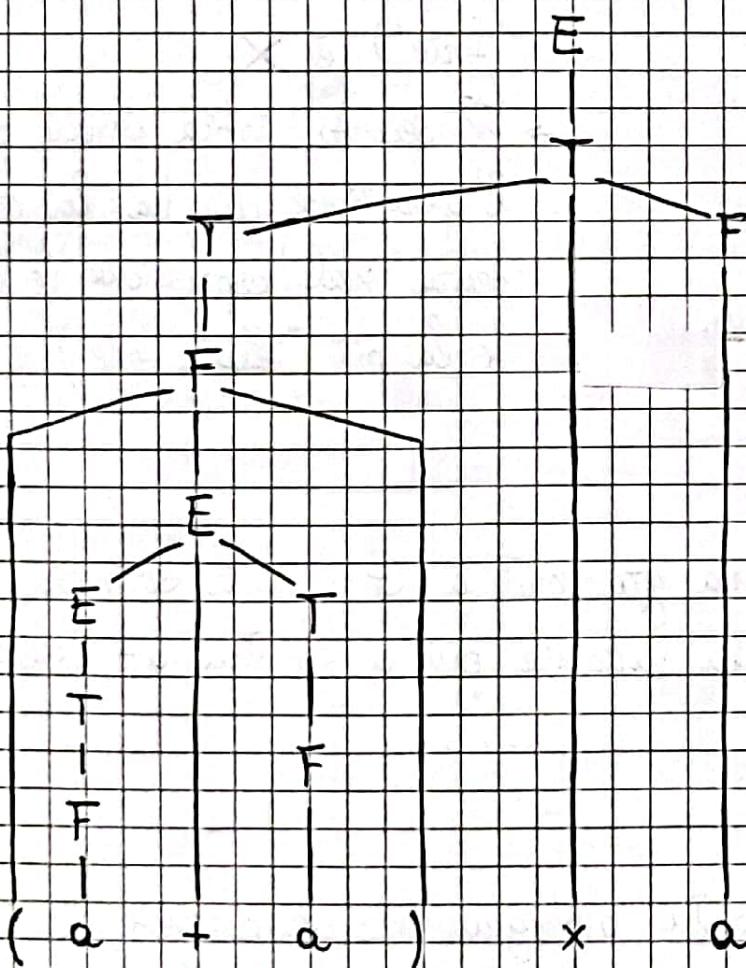
Per esempio si ha che $E \Rightarrow^* a+a \alpha$

Costruiamo l'albero sintattico corrispondente a "a+aα":



Si ha anche che $E \Rightarrow^* (a+a)\alpha$

Costruiamo l'albero sintattico corrispondente a "(a+a)α":

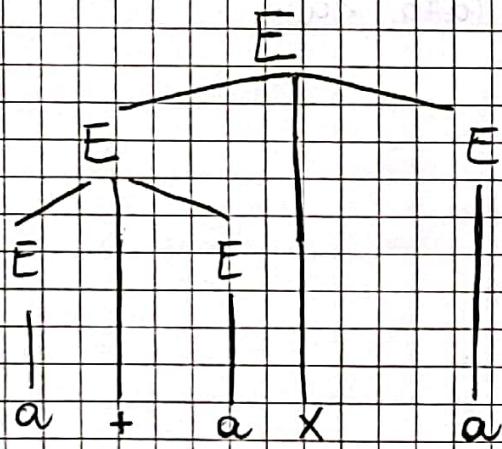
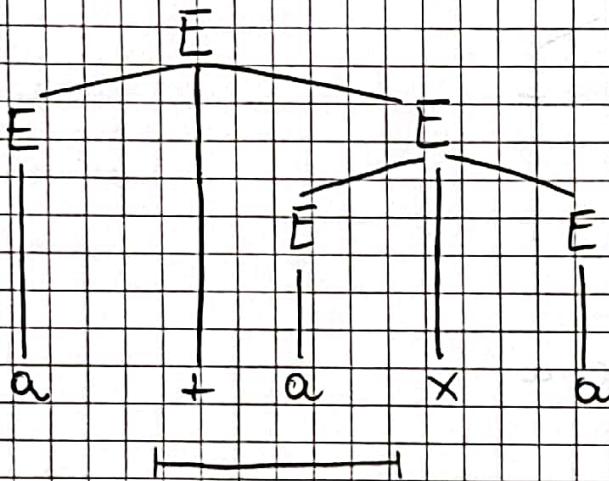


NB: Non sempre una stringa generata da un CFG ha un unico albero sintattico corrispondente. Vediamo un esempio di CFG in cui una stringa si può rappresentare con due alberi sintattici.

$G_n = (\{E\}, \{a, +, \times, (,), \}, R, E)$

$R = \{E \rightarrow E \times E \mid E + E \mid (E) \mid a\}$

Per esempio, $E \Rightarrow^* a + a \times a$



I due alberi sintattici non solo differiscono per l'ordine delle operazioni con il quale li abbiamo costruiti, ma hanno proprio significati diversi:

→ Il primo indica che l'operazione che ha la precedenza nell'espressione (e quindi la più "forte") è \times

→ Il secondo indica invece che l'operazione che ha la precedenza nell'espressione (e quindi la più ~~"debole"~~ "forte") è $+$

Definizione:

Una derivazione $u \Rightarrow^* v$ in una grammatica G è una derivazione a sinistra se ad ogni passo è la variabile più a sinistra ad essere sostituita.

Definizione:

Una stringa $w \in L(G)$ è derivata ambigamente se possiede due o più derivazioni a sinistra (\Rightarrow due o più alberi sintattici).

Definizione:

Una CFG G è ambigua se $\exists w \in L(G)$ t.c. w è derivata ambigamente.

Definizione:

Un CFL è inerentemente ambiguo se può essere generato soltanto da CFG ambigue.

Definizione (Forma Normale di Chomsky - CNF):

Una CFG $G = (V, \Sigma, R, S)$ è detta in Forma Normale di Chomsky se ogni regola è della forma:

$$A \rightarrow BC \quad A \in V; B, C \in V \setminus \{S\}$$

$$A \rightarrow a \quad A \in V; a \in \Sigma$$

$$S \rightarrow \epsilon$$

Teorema:

Ogni CFL è generato da una CFG nella Forma Normale di Chomsky.

Dim:

Sia L un CFL $\Rightarrow \exists$ CFG G t.c. $L(G) = L$

$$G = (V, \Sigma, R, S)$$

Trasformiamo G finché non diventa nella Forma Normale di Chomsky.

$$1) S_0 \rightarrow S \quad S_0 \text{ è la nuova variabile di partenza}$$

$$2) \text{ Se in } G \text{ abbiamo che } A \rightarrow \epsilon, A \neq S_0 \text{ e, per esempio,}$$

$B \rightarrow uAwAz$, sistemiamo le regole in modo tale che

$$B \rightarrow uwA_z | uwz | uwz$$

Caso particolare: $B \rightarrow A \in R$. In tal caso, la regola diventerebbe $B \rightarrow \epsilon$, ma solo se $B \rightarrow \epsilon$ non era stata già rimossa.

$$3) \text{ Se in } G \text{ abbiamo } A \rightarrow B \text{ allora, per ogni } B \rightarrow u, \text{ introduciamo la regola } A \rightarrow u, \text{ a meno che } A \rightarrow u \text{ è stata già rimossa.}$$

$$4) \text{ Se in } G \text{ abbiamo } A \rightarrow u_1 u_2 \dots u_k, k \geq 2, u_i \in V \vee u_i \in \Sigma \text{ allora possiamo sistemare la regola in modo tale che}$$

$$A \rightarrow u_1' A_1 \quad A_1 \rightarrow u_2' A_2 \quad A_{k-1} \rightarrow u_{k-1}' u_k' \quad , \text{ con}$$

$$u_i' = \begin{cases} u_i & \text{se } u_i \in V \\ U_i & \text{se } u_i \in \Sigma \end{cases}; \text{ inoltre, introduciamo la regola } U_i \rightarrow u_i \quad \forall u_i \in \Sigma$$

Esercizio:

Trasformare la seguente grammatica in forma Normale di Chomsky:

$$G = (V, \Sigma, R, S)$$

$$V = \{A, B, S\}$$

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

$$R = \{S \rightarrow ASA \mid aB, \quad A \rightarrow B \mid S, \quad B \rightarrow b \mid \epsilon\}$$

1) Aggiungiamo la regola $S_0 \rightarrow S$.

2) Rimuoviamo la regola $B \rightarrow \epsilon$ e introduciamo $S \rightarrow a$, $A \rightarrow \epsilon$ (dove abbiamo sostituito B con ϵ nelle regole dove B compare a destra).

3) Rimuoviamo la regola $A \rightarrow \epsilon$ e introduciamo $S \rightarrow SA$, $S \rightarrow AS$, $S \rightarrow S$.

4) La regola $S \rightarrow S$ è del tutto inutile: può essere rimossa tranquillamente.

5) In $S_0 \rightarrow S$, sostituendo S con tutte le regole in cui compare S stesso, otteniamo: $S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$.

6) Facciamo la stessa cosa con la regola $A \rightarrow B$, che verrà trasformata in $A \rightarrow b$.

7) Facciamo la stessa cosa anche con la regola $A \rightarrow S$, che verrà trasformata in $A \rightarrow b \mid ASA \mid aB \mid a \mid SA \mid AS$.

8) Per rimuovere le regole $S_0 \rightarrow ASA$, $S \rightarrow ASA$, $A \rightarrow ASA$, introduciamo $A_1 \rightarrow SA$ e sostituiamo le regole da eliminare rispettivamente con $S_0 \rightarrow AA_1$, $S \rightarrow A_1 A_1$, $A \rightarrow AA_1$.

9) Per rimuovere le regole $S_0 \rightarrow aB$, $S \rightarrow aB$, $A \rightarrow aB$, introduciamo $U_1 \rightarrow a$ e sostituiamo le regole da eliminare rispettivamente con $S_0 \rightarrow U_1 B$, $S \rightarrow U_1 B$, $A \rightarrow U_1 B$.

In conclusione, le regole della nuova grammatica nella forma Normale di Chomsky sono:

$$S_0 \rightarrow AA_1 \mid U_1 B \mid a \mid SA \mid AS$$

$$B \rightarrow b$$

$$S \rightarrow AA_1 \mid U_1 B \mid a \mid SA \mid AS$$

$$A_1 \rightarrow SA$$

$$A \rightarrow b \mid AA_1 \mid U_1 B \mid a \mid SA \mid AS$$

$$U_1 \rightarrow a$$

Anche i CFL sono caratterizzati da un modello di calcolo (o macchina): il PUSH DOWN AUTOMATA (PDA), che è caratterizzato da una quantità infinita di memoria organizzata in STACK.

Definizione (automa push down):

È una tupla costituita da $(Q, \Sigma, \Gamma, \delta, q_0, F)$, dove:

Q = insieme degli stati intorno della macchina

Σ = alfabeto di input, ovvero l'alfabeto letto dalla macchina

Γ = alfabeto dello stack, ovvero l'alfabeto dei simboli che possono essere memorizzati sullo stack

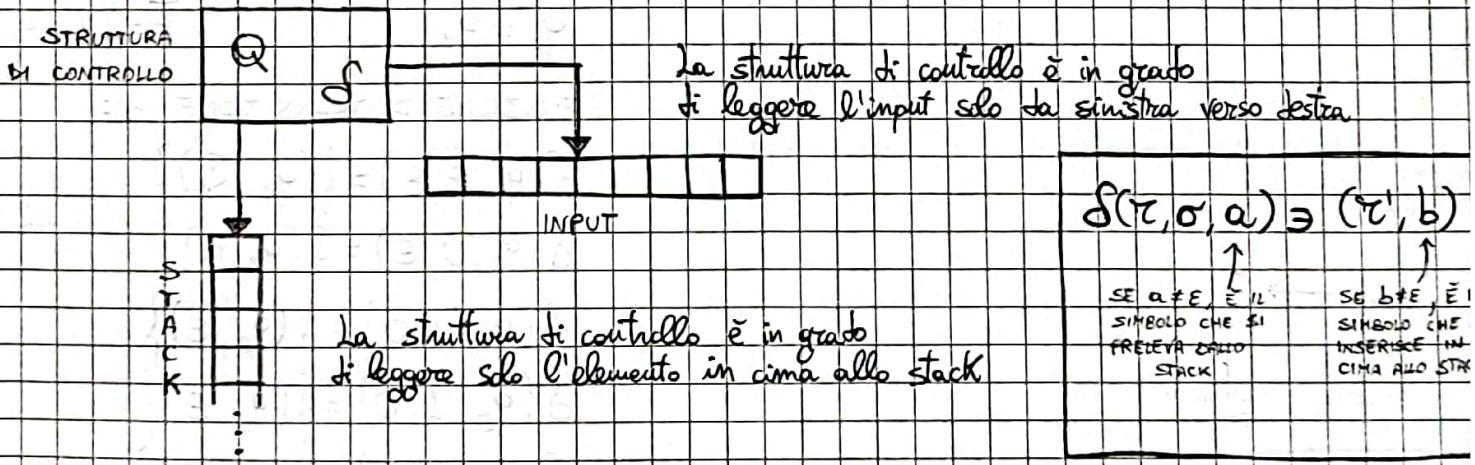
$$f: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow 2^{Q \times \Gamma_\varepsilon}, \quad \Sigma_\varepsilon := \sum \cup \{\varepsilon\}, \quad \Gamma_\varepsilon := \Gamma \cup \{\varepsilon\}$$

\Rightarrow si tratta di una macchina non deterministica

$q_0 \in Q$ = stato di partenza

$F \subseteq Q$ = insieme degli stati di accettazione

SCHEMA INFORMATIVO DI UN AUTOMA PUSH DOWN:



Definizione (computazione accettante di un PDA):

Sia M un PDA e sia $w \in \Sigma^*$ una stringa data in input a M .

Esiste una computazione accettante se:

- 1) \exists sequenza di caratteri $w = w_1 w_2 \dots w_m$ t.c. $w_i \in \Sigma_\epsilon$
 - 2) \exists sequenza di stati interni r_0, r_1, \dots, r_m t.c. $r_i \in Q$
 - 3) \exists sequenza di stringhe s_0, s_1, \dots, s_m t.c. $s_i \in \Gamma^*$;
ciascuna di queste stringhe rappresenta la configurazione dello stack in un

certo istante.

Le caratteristiche di una computazione accettante sono:

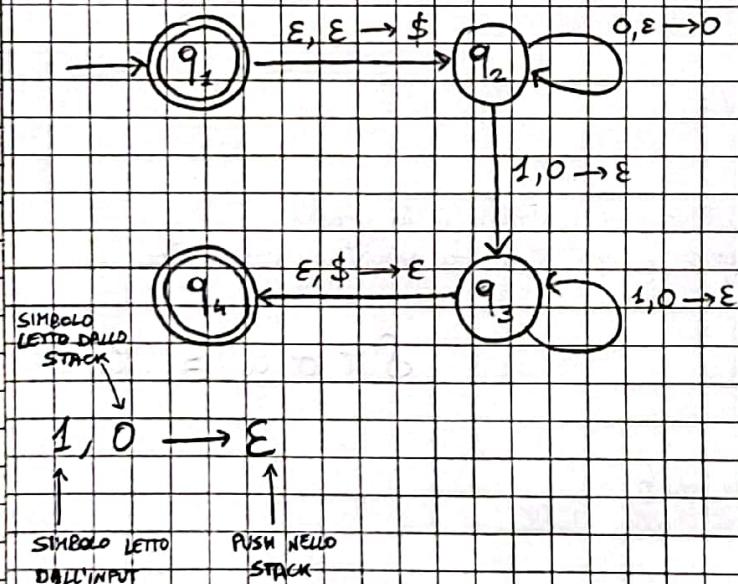
- 1) $r_0 = q_0$
- 2) $s_0 = \epsilon \rightarrow$ lo stack è inizialmente vuoto
- 3) $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, dove $s_i = at$, $s_{i+1} = bt$,
 $a, b \in \Gamma_\epsilon$, $t \in \Gamma^*$
- 4) $r_m \in F$

24/03/2020

PROBLEMA TECNICO: come fa un PDA a capire se lo stack è vuoto o meno quando va a leggerlo per prelevare un simbolo?

SOLUZIONE: introduzione di un simbolo speciale, \$, considerato per definizione l'ultimo simbolo dello stack oltre il quale non c'è niente.

Esempio di costruzione di un PDA a partire da un CFL ($B = \{0^m 1^n \mid m \geq 0\}$):



$$\begin{aligned} Q &= \{q_1, q_2, q_3, q_4\} \\ \Sigma &= \{0, 1\} \\ \Gamma &= \{0, \$\} \\ q_0 &= q_1 \\ F &= \{q_1, q_4\} \end{aligned}$$

FUNZIONE DI TRANSIZIONE δ :

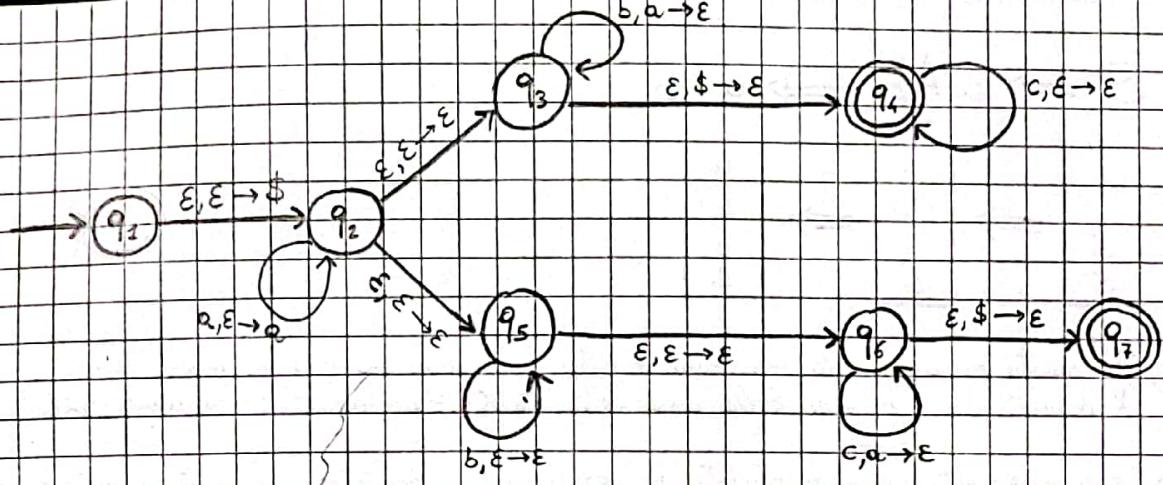
$$\begin{aligned} \delta(q_1, \epsilon, \epsilon) &= \{(q_2, \$)\} \\ \delta(q_2, 0, \epsilon) &= \{(q_2, 0)\} \\ \delta(q_2, 1, 0) &= \{(q_3, \epsilon)\} \\ \delta(q_3, 1, 0) &= \{(q_3, \epsilon)\} \\ \delta(q_3, \epsilon, \$) &= \{(q_4, \epsilon)\} \end{aligned}$$

Esercizio 1:

Costruire un PDA che accetti il seguente linguaggio:

$$\{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i=j \vee i=k)\}$$

Esistono diverse soluzioni a questo esercizio. Vediamo una.

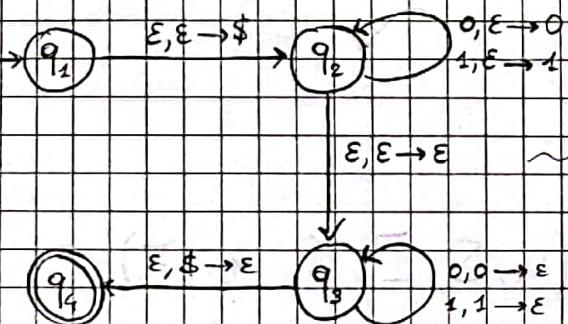


La biforcazione è fondamentale per poter confrontare in parallelo il numero di b e il numero di c con il numero di a .

Esercizio 2:

Costruire un PDA che accetti il seguente linguaggio:

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



Qui la macchina sfrutta il determinismo.
Non saendo a priori quando termina w e inizia w^R , prova in parallelo tutte le possibilità di separazione della stringa.

Teorema:

L è un CFL \Leftrightarrow un certo PDA lo riconosce.

Dim:

$$\Rightarrow L \text{ è un CFL} \Rightarrow \exists \text{ PDA } P \text{ t.c. } L(P) = L$$

Sappiamo che se L è un CFL $\Rightarrow \exists \text{ CFG } G = (V, \Sigma, R, S) \text{ t.c. } L(G) = L$

L'idea è costruire un PDA P t.c. $\forall w \in \Sigma^* \quad P(w)$ "simula" $S \xrightarrow{*} w$

In altre parole, vogliamo costruire un PDA in modo tale che, a ogni iterazione, i caratteri salvati nello stack corrispondano esattamente a quelli della stringa w tale che $S \xrightarrow{*} w$ in quella stessa iterazione.

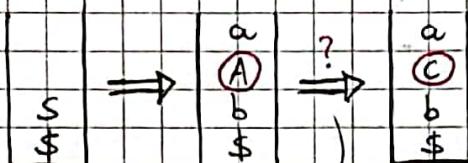
In questo modo, il PDA deve accettare l'input w se e solo se corrisponde ai caratteri salvati nello stack.

C'è solo un problema. Vediamolo con un esempio.

PASSI DI
DERIVAZIONE:

$$S \Rightarrow aAb \Rightarrow acb$$

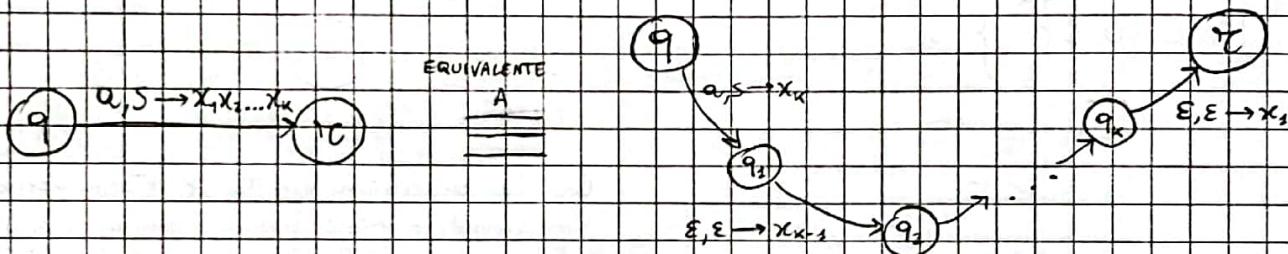
STACK:



In questo passaggio, la macchina dovrebbe fare una pop e una push di elementi che non sono sulla cima dello stack: sono operazioni inammissibili!

SOLUZIONE: visto che sono previste le derivazioni a sinistra, gli unici caratteri che danno problemi sono i terminali, i quali possono essere subito confrontati con i simboli corrispondenti dell'input e, quindi, estratti dallo stack. In caso di corrispondenza, la macchina può immediatamente rigettare l'input.

NB: Utilizzeremo una semplificazione nella notazione:



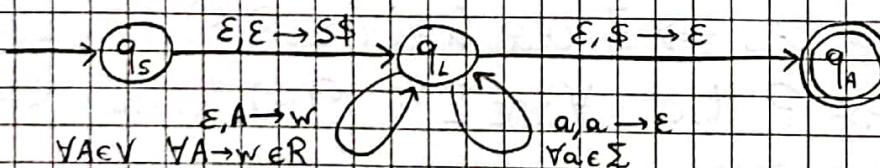
$$G = \{V, \Sigma, R, S\} \rightarrow P = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

$Q = \{q_s, q_L, q_A\} \cup \{\text{eventuali stati intermedi per il push della stringa}\}$

Σ corrisponde all'insieme dei terminali di G

$$\Gamma = \{\$\}, V \cup \Sigma ; \quad q_0 = q_s ; \quad F = \{q_A\}$$

Rappresentiamo graficamente l'insieme delle transizioni δ :



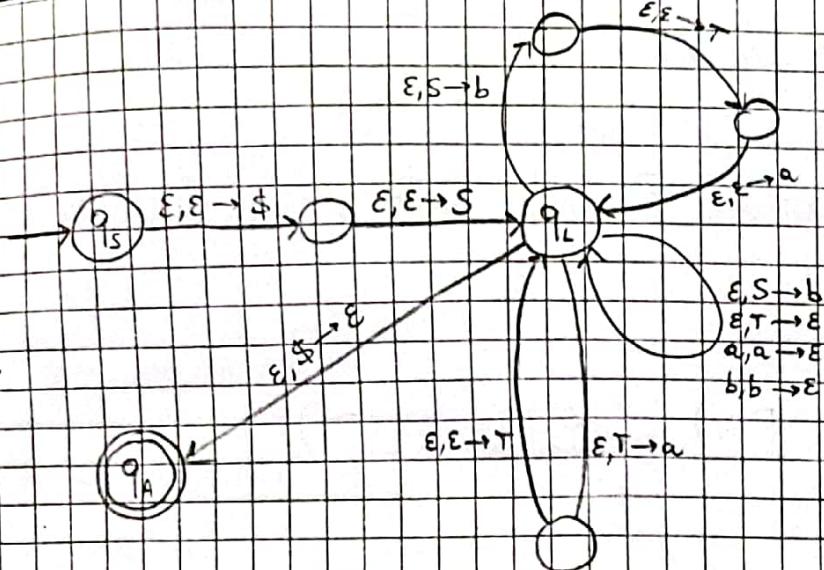
E Esempio:

$$G = (V, \Sigma, R, S)$$

$$R = \{S \rightarrow aTb \mid b, T \rightarrow Ta \mid \epsilon\}$$

$$V = \{S, T\} ; \quad \Sigma = \{a, b\} ; \quad S = S$$

Nella pagina seguente è riportato il PDA corrispondente.

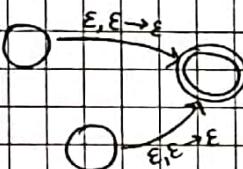


$\Leftrightarrow P$ è un PDA $\Rightarrow \exists L(P)$, che è un CFL (e quindi \exists CFG G t.c. $L(G) = L(P)$)
 $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$

È necessario applicare delle restrizioni a P , che però non sono limitanti.

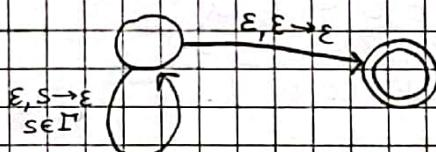
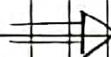
① $|F| = 1$, $F = \{q_A\}$

Se ci sono più stati di accettazione, possiamo ridurli a uno nel seguente modo:



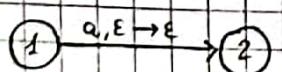
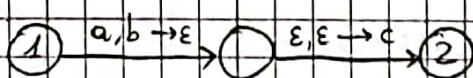
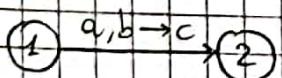
② Nel momento in cui P accetta una stringa, lo stack deve essere vuoto.

Se non lo è, possiamo effettuare la seguente modifica:



③ Ogni transizione deve rappresentare o una PUSH o una POP.

Vediamo come possono essere modificate le transizioni non consentite:



A questo punto vogliamo costruire una grammatica $G = (V, \Sigma, R, S)$ che simuli le iterazioni di P : $S \Rightarrow^* w \Leftrightarrow P(w)$ accetta

L'insieme dei terminali corrisponde all'alfabeto Σ di P :

$$V = \{A_{pq} \mid p, q \in Q\}$$

$$S = A_{q, q_A}$$

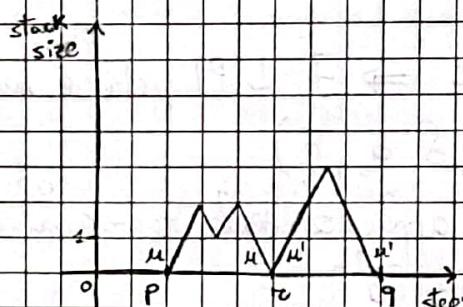
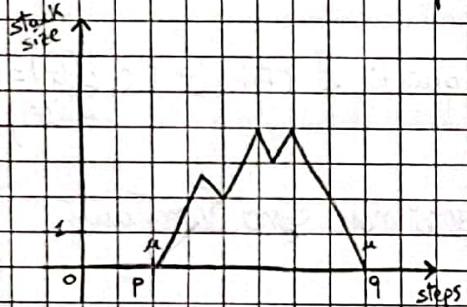
A_{pq} genera tutte le stringhe che portano P dallo stato p con stack vuoto allo stato q con stack vuoto.

In particolare:

$$A_{q, q_A} \Rightarrow^* w \in \Sigma^*$$

significa che $P(w)$ accetta ($w \in L(P)$)

Ci sono due scenari possibili:



→ CASO 1: il simbolo u inserito nello stack nello stato p sarà lo stesso che poi sarà estratto dallo stack nello stato q .

→ CASO 2: poiché c'è un ulteriore stato intermedio* in cui lo stack è vuoto, il simbolo u inserito nello stack nello stato p e il simbolo u estratto dallo stack nello stato q non sono necessariamente uguali

Vediamo come costruire le regole per i due casi.

CASO 2:

$$A_{pq} \rightarrow A_{pr} A_{rq} \quad \forall p, q, r \in Q$$

Il resto è identico al caso 1.

CASO 1:

$$(r, u) \in \delta(p, a, \epsilon) \rightarrow \text{push } u$$

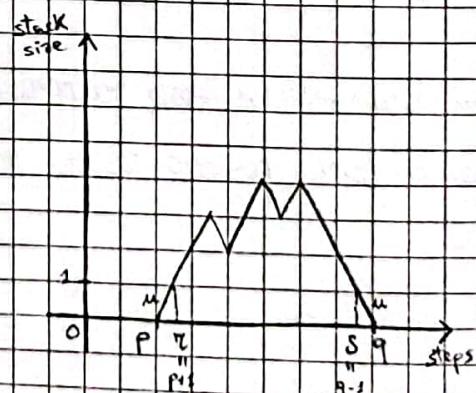
$$(q, \epsilon) \in \delta(s, b, u) \rightarrow \text{pop } u$$

Si può applicare ricorsivamente la seguente regola (facendo finta che lo stack rimanga vuoto):

$$A_{pq} \rightarrow a A_{rs} b \quad \forall a, b \in \Sigma \quad \forall p, q, r, s \in Q \quad \forall u \in \Gamma$$

All'ultima iterazione, invece, viene applicata la regola

$$A_{pp} \rightarrow \epsilon \quad \forall p \in Q$$



Dimostriamo per induzione che, se $A_{pq} \Rightarrow^* x$, allora x porta P dallo stato p con stack vuoto allo stato q con stack vuoto.

PASSO BASE ($A_{pq} \Rightarrow x$ cioè $p=q$, $x=\epsilon$, $A_{pp} \rightarrow \epsilon$):

Poiché nel caso banale la macchina non cambia stato ($p=q$), l'assunzione da dimostrare in realtà è ovvia: lo stack rimane certamente invariato.

PASSO INDUTTIVO (DERIVAZIONI LUNGHE FINO A K):

$A_{pq} \Rightarrow^* x$ CON K+1 DERIVAZIONI

Ci sono tre casi possibili:

① $A_{pp} \rightarrow \epsilon$

Applicando induttivamente questa regola, lo stack rimane certamente invariato.

② $A_{pq} \rightarrow a A_{rs} b$

$x = a y b$, $A_{rs} \Rightarrow^* y$ CON K DERIVAZIONI ($< K+1$)

\Rightarrow ci si può induttivamente ricondurre al caso base lasciando lo stack invariato.

③ $A_{pq} \rightarrow A_{pqr} A_{rq}$

$x = y z$, $A_{pqr} \Rightarrow^* y$ CON AL PIÙ K DERIVAZIONI

$A_{rq} \Rightarrow^* z$ CON AL PIÙ K DERIVAZIONI

\Rightarrow anche in questo scenario ci si può induttivamente ricondurre al caso base lasciando lo stack invariato. ✓

Dimostriamo ora per induzione il viceversa, ovvero:

$P(x)$ computa dallo stato p con stack vuoto allo stato q con stack vuoto \Rightarrow

$\Rightarrow A_{pq} \Rightarrow^* x$

$|P(x)|$

PASSO BASE (LUNGHEZZA DELLA COMPUTAZIONE = 0):

$P(x)$ non compie alcun passo $\Rightarrow p=q$, $x=\epsilon \Rightarrow A_{pp} \rightarrow \epsilon \quad \forall p \in Q$ è una regola che fa parte della grammatica

PASSO INDUTTIVO (LUNGHEZZA DELLA COMPUTAZIONE = K+1):

I casi possibili sono i due che abbiamo introdotto nella pagina precedente:

① Tra p e q non ci sono ulteriori stati intermedii in cui lo stack è vuoto, per cui il simbolo u inserito nello stack nello stato p sarà lo stesso che poi sarà estratto dallo stack nello stato q :

$$\begin{aligned} \delta(p, a, \varepsilon) &\ni (r, u) \\ \delta(s, b, u) &\ni (q, \varepsilon) \end{aligned} \Rightarrow A_{pq} \rightarrow a A_{rs} b$$

Di conseguenza, la stringa x si può scrivere come ayb

$$\Rightarrow A_{rs} \Rightarrow^* y \quad \Rightarrow \quad A_{pq} \Rightarrow^* ayb = x$$

② Tra p e q c'è un ulteriore stato intermedio r in cui lo stack è vuoto, per cui possiamo spiegare intuitivamente la computazione $P(x)$ in due computazioni, di cui una va dallo stato p allo stato r mantenendo lo stack vuoto, e l'altra va dallo stato r allo stato q sempre mantenendo lo stack vuoto. Quindi:

→ Possiamo scrivere x come yz .

→ Possiamo intuitivamente introdurre le seguenti regole nella grammatica a partire dalle nostre due sottocomputazioni (che hanno certamente lunghezza $\leq K$):

$$A_{pr} \Rightarrow^* y$$

$$A_{rq} \Rightarrow^* z$$

Di conseguenza, $\forall p, q, r \in Q$

$$A_{pq} \rightarrow A_{pr} A_{rq} \Rightarrow^* yz = x$$

28/03/2020

Corollario (che abbiamo già visto):

Se A è un linguaggio regolare $\Rightarrow A$ è un CFL.

Dim. alternativa:

A regolare $\Rightarrow \exists$ NFA M t.c. $L(M) = A$

M è un PDA che non usa lo stack $\Rightarrow A$ è un CFL

Pumping Lemma (per CFL):

Se A è un CFL, allora $\exists L > 0$ ("pumping length") tale che:
 se $s \in A$, $|s| \geq L \Rightarrow s = uvxyz$, dove:

$$1) \forall i \geq 0 \quad u v^i x y^i z \in A$$

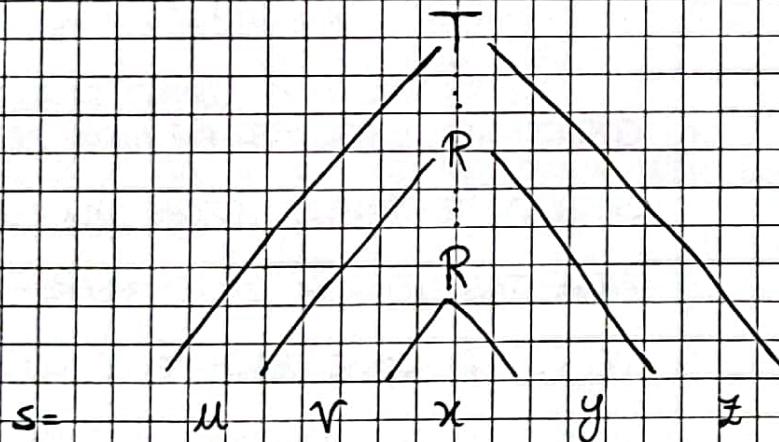
$$2) |v y| > 0$$

$$3) |v x y| \leq p$$

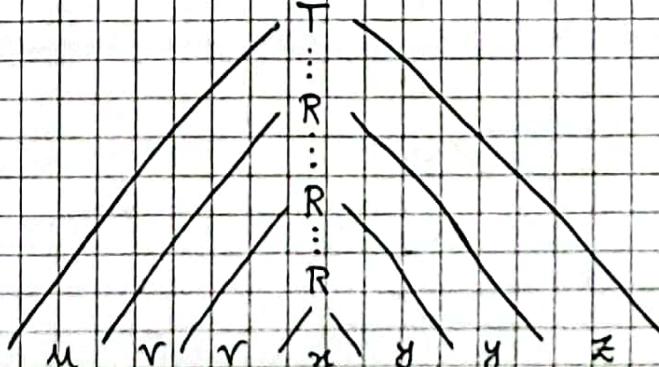
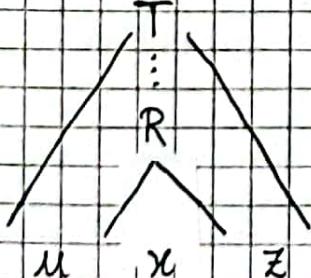
N.B.: Un linguaggio si dice "libero dal contesto" se è generato da una CFG $G = (V, \Sigma, R, T)$. A livello intuitivo, questo significa che una variabile $V \in V$ può essere trasformata secondo una delle regole R a prescindere dai caratteri che precedono o seguono V stesso (e quindi a prescindere dal "contesto").

Idea della dim:

Avere $|s| \geq p$ tra le ipotesi significa avere una stringa s abbastanza lunga da dover applicare il PRINCIPIO DELLA PICCIONATAI: ai passi di derivazione di $T \Rightarrow^* s$, secondo cui una variabile $R \in V$ deve ricopertasi almeno due volte nella generazione di s . Perciò, l'albero sintattico è del tipo:



In questo caso, la prima espansione di R produce le stringhe v, y , mentre la seconda espansione produce x . Poiché operiamo con una grammatica LIBERA DAL CONTESTO, nulla impedisce che la prima espansione di R produca subito la stringa x , oppure che la seconda espansione produca ancora v, y . Perciò, sono possibili anche i seguenti altri sintattici:



Se $|vxy| > p$, si può applicare il Pumping Lemma sulla sola sottostringa vxy .

Dim. formale:

$A \text{ CFL} \Rightarrow \exists \text{ CFG } G = (V, \Sigma, R, T) \text{ t.c. } L(G) = A$

$$V = \{V_1, V_2, \dots, V_m\}$$

$$R = \{V_i \rightarrow U_1^i U_2^i \dots U_k^i\}$$

$$b := \max_{1 \leq i \leq m} |U_1^i U_2^i \dots U_k^i|$$

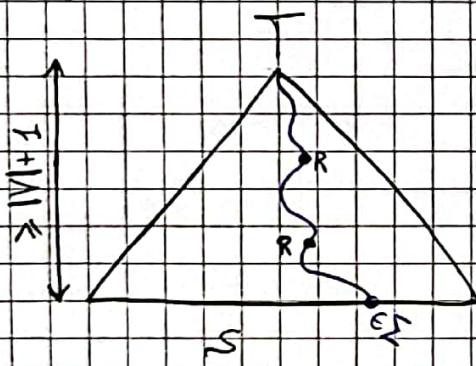
Questo significa che, in un albero sintattico, da una variabile qualsiasi possono partire al massimo b ramificazioni.

$$p := b^{|V|+1}$$

Sia $h_A :=$ altezza dell'albero sintattico.
Se $h_A \leq h \Rightarrow |S| \leq b^h *$

Sia $s \in A$, $|S| \geq p = b^{|V|+1}$. Allora l'altezza del suo albero sintattico deve essere $h_A \geq |V|+1$.

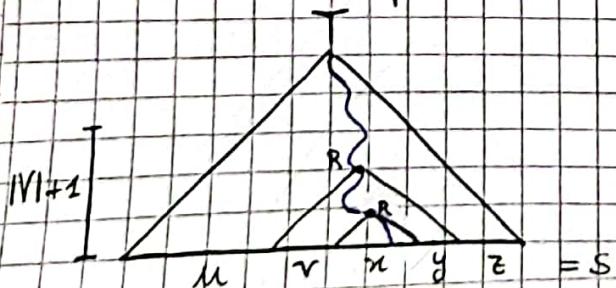
Se G è una grammatica ambigua, allora s può essere generata con più alberi sintattici differenti. In tal caso, consideriamo quello col minor numero di nodi.



CONSIDERIAMO IL PERCORSO PIÙ LUNGO DA T A UN SIMBOLO TERMINALE. POICHÉ $h_A \geq |V|+1$, ANCHE TALE PERCORSO DEVE ESSERE LUNGO ALMENO $|V|+1$. DI CONSEGUENZA, DEVE CONTENERE ALMENO $|V|+2$ NODI, DI CUI SOLO L'ULTIMO RAPPRESENTA UN TERMINALE. PERCIÒ CI SONO ALMENO $|V|+1$

NODI CHE RAPPRESENTANO VARIABILI E, PER IL PRINCIPIO DELLA PICCIONAIA, ALMENO UNA VARIABILE R DEVE AVERE PIÙ DI UN'OCCHERENZA.

In particolare, non sappiamo quanto è precisamente alto l'albero, ma la cosa certa è che almeno due occorrenze di R si trovano negli ultimi $|V|+1$ nodi del percorso che abbiamo considerato.



* SI DEVE AVERE NECESSARIAMENTE CHE $|YXY| \leq b^{|V|+1} = p \rightarrow$ Terzo punto dell'enunciato

Come abbiamo già anticipato nell'"idea della dimostrazione", poiché stiamo operando in una grammatica libera dal contesto, qualunque occorrenza di R può prototrope la parte v, y della stringa e qualunque occorrenza di R può prototrope x . Perciò:

Se $s = \cancel{uxxyz} \in A \Rightarrow uxz \in A, uvvxyyz \in A$ e così via.

Quindi anche il primo punto dell'enunciato è dimostrato.

Infine, possiamo dire che $|vxy| \neq 0$ perché, in caso contrario, avremmo che:

→ Esiste la possibilità di generare s con più alberi sintattici, perché il numero di occorrenze di R diventa arbitrario (supponiamo n). Infatti,

le prime $n-1$ occorrenze producono $\cancel{v}y=\epsilon$, mentre l'ultima produce x .

→ L'assunzione del considerare l'albero col minor numero di nodi viene violata, poiché, appunto, ^{in tal modo} possiamo eliminare la prima occorrenza di R (e quindi eliminare almeno un nodo) dal nostro albero mantenendo inalterata la stringa s generata.

In conclusione, anche il secondo punto dell'enunciato è dimostrato. ■

Esercizio 1:

$$B = \{a^m b^n c^r \mid m \geq 0\}$$

Dimostrare che B non è un CFL.

Per assurdo, supponiamo che B sia un CFL $\Rightarrow \exists$ pumping length ($p > 0$)

Consideriamo la stringa $s = a^p b^p c^p$. È ovidente che $|s| \geq p$.

$$s = \underbrace{a \dots a}_p \underbrace{b \dots b}_p \underbrace{c \dots c}_p \rightarrow \text{DOVREMO SCRIVERLA COME } \cancel{uxxyz}$$

$|vxy| \leq p \Rightarrow vxy$ non può contenere tutti e tre i simboli: (a, b, c) \Rightarrow quando pompo v, y , almeno uno dei simboli non viene aumentato, per cui, ad esempio, $uv^2xy^2z \notin B$.

ASSURDO $\Rightarrow B$ non è un CFL.

OPPURE:

Possiamo non sfruttare la condizione $|vxy| \leq p$. Vediamo come:

Se suddividiamo s in modo che v contenga sia delle a che delle b ($v \in a^+ b^+$), quando poniamo v (e y) verso l'alto, otteniamo la seguente stringa:

$\underbrace{a \dots a}_{u} \underbrace{a \dots b}_{v} \underbrace{a \dots b}_{v} \dots$

che evidentemente non è della forma $a^m b^m c^m$, per cui non appartiene al linguaggio B . Per lo stesso motivo, anche la sottostringa y non può contenere due simboli diversi.

L'unica alternativa per suddividere s è far sì che sia v sia y compresi in un unico simbolo. Ma, in questo modo, se per esempio vogliamo ponere v, y verso l'alto, aumenteremo solo il numero di due simboli su tre, ottenendo quindi una nuova stringa che non appartiene al linguaggio.

In definitiva, anche con questo secondo metodo abbiamo dimostrato che B non è un CFL.

Esercizio 2:

$$C = \{ a^i b^j c^k \mid 0 \leq i \leq j \leq k \}$$

Dimostrare che C non è un CFL.

Per assurdo, supponiamo che C sia un CFL $\Rightarrow \exists$ pumping length ($p > 0$)
Consideriamo la stringa $s = a^p b^p c^p$. È evidente che $|s| \geq p$.

$$s = \underbrace{a \dots a}_p \underbrace{b \dots b}_p \underbrace{c \dots c}_p \quad \rightarrow \text{DOVREMMO SCRIVERLA COME } a^p b^p c^p$$

$$|vxy| \leq p \Rightarrow vxy \notin a^+ b^+ c^+$$

CASO 1 ("No a"):

$vxy \in b^* c^* \Rightarrow uxz$ ha meno b e/o c che $a \Rightarrow uxz \notin C$

CASO 2 ("No c"):

$vxy \in a^* b^* \Rightarrow uvxyz$ ha più a e/o b che $c \Rightarrow uvxyz \notin C$

CASO 3 ("No b" \rightarrow "No c"):

$vxy \in a^* \Rightarrow ry \in a^+ \Rightarrow uvxyz$ ha più a che b e $c \Rightarrow uvxyz \notin C$

CASO 4 ("No b" → "No a"):

$\forall xy \in C^* \Rightarrow \forall y \in C^* \Rightarrow uxz$ ha meno c che a + b $\Rightarrow uxz \notin C$

ASSURDO $\Rightarrow C$ non è un CFL.

Teorema:

I CFL sono chiusi rispetto alle operazioni di unione, concatenazione e Kleene.

Dim:

A, B sono CFL $\Rightarrow \exists G_A = (V_A, \Sigma_A, R_A, S_A)$

$\exists G_B = (V_B, \Sigma_B, R_B, S_B)$

$\rightarrow G_{A \cup B} = (V_A \cup V_B \cup \{S\}, \Sigma_A \cup \Sigma_B, R_A \cup R_B \cup \{S \rightarrow S_A | S_B\}, S)$

$\rightarrow G_{A \cdot B} = (V_A \cup V_B \cup \{S\}, \Sigma_A \cup \Sigma_B, R_A \cup R_B \cup \{S \rightarrow S_A S_B\}, S)$

$\rightarrow G_{A^*} = (V_A \cup \{S\}, \Sigma_A, R_A \cup \{S \rightarrow SS_A | \epsilon\}, S)$

Teorema:

I CFL sono chiusi rispetto all'operazione di rovesciamento.

$A \stackrel{\text{def}}{\in} \text{CFL} \Rightarrow A^R = \{w_k \dots w_1 | w_k \dots w_1 \in A\}$ è un CFL

Dim:

A CFL $\Rightarrow \exists G_A = (V_A, \Sigma_A, R_A, S_A)$

$G_{A^R} = (V_A, \Sigma_A, R_{A^R}, S_A)$, dove le regole di R_{A^R} sono le stesse di R_A ma con i termini di destra tutti invertiti di ordine.

Teorema:

I CFL NON sono chiusi rispetto all'intersezione.

Dim:

Facciamo un controesempio.

Abbiamo già provato che $B = \{a^m b^n c^n | m \geq 0\}$ non è un CFL.

Ma B è dato dall'intersezione tra $B' = \{a^m b^n c^i | m \geq 0 \wedge i \geq 0\}$ e

$B'' = \{a^i b^n c^m | n \geq 0 \wedge i \geq 0\}$, che già sappiamo essere due CFL.

Teorema:

I CFL sono chiusi rispetto all'automorfismo ($h: \Sigma \rightarrow \Gamma^*$):

A è un CFL $\Rightarrow h(A) = \{h(w) \mid w \in A\}$ è un CFL

Dim:

A CFL $\Rightarrow \exists G = (V, \Sigma, R, T)$ che genera A .

$G_H = (V \cup H_\Sigma, \Gamma, R \cup R_H, T)$, dove, per ogni occorrenza di un terminale $a \in \Sigma$ nella parte destra delle regole di R , si sostituisce ~~tutta~~
a con una nuova variabile H_a e si introduce la regola $H_a \rightarrow h(a) \in \Gamma^*$

Abbiamo osservato che, se prendiamo un PDA e non usiamo lo stack, stiamo effettivamente avendo a che fare con un NFA. Ma esiste un particolare automa a pila che ha un corrispondente deterministico? La risposta in effetti è sì.

Definizione (automa push down deterministico - DPDA):

È una macchina $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ definita come un PDA con delle differenze nella funzione di transizione δ :

$$\delta: Q \times \Sigma \times \Gamma \rightarrow (Q \times \Gamma) \cup \{\emptyset\}$$

$$(q, a, x) \longrightarrow \begin{cases} \emptyset \\ (q', x') \end{cases}$$

$\forall q \in Q, a \in \Sigma, x \in \Gamma$ esattamente un solo elemento dell'insieme $\{\delta(q, a, x), \delta(q, \epsilon, x), \delta(q, a, \epsilon), \delta(q, \epsilon, \epsilon)\}$ è diverso da \emptyset
 $\Rightarrow |\{\delta(q, a, x), \delta(q, \epsilon, x), \delta(q, a, \epsilon), \delta(q, \epsilon, \epsilon)\} \setminus \{\emptyset\}| = 1$

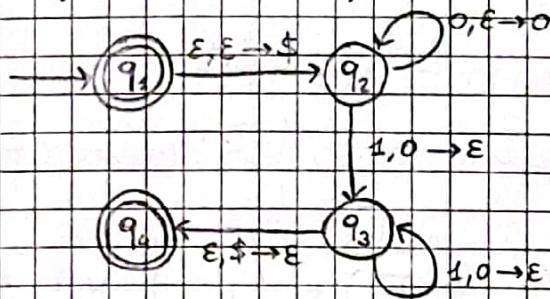
In altre parole:

\rightarrow Se lo stack non è vuoto, allora per ogni simbolo letto dall'input c'è una sola mossa legale da fare.

\rightarrow Se lo stack è vuoto, allora per ogni simbolo letto dall'input o c'è una sola mossa legale che non effettua una "pop" dallo stack, o non c'è alcuna mossa legale.

Esempio 1:

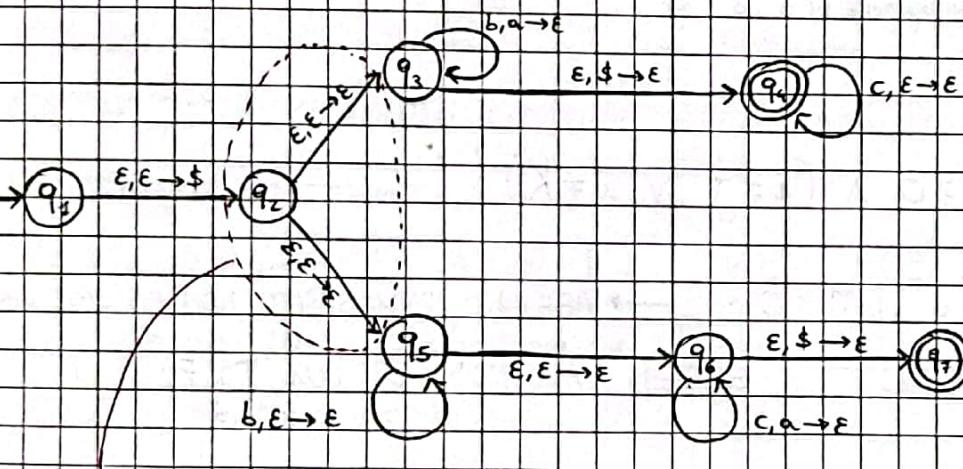
$$B = \{0^m 1^m \mid m \geq 0\}$$



Questo automa, che abbiamo già costruito in precedenza, secondo la definizione è, in realtà, un DPDA $\Rightarrow B$ è un linguaggio libero dal contesto DETERMINISTICO (DCFL).

Esempio 2:

$$\{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i=j \vee i=k)\}$$



Questa biforcazione non è consentita nei DPDA: l'automa non è deterministico.

Teorema:

Ogni linguaggio regolare è un DCFL.

Dim:

A regolare $\Rightarrow \exists$ DFA M s.t. $L(M) = A$

Ma M è un DPDA che non usa lo slack $\Rightarrow A$ è un DCFL

Teorema:

I DCFL sono chiusi rispetto all'operazione di complemento.

Questo teorema deriva dal seguente lemma:

Lemma:

Ogni DPDA ha un DPDA equivalente che legge sempre per intero la stringa data in input (anche quando è da rifiutare).

Corollario:

Se A è un CFL, A^c NON è un CFL $\Rightarrow A$ NON è un DCFL.

Lemma:

Se A è un CFL, B è un linguaggio regolare $\Rightarrow A \cap B$ è un CFL.

Dim:

A CFL $\Rightarrow \exists$ PDA $M_A = (Q_A, \Sigma, \Gamma, S_A, q_0^A, F_A)$ t.c. $L(M_A) = A$

B regolare $\Rightarrow \exists$ DFA $M_B = (Q_B, \Sigma, \delta, q_0^B, F_B)$ t.c. $L(M_B) = B$

$M_{A \cap B} = (Q_A \times Q_B, \Sigma, \Gamma, \delta', (q_0^A, q_0^B), F_A \times F_B)$

$$\text{E' UNA COMBINAZIONE DI } \delta^A, \delta^B \rightarrow \delta'((q^A, q^B), x, y) = \left\{ \begin{array}{l} \delta(q^A, x, y) | (q^A, y) \in \delta^A(q^A, x, y) \\ \delta(q^B, x, y) | (q^B, y) \in \delta^B(q^B, x, y) \end{array} \right\} \rightarrow \text{se } x \in$$

Transizioni che non consumano simboli dell'input e simulano sia il PDA

Transizioni che consumano simboli dell'input e

simulano sia il PDA che il DFA

Esempio:

$A = \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i \neq j \vee i \neq k)\}$ \rightarrow È UN CFL

$A^c \cap a^* b^* c^* = \{a^m b^m c^m \mid m \geq 0\}$ \rightarrow ABBIAMO DIMOSTRATO NON ESSERE UN CFL

$\Rightarrow A^c$ NON è un CFL $\xrightarrow{\text{Corollario}} A$ NON è un DCFL

Teorema:

La classe dei DCFL NON è chiusa rispetto a unione, intersezione, operatore f. Kleene, concatenazione, rovesciamento e omomorfismo.

03/04/2020

Definizione:

Sia A un linguaggio t.c. $A \in \Sigma^*$ e sia \dashv un simbolo t.c. $\dashv \notin \Sigma$. Un "linguaggio marcato" è $A\dashv := \{w\dashv \mid w \in A\}$.

Teorema:

A è un DCFL $\Leftrightarrow A\dashv$ è un DCFL

Dim:

$\Rightarrow) A$ DCFL $\Rightarrow \exists$ DPDA P t.c. $L(P) = A$

Possiamo costruire un automa P' che simula P finché non legge " \dashv ". Se nel momento in cui legge " \dashv ", si trova in uno

stato di accettazione, allora accetta la stringa data in input, altrimenti no. Perciò, P' è un DPDA t.c. $L(P') = A \dashv$ \square

$\Rightarrow A \dashv \text{DCFL} \Rightarrow \exists \text{DPDA } P \text{ t.c. } L(P) = A \dashv$

Anche in questo caso bisognerebbe costruire un PDPA P' che, dato $w \in \Sigma^*$, $P'(w)$ simuli $P(w)$, tenendo però traccia di che cosa farebbe P se il prossimo carattere da leggere fosse " \dashv ". La difficoltà sta nel fatto che, nel momento in cui P legge " \dashv ", potrebbe esaminare il contenuto dello stack per decidere se accettare la stringa data in input o meno (e potrebbero servire molti passi). Di conseguenza P' , nel momento in cui simula P , deve tenere anche traccia di se il contenuto corrente dello stack porta all'accettazione della stringa o meno.

$$P = (Q, \Sigma \cup \{\dashv\}, \Gamma, \delta, q_0, F) \quad \text{t.c. } L(P) = A \dashv$$

Imponiamo una condizione su P che NON è limitante: le transizioni δ possono essere solo di 3 tipi:

(1) SOLO LETTURA DI UN SIMBOLO DALL'INPUT

$$\delta(q, a, \epsilon) = (\tau, \epsilon)$$

(2) SOLO PUSH NELLO STACK

$$\delta(q, \epsilon, x) = (\tau, x)$$

(3) SOLO POP DALLO STACK

$$\delta(q, \epsilon, x) = (\tau, \epsilon)$$

Costruiamo P' :

$$P' = (Q', \Sigma, \Gamma', \delta', q'_0, F')$$

$$\Gamma' := \Gamma \cup 2^Q$$

STACK DI P

x
y
z
:

STACK DI P'

Rx
z
Ry
y
Rz
z
:

Insieme degli stati da cui P accetterebbe se nel suo stack ci fossero i simboli y, z, \dots

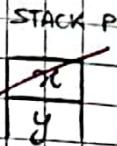
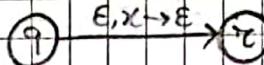
DEFINIAMO LA FUNZIONE DI TRANSIZIONE δ'

$$\delta'(q'_0, \epsilon, \epsilon) = (q_0, R_0)$$

$R_0 = \{q \in Q \mid P$ sia dallo stato q può entrare in uno stato di accettazione senza leggere alcun simbolo di input $\} = \{\text{stati immediatamente accettanti}\}$

→ Caso in cui P effettua una pop:

$$\delta(q, \epsilon, x) = (\tau, \epsilon)$$



P' DEVE EFFETTUARE 2 POP:

$$\delta'(q, \epsilon, R) = (q', \epsilon) \quad \forall R \in 2^Q$$

$$\delta'(q', \epsilon, x) = (\tau, \epsilon)$$

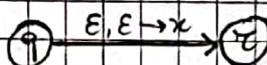


STACK P'

P _{in}
x
R _y
y
⋮

→ Caso in cui P effettua una push:

$$\delta(q, \epsilon, x) = (\tau, x)$$



STACK P

x
y

P' DEVE EFFETTUARE LE SEGUENTI TRANSIZIONI:

$$\delta'(q, \epsilon, R) = (q^{R_{\text{new}}}, R) \quad \forall R \in 2^Q$$

$$\delta'(q^{R_{\text{new}}}, \epsilon, x) = (q^{R_{\text{new}} \cup S}, x)$$

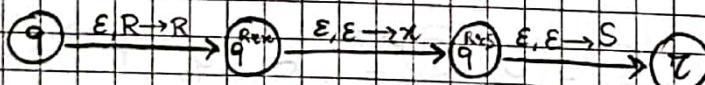
$$\delta'(q^{R_{\text{new}} \cup S}, \epsilon, \epsilon) = (\tau, S)$$

$$S := R_0 \cup \{ q \in Q \mid \delta(q, \epsilon, x) = (\tau, \epsilon), \forall x \in R \}$$

STACK P'

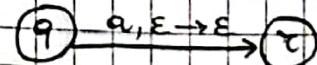
R _x	→ S
x	
R _y	→ R
y	
⋮	

SE IN CIMA ALLO STACK AL POSTO DI R CI FOSSE UN R'
DIVERSO, P' AVREBBE LA NECESSITÀ DI PASSARE PER UNO
STATO INTERMEDIO DIVERSO q^{R'_{\text{new}}} PERCHÉ POI DOVRÀ ESSERE
RÉ DIVERSO L'INSIEME S



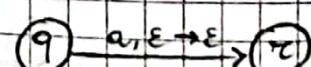
→ Caso in cui P legge dall'input un simbolo ≠ + e, a partire dallo stato corrente di P, non escono frecce marcate con + e con -:

$$\delta(q, a, \epsilon) = (\tau, \epsilon) \quad a \neq +$$



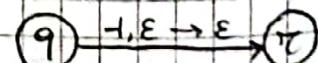
P' DEVE SIMULARE PERFETTAMENTE P:

$$\delta'(q, a, \epsilon) = (\tau, \epsilon)$$



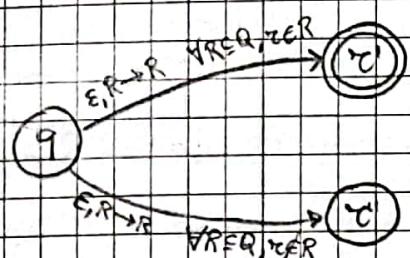
→ Caso in cui P legge dall'input "—" e, a partire dallo stato corrente di P, non escono altre frecce:

$$\delta(q, -, \epsilon) = (\tau, \epsilon)$$



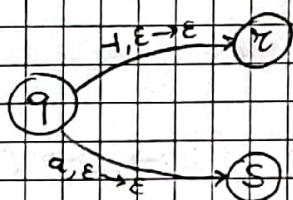
P' DEVE EFFETTUARE UNA DELLE SEGUENTI TRANSIZIONI:

$$\begin{cases} \delta'(q, \epsilon, R) = (\tau', R) & \tau' \in F' \quad \text{se } \tau \in R \quad \forall R \in 2^{\omega} \\ \delta'(q, \epsilon, R) = (\tau'', R) & \tau'' \notin F' \quad \text{se } \tau \notin R \quad \forall R \in 2^{\omega} \end{cases}$$



→ Caso in cui P legge dall'input un simbolo ϵ , a partire dallo stato corrente di P , escono sia frecce marcate con " \dashv ", sia frecce marcate con simboli $\neq \dashv$:

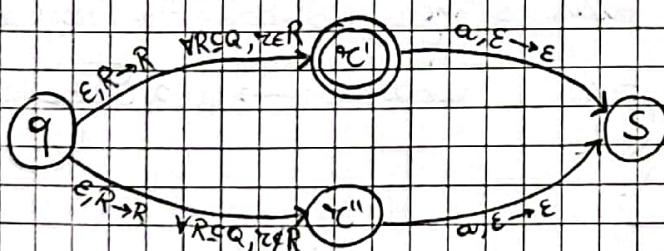
$$\delta(q, a, \epsilon) = (S, \epsilon)$$



P' DEVE EFFETTUARE UNA DELLE SEGUENTI COPPIE DI TRANSIZIONI:

$$\begin{cases} \delta'(q, \epsilon, R) = (\tau', R) & \tau' \in F' \quad \text{se } \tau \in R \quad \forall R \in 2^{\omega} \\ \delta'(\tau', a, \epsilon) = (S, \epsilon) & \end{cases}$$

$$\begin{cases} \delta'(q, \epsilon, R) = (\tau'', R) & \tau'' \notin F' \quad \text{se } \tau \notin R \quad \forall R \in 2^{\omega} \\ \delta'(\tau'', a, \epsilon) = (S, \epsilon) & \end{cases}$$



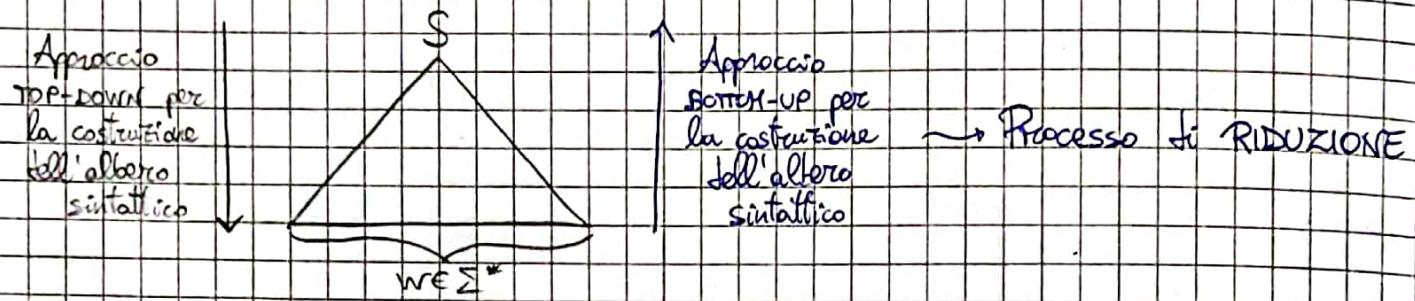
NB: Se $a = \dashv \Rightarrow$ la computazione di P' è analoga a quella del P nel ultimo caso elencato.

Adesso rimane solo da provare formalmente che, con questa costituzione di P' , si ottiene che $L(P') = A$ (l'indicazione lasciata per esercizio).

Torniamo a parlare delle grammatiche libere dal contesto...

CFG $G = (V, \Sigma, R, S)$

$S \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow w \in \Sigma^*$ → Processo di DERIVAZIONE



STEP DI RIDUZIONE: $u \rightarrow u'$

"reducing string" ↗ "reduced string" ↘

SI TROVA NELLA PARTE DESTRA DI UNA REGOLA

SI TROVA NELLA PARTE SINISTRA DELLA STESSA REGOLA

Se $u \rightarrow^* v$ ($u \rightarrow \dots \rightarrow v$), allora si dice che u è RIDUCIBILE a v .

Se $u \rightarrow^* S$, allora \exists una riduzione da u (alla variabile iniziale S).

Esempio:

Supponiamo di avere una CFG che ha come regola $S \rightarrow abSb \mid \epsilon$

DERIVAZIONE: $S \Rightarrow abSb \Rightarrow ababSbb \Rightarrow ababbb$

RIDUZIONE: $ababbb \xrightarrow{\epsilon} ababSbb \xrightarrow{!} abSb \xrightarrow{!} S$

NB: Esistono riduzioni a partire dalla stringa ababbb che non possono portare a S . Per esempio: $ababbb \xrightarrow{!} abaSbb \times$

Definizione (riduzione a sinistra):

Ogni "reducing string" viene ricordata solo dopo tutte le altre "reducing string" che si trovano interamente alla sua sinistra.

NB: Una riduzione a sinistra è una derivazione a destra al rovescio.

Sia $w \in L(G)$.

Riduzione a sinistra:

$w \xrightarrow{!} u_1 \xrightarrow{!} u_2 \xrightarrow{!} \dots \xrightarrow{!} u_i \xrightarrow{!} u_{i+1} \xrightarrow{!} \dots \xrightarrow{!} S$

Consideriamo la riduzione $u_i \rightarrow u_{i+1}$

Supponiamo che tale riduzione sia relativa alla regola $T \rightarrow h$

Allora: $u_i = xhy$

$$u_{i+1} = xTy$$

$$T \in V; h, x \in (\Sigma \cup V)^*; y \in \Sigma^*$$

Definiamo con HANDLE (maniglia) di u_i la coppia $(h, T \rightarrow h)$ costituita dalla stringa h e dalla regola $T \rightarrow h$.

NB: Tale definizione è valida solo per le stringhe che (in partenza) appartengono a $L(G)$.

Teorema:

Se una grammatica NON è ambigua \Rightarrow ogni stringa valida ha soltanto un handle.

Dim:

G non ambigua $\Rightarrow \forall w \in L(G) \exists!$ albero sintattico che genera $w \Rightarrow \exists!$ derivazione a destra per $w \Rightarrow \exists!$ riduzione a sinistra per $w \Rightarrow \exists!$ handle $\forall u_i$.

Vediamo un esempio di una grammatica ambigua • G_0 :

$$R \rightarrow S \mid T$$

$$S \rightarrow aSb \mid ab$$

$$T \rightarrow aTbb \mid aT \mid a$$

$$R \Rightarrow S \Rightarrow aSb \Rightarrow aabb$$

$$R \Rightarrow T \Rightarrow aTbb \Rightarrow aabb$$

} Sono 2 derivazioni a destra differenti che portano alla stringa aabb

$$aabb \rightarrow aTbb \rightarrow T \rightarrow R$$

$$aabb \rightarrow aSb \rightarrow S \rightarrow R$$

} Sono 2 riduzioni a sinistra differenti a partire dalla stringa aabb

Per esempio, la stringa aabb ha due handle differenti:

$$(a, T \rightarrow a)$$

$$(ab, S \rightarrow ab)$$

E' esaminiamo adesso un'altra grammatica G_1 :

$$R \rightarrow S T$$

$$S \rightarrow a S b \mid a b$$

$$T \rightarrow a T b b \mid a b b$$

$$L(G_1) = B \cup C$$

$$B = \{a^m b^m \mid m \geq 1\}$$

$$C = \{a^m b^{2m} \mid m \geq 1\}$$

È facile convincersi che $L(G_1)$ NON è un DCFN perché l'automa corrispondente, una volta terminata la a, dovrebbe "indovinare" se dovrà contare lo stesso numero di b oppure il doppio, per cui il non determinismo è fondamentale.

Vediamo cosa succede in termini di riduzione a sinistra:

$$aaaabb \rightarrow aaSbb \rightarrow aSb \rightarrow S \rightarrow R$$

$$aaaabb \rightarrow aaTbbbb \rightarrow aTbb \rightarrow T \rightarrow R$$

Consideriamo ora un'altra grammatica G_2 :

$$R \rightarrow 1S \mid 2T$$

$$S \rightarrow aSb \mid ab$$

$$T \rightarrow aTbb \mid abb$$

$$L(G_2) = B \cup C$$

$$B = \{1a^m b^m \mid m \geq 1\}$$

$$C = \{2a^m b^{2m} \mid m \geq 1\}$$

Stavolta il linguaggio $L(G_2)$ è un DCFN perché in effetti l'automa corrispondente capisce quante b dovrà contare già dalla lettura del primo simbolo (1 oppure 2).

Vediamo un'ulteriore grammatica G_3 :

$$S \rightarrow T \dashv$$

$$T \rightarrow T(T) \mid \epsilon$$

Esempio di riduzione a sinistra:

$$\begin{array}{ccccccc} ()() & \xrightarrow{\quad} & T(()) & \xrightarrow{\quad} & T(T)() & \xrightarrow{\quad} & T() \dashv \\ & \uparrow \varepsilon & & & & \uparrow & \\ & T \dashv & & & & T(T) & \xrightarrow{\quad} \\ & \xrightarrow{\quad} & & & & \xrightarrow{\quad} & S \end{array}$$

Definizione:

Un handle di una stringa valida $v = xhy$ è un HANDLE FORZATO se h è l'unico handle in ogni stringa valida xhw , dove $w \in \Sigma^*$.

Definizione:

Una CFG è DETERMINISTICA (DCFG) se ogni stringa valida ha un handle forzato.

Per esempio, G_2 NON è una DCFG perché l'handle relativo a una stringa valida dipende da quante b essa contiene a destra.

04/04/2020

Definizione (automa DK):

Per ogni CFG G possiamo costruire un DFA ΔK che accetta il suo input $z \xrightarrow{\text{DK}}$

- (a) z è il prefisso di una certa stringa valida $v = zy$
- (b) z termina con un handle di v .

→ Un DK riconosce le maniglie delle stringhe valide di G .

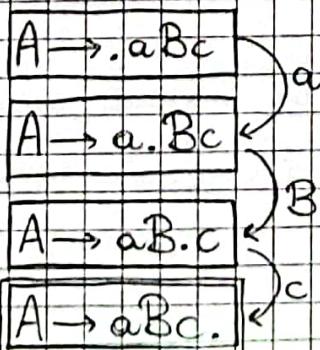
UN PRIMO APPRACCIAMENTO PER COSTRUIRE UNA MACCHINA DK:

- Partire da un NFA J
- Ricavare da J un NFA K
- Ricavare da K un DFA ΔK

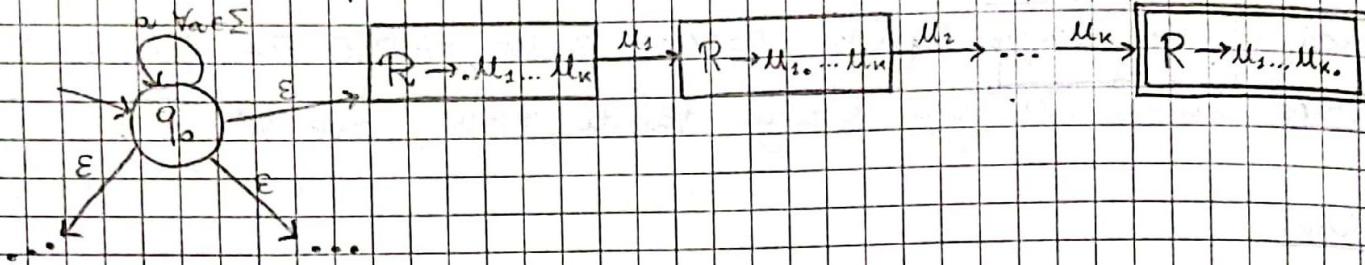
J : NFA che accetta ogni stringa in input che termina con la parte destra di una regola della grammatica G .

Gli stati di J sono detti DOTTED RULES.

Per esempio, se abbiamo la regola $A \rightarrow aBc$, gli stati corrispondenti sono:

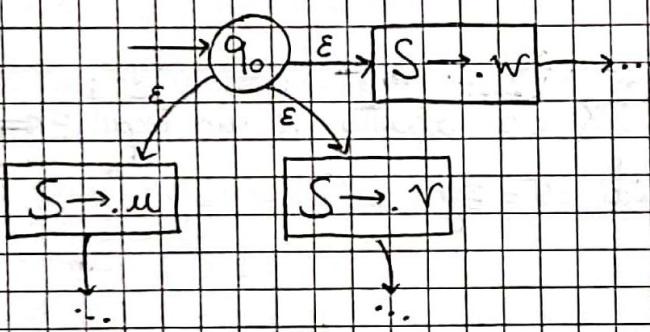


Vediamo com'è fatto l'automa J:



K: NFA che segue la stessa logica di J ma che segue passo passo tutte le regole della derivazione a partire dalla variabile iniziale S della grammatica.

Vediamo com'è fatto K, supponendo di avere la regola $S \rightarrow u.v.r.w$, $u, v, r, w \in \Sigma$



$\forall a \in (\Sigma \cup V), B \rightarrow uav$

$$B \rightarrow u.av \xrightarrow{a} B \rightarrow ua.v$$

$\forall B \rightarrow uCr, C \rightarrow r$

$$B \rightarrow u.Cr \xrightarrow{\epsilon} C \rightarrow r$$

DK: DFA che si ottiene dal NFA K esattamente allo stesso modo che a
biaimo visto all'inizio del corso per ricavare un DFA da un qualunque
NFA.

DK-TEST:

Data una grammatica CFG G, si costruisca il DFA DK corrispondente.
Allora G è una DCFG \iff

1) Ogni stato di accettazione di DK include esattamente una regola completa.

$$B \rightarrow \dots$$

2) Ogni stato di accettazione non include alcuna "faded rule" in cui un simbolo terminale segue immediatamente il punto.

$$B \rightarrow \dots.a \dots \quad a \in \Sigma$$

→ NO

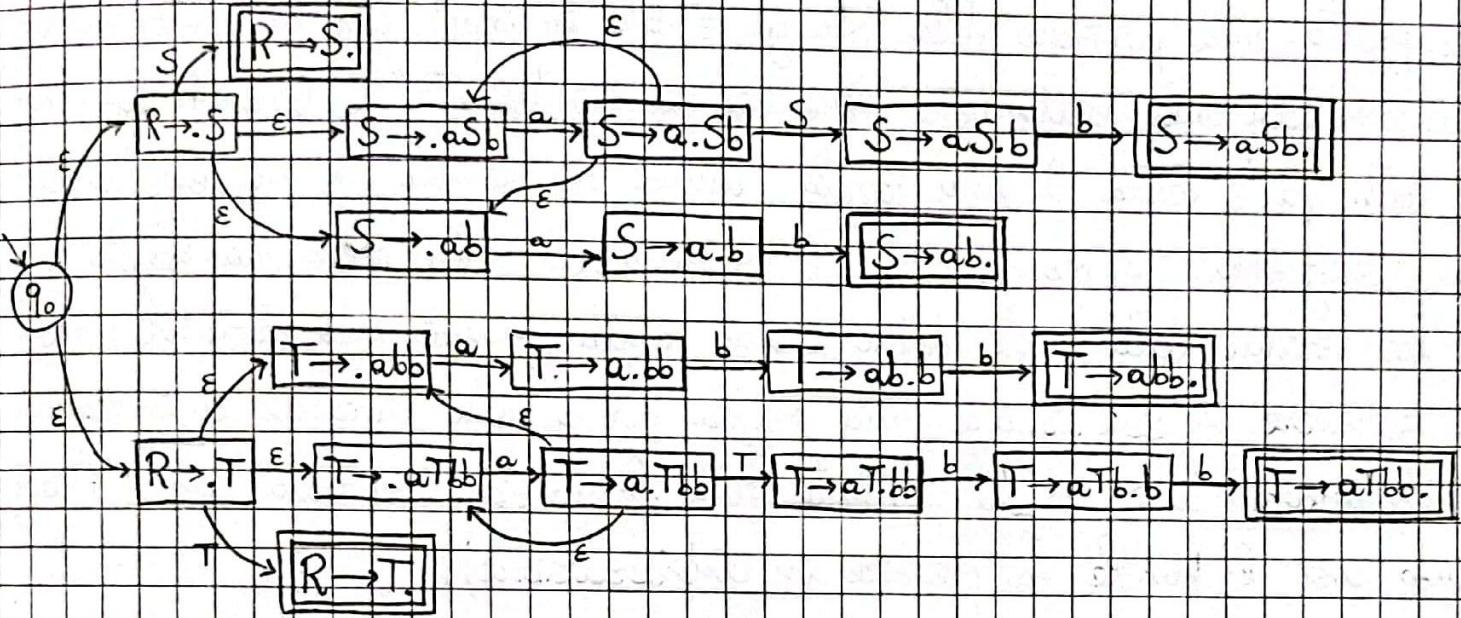
Esercizio 1:

Costruire un NFA K a partire dalla seguente grammatica G_1 :

$$R \rightarrow S \mid T$$

$$S \rightarrow aSb \quad ab$$

$$T \rightarrow aTbb \mid abb$$

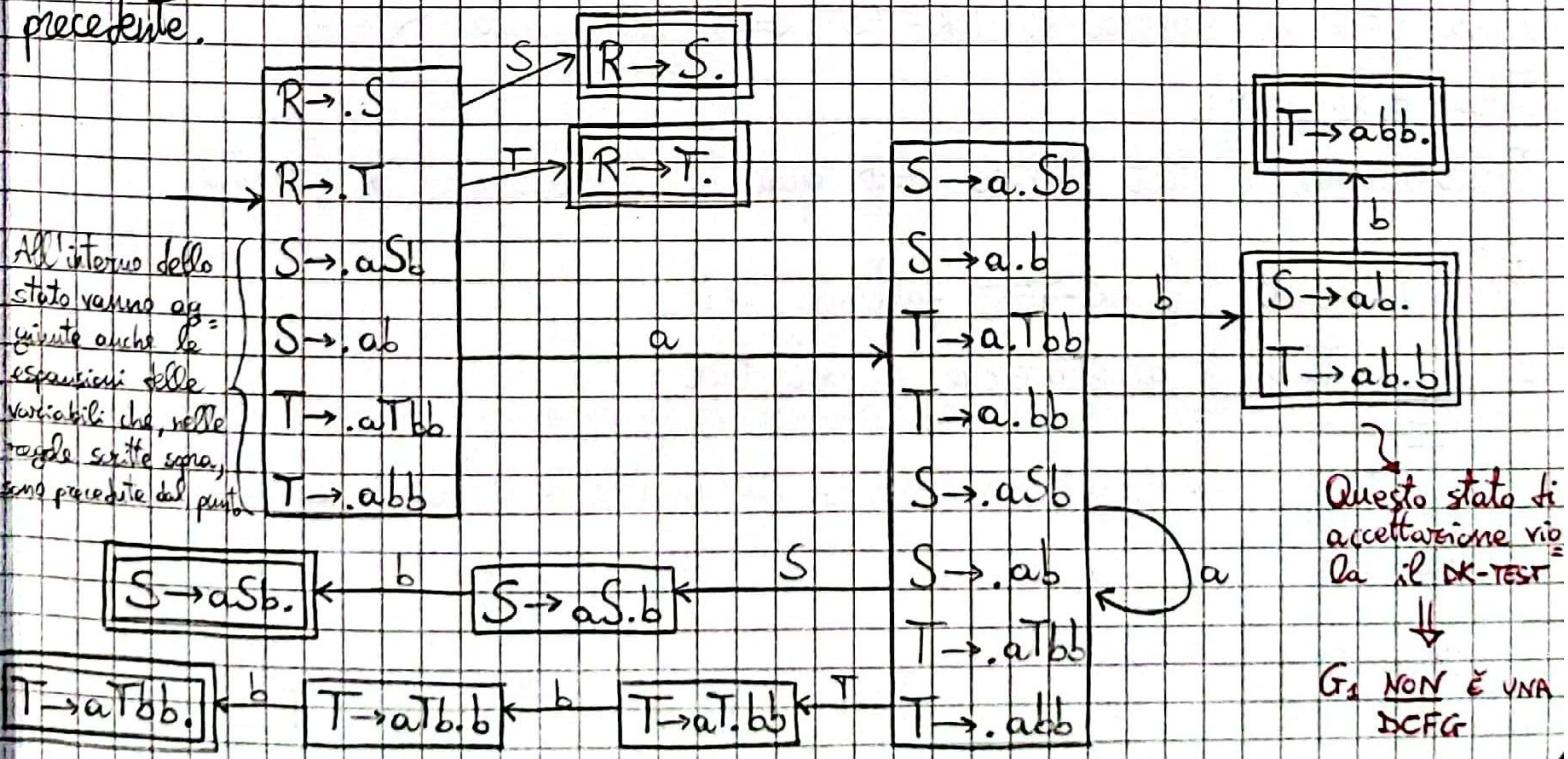


NB: K accetta per esempio la stringa $aaSb$, poiché:

- (a) $aaSb$ è il prefisso di una stringa valida ($aaSbb$)
- (b) $aaSb$ termina con un handle di $aaSbb$ (aSb).

Esercizio 2:

Costruire un DFA DK a partire dalla stessa grammatica G_1 dell'esercizio precedente.



Teorema:

Il NFA K può entrare nello stato $\boxed{T \rightarrow u.v}$ leggendo l'input $z \iff z = xu$ e $xvuy$ è una stringa valida con un handle $(uv, T \rightarrow uv)$ per qualche $y \in \Sigma^*$.

Idea della dim:

K confronta una porzione della stringa z data in input con la parte destra di una regola alla volta (che viene via via "indovinata"). Quando la stringa della parte destra di una regola include un simbolo, K può effettuare una "shift-move" e procedere con l'esaminazione della medesima regola. Se la stringa della parte destra di una regola include una variabile, K può scegliere di selezionare una regola per questa variabile e iniziare a esaminare tale regola tramite una transizione marcata con " ϵ " (cambiando così l'handle da prendere in considerazione).

Cordillario:

Il NFA K può entrare in uno stato di accettazione $\boxed{T \rightarrow h.}$ leggendo l'input $z \iff z = xh$ e $(h; T \rightarrow h)$ è un handle di qualche stringa valida xhy , $y \in \Sigma^*$.

Teorema:

Una CFG G supera il DK-TEST $\iff G$ è una DCFG.

Idea della dim:

Provare che il test fallisce \iff qualche handle non è forzato.

Effettivamente, il DK-TEST fallisce se:

(I) Uno stato di accettazione è del tipo:

$$\boxed{T \rightarrow h.}$$

\longrightarrow IN QUESTO CASO LA GRAMMATICA È ADDIRITTURA AMBIGUA

OPPURE

(II) Uno stato di accettazione è del tipo:

$$\boxed{T \rightarrow h.}$$
$$\boxed{S \rightarrow h.a}$$

LA PRIMA REGOLA IMPLICA CHE $\exists xhy$ VALIDA CHE HA COME HANDLE $(h, T \rightarrow h)$
LA SECONDA REGOLA IMPLICA CHE $\exists xhay'$ VALIDA CHE HA COME HANDLE $(ha, S \rightarrow ha)$
Il handle NON è forzato.

Esercizio 3:

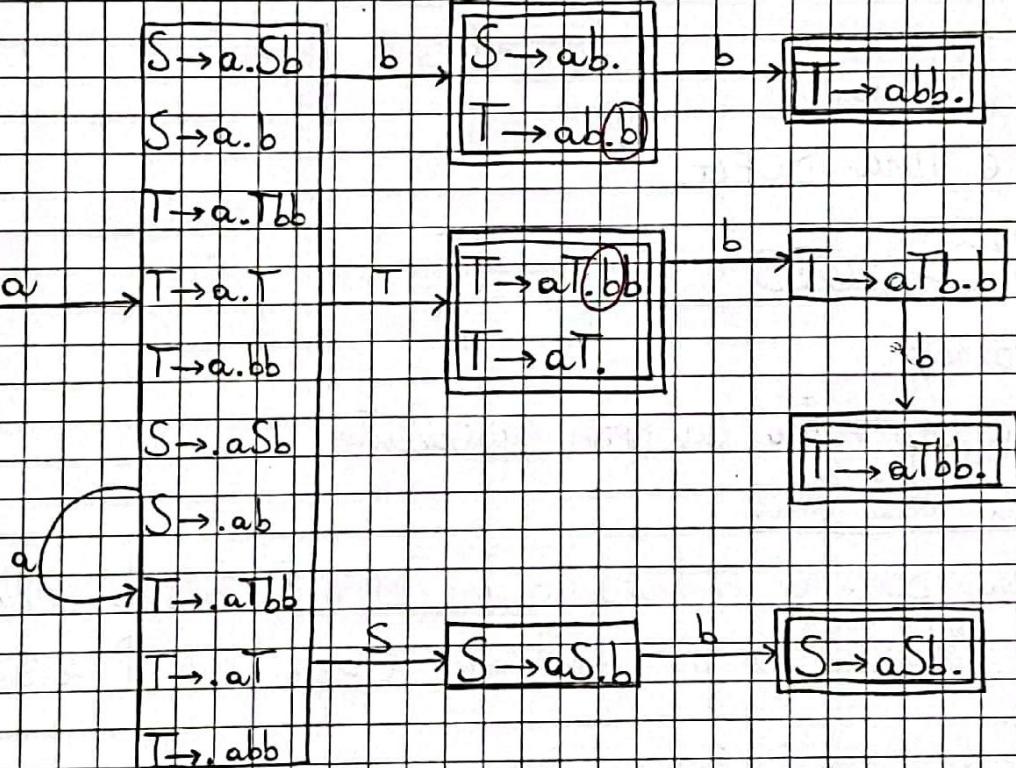
Determinare se la seguente grammatica G_0 è una DCFG oppure no:

$$R \rightarrow SIT$$

$$S \rightarrow aSb \mid ab$$

$$T \rightarrow aTbb \mid aT \mid abb$$

$$\boxed{R \rightarrow S.}$$
$$\boxed{R \rightarrow T.}$$
$$\xrightarrow{} \boxed{S \rightarrow .aSb}$$
$$\boxed{S \rightarrow .ab}$$
$$\boxed{T \rightarrow .aTbb}$$
$$\boxed{T \rightarrow .aT}$$
$$\boxed{S / T \rightarrow .abb}$$



G_0 NON è una DCFG.

Esercizio 4:

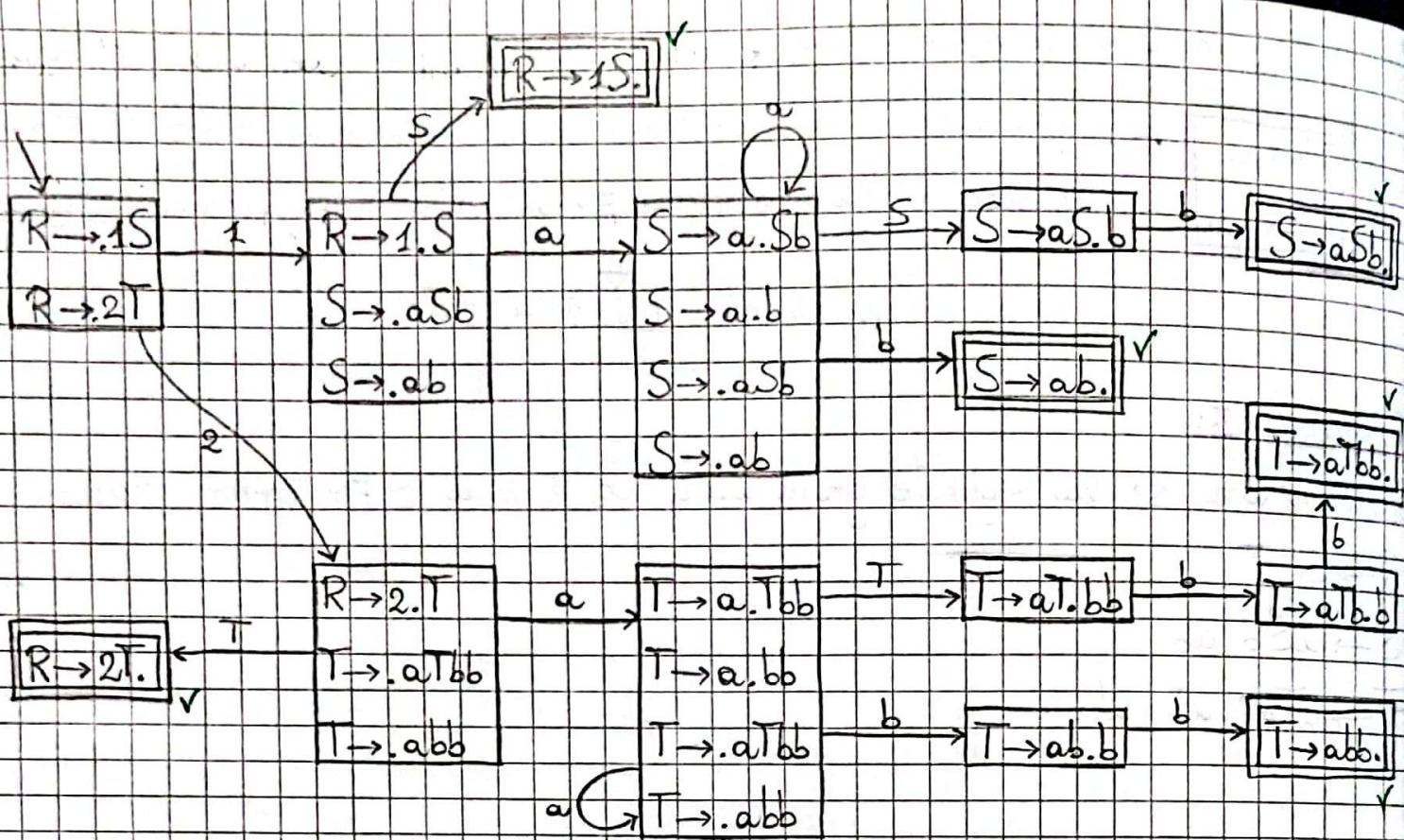
Determinare se la seguente grammatica G_2 è una DCFG oppure no:

$$R \rightarrow 1S \mid 2T$$

$$S \rightarrow aSb \mid ab$$

$$T \rightarrow aTbb \mid abb$$

Vedere pagina seguente



G_2 è una DCFG.

05/04/2020

Teorema:

Ogni DCFG ha un DPDA equivalente.

Idea della dim:

Dalla DCFG G costruiamo un DPDA P basato sulla simulazione del DFA Δ_K . Vediamo con uno pseudocodice i passi che P dovrebbe compiere:

OPERATIONS OF P (DPDA, Σ):

$\leftarrow \Sigma$ is the input string

$q :=$ start state of Δ_K

$\leftarrow q$ is the actual state of P

$i := 1$

$\leftarrow i$ is the index of the next symbol that will be read by P

while $q \neq$ $S \rightarrow h$.

push (q)

if $q =$ $T \rightarrow u$ then

pop and discard all symbols from the stack

$q := \text{pop}()$

push (q)

$q := (q \xrightarrow{T})$

← follow the transition tagged with "T" from q

else

$q := (q \xrightarrow{z_i})$

← follow the transition tagged with "z_i" from q

$i := i + 1$

if all z symbols have been read then

accept Z

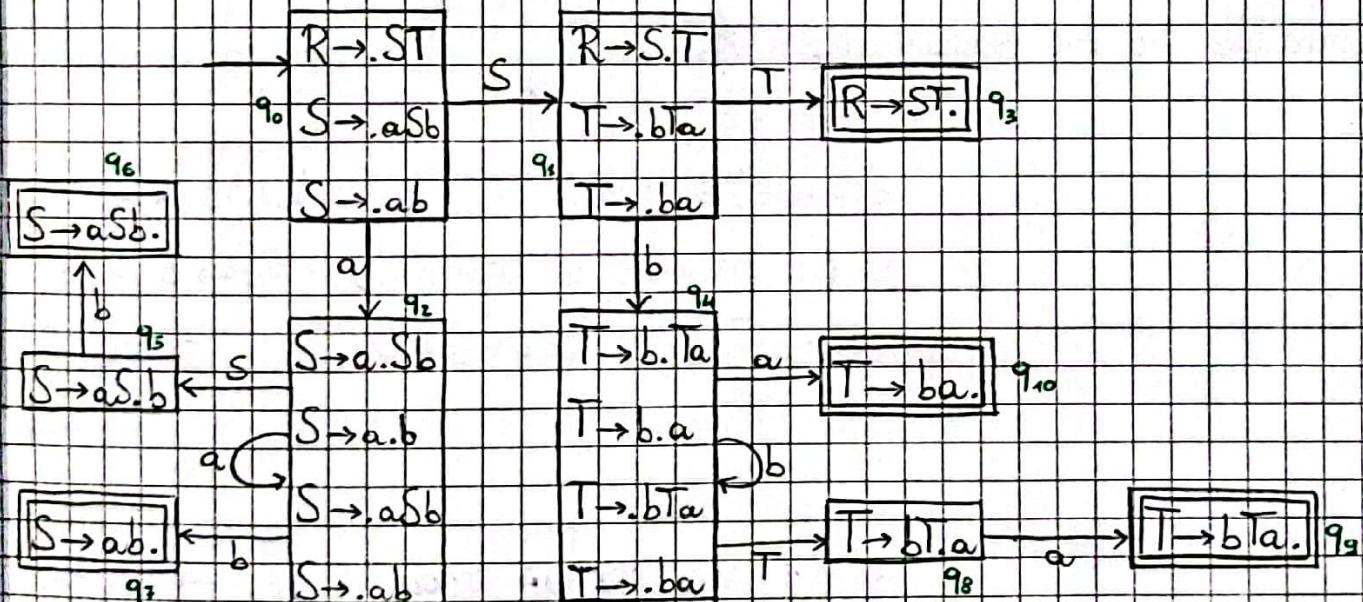
else

reject Z

Esempio:

$R \rightarrow ST$

DCFG: $S \rightarrow aSb \mid ab$
 $T \rightarrow bTa \mid ba$



P(Z) $Z = "aabbbbbbbaaaa"$

INPUT

a a b b b b a a a

q₀ q₀

q₂ q₂ q₂ q₂ q₂ q₁ q₁

q₂ q₂ q₅ q₅ q₄ q₄

q₇ q₆ q₄ q₄ q₄ q₄ q₄ q₈ q₈ q₈ q₉ q₉

q₇ q₆ q₄ q₄ q₄ q₄ q₄ q₈ q₈ q₈ q₉ q₉

STACK

la stringa
è finita:
accetto

NB: Il viceversa è falso:

~~$\forall \text{DPDA } P \exists \text{CFG } G \text{ t.c. } L(P) = L(G)$~~

Teorema:

Ogni DPDA che riconosce un linguaggio marcato ha una DCFG equivalente.

Idea della dim:

Riprendiamo la dimostrazione del seguente asserto:

$\forall \text{PDA } P \exists \text{CFG } G \text{ t.c. } L(P) = L(G)$

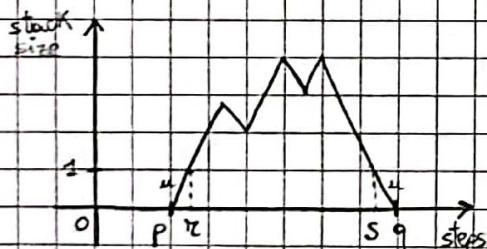
Le restrizioni che avevamo imposto su P erano:

- (1) Esiste un solo stato di accettazione.
- (2) Ogni passo o è una push, o una pop o la lettura di un simbolo dell'input.
- (3) Quando P accetta l'input, lo stack deve essere vuoto.

Per costruire la CFG equivalente con $V = \{A_{pq} \mid A_{pq} \text{ genera tutte le stringhe che portano } P \text{ dallo stato } p \text{ con stack vuoto allo stato } q \text{ con stack vuoto}\}$, ci era

vanno imbattuti in due scenari possibili:

CASO 1:



$$\delta(p, a, \epsilon) \ni (r, \mu)$$

SI PUÒ APPLICARE LA REGOLA

$$\forall a, b \in \Sigma$$

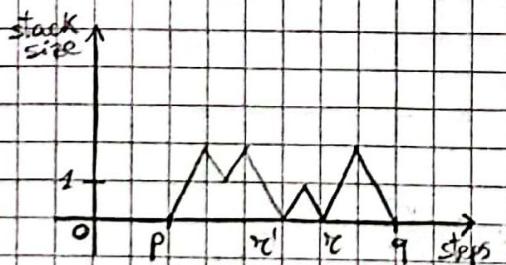
$$\delta(s, b, \mu) \ni (q, \epsilon)$$

$$A_{pq} \rightarrow a A_{rs} b$$

$$\forall p, q, r, s \in Q$$

$$\forall a \in \Sigma$$

CASO 2:



SI PUÒ APPLICARE LA REGOLA

$$\forall p, r, q \in Q$$

$$A_{pq} \rightarrow A_{pr} A_{rq}$$

Cioè che dobbiamo fare è adattare questa dimostrazione al DPDA DETERMINISTICO.

RESTRIZIONI DELL'AUTOMA:

Le restrizioni (1), (2) sono tranquillamente applicabili anche al caso deterministico. La restrizione (3) è problematica perché l'automa dovrebbe intendere non-deterministicamente qual è l'iterazione in cui l'input viene

consumato del tutto ed è necessario saltare allo stato di svuotamento dello stack. A risolvere tale non-determinismo è l'ipotesi del teorema per cui il DPDA riconosce un LINGUAGGIO MARCATO.

Passando alla costruzione della grammatica, un altro non-determinismo è insito nella regola $A_{pq} \rightarrow A_{pr} A_{rq}$ nel caso 2: se ci sono più istanti intermedi tra p e q in cui lo stack si svuota, l'automa dovrebbe decidere non-deterministicamente a quale di questi istanti dovrebbe ~~non~~ corrispondere lo stato r . Per ovviare a quest'altra faccenda, si può presumere che l'istante marcato con "r" sia sempre quello più vicino a q (e quindi quello più lontano da p).

Vediamo quindi come si costruiscono le regole della grammatica per abbattere la dimostrazione al caso deterministico:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_A\})$$

$$G = (V, \Sigma, R, S)$$

$$V = \{A_{pq} \mid p, q \in Q\}$$

$$S = A_{q_0} A_{q_A}$$

$$\forall p \in Q \quad A_{pp} \rightarrow \epsilon$$

$$\forall p, q, r, s, t \in Q \quad \forall u \in \Gamma$$

$$\delta(r, a, \epsilon) = (s, u)$$

"push(u)"

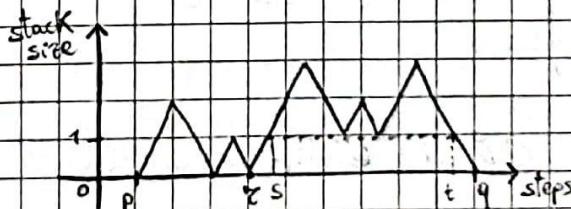
$$\begin{aligned} \forall a, b \in \Sigma_E, \\ \delta(t, b, u) = (q, \epsilon) \end{aligned}$$

"pop(u)"

SI PUÒ INTRODURRE LA REGOLA

$$A_{pq} \rightarrow A_{pr} a A_{rs} b$$

In questo modo r è sempre equivalente lo stato più vicino a q con lo stack vuoto
 \Rightarrow non c'è più il non-determinismo



Per completare formalmente la dimostrazione bisognerebbe provare che tutte e sole le regole introdotte costituiscono la grammatica equivalente all'automa di partenza (operazione del tutto analoga alla parte finale della dimostrazione del caso non-deterministico) e che questa grammatica che abbiamo creato è effettivamente deterministica col DK-TEST (passaggi che salteremo). \square

Definizione:

Un linguaggio A si dice "PREFIX-FREE" se $\forall w \in A \quad \forall z \neq \epsilon \quad wz \notin A$.

Lemma:

A è prefix-free.

Lemma:

Se G è una DCFG $\Rightarrow L(G)$ è prefix-free.

Dim:

Assumiamo che la variabile di partenza S della grammatica non appaia mai nella parte destra di una regola di G (altrimenti aggiungiamo una nuova variabile di partenza S_0 e una nuova regola $S_0 \rightarrow S$).

Per assurdo, supponiamo che $w \in L(G)$, $wz \in L(G)$, $|z| > 0$.

w è una stringa valida \Rightarrow ha handle focato ($h, T \rightarrow h$)

Riduciamo adesso le stringhe w , wz tenendo in considerazione l'handle f

zato: $w = xhy \rightarrow u \rightarrow^* S$

$wz = xhyz \rightarrow uz \rightarrow^* (Sz) \rightarrow$ NON PUÒ ESSERE RIDOTTA ULTERIORMENTE

ASSURDO poiché avevamo assunto che la variabile iniziale non apparisse mai nella parte destra di una regola di G .

Lemma:

La classe dei linguaggi generati dalle DCFG è la classe dei DCFL prefix-free.

NB: Non c'è un'effettiva equivalenza tra DCFG e DCFL: possiamo solo dire che l'insieme dei linguaggi generati dalle DCFG è strettamente contenuto nell'insieme dei DCFL.

Esempio:

$L = \{0^m 1^n \mid 0 < m < n\}$ è un DCFL ma \nexists DCFG G t.c. $L(G) = L$

Infatti L NON è prefix-free:

$0001 \in L$

$00011 \in L$

10/04/2020

LE GRAMMATICHE LR(K)

Definizione (Lookahead):

Supponiamo che $v = xyh$ sia una stringa valida e supponiamo che $(h, \rightarrow T \rightarrow h)$ sia un handle di v . Diciamo che $(h, \rightarrow T \rightarrow h)$ è "forzato dal lookahead K" se è l'unico handle di ogni stringa valida $xy\hat{y}$, dove $\hat{y} \in \Sigma^*$ e y, \hat{y} coincidono sui loro primi K simboli (se $|y| < K$ oppure $|\hat{y}| < K$ allora la stringa più corta tra le due deve essere il prefisso di quella più lunga).

Osservazione:

Handle forzato \equiv Lookahead 0

Definizione:

Una GRAMMATICA LR(K) è una CFG tale che l'handle di ogni stringa valida è forzato dal lookahead K.

$LR(0) \equiv DCFG$

$LR(K)$

Left-to-Right input processing \uparrow \uparrow Lookahead

Rightmost derivation \equiv Leftmost reduction

Se G è una $LR(K)$, $K > 0 \Rightarrow \exists$ DPDA P t.c. $L(P) = L(G) \Rightarrow L(G)$ è LR^m DCFL

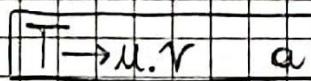
In particolare, $LR(0) \subsetneq LR(1) \equiv LR(2) \equiv \dots \equiv LR(K) \equiv DCFL$

Per semplicità, noi tratteremo ~~gli~~ le $LR(1)$.

Anch'esse hanno un loro DK_1 -TEST, che si può effettuare su particolari DFA DK_1 , i quali possono essere costruiti direttamente oppure tramite i NFA K_1 .

Vediamo le caratteristiche dell'automa K_1 :

\rightarrow STATO QUALSIASI



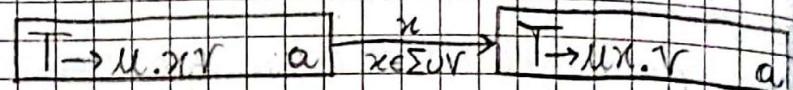
K_1 ha recentemente letto u , che dovrebbe essere una parte di un handle $(uv, T \rightarrow uv)$ purché v segua u , a segua v .

→ STATO INIZIALE

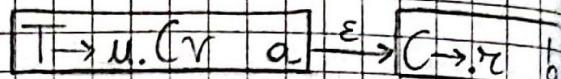
→ $S \rightarrow u \quad a$

$\forall S \rightarrow u, \forall a \in \Sigma, S$ variabile iniziale

→ TRANSIZIONE "SHIFT-MOVE"



→ TRANSIZIONE ϵ CHE VA IN UN'ALTRA REGOLA



b è il primo simbolo ~~di~~ di qualunque stringa che può essere derivata da r
OPPURE b = a se ~~di~~ r deriva ϵ .

Passando all'automa DK1, consideriamone un suo stato di accettazione:

$$\begin{array}{|c|} \hline R_1 \rightarrow u. & a_1 \\ \hline R_2 \rightarrow w.r & a_2 \\ \hline \end{array}$$

R₁ e R₂ sono CONSISTENTI se:

a) R₁ e R₂ sono entrambi complete e a₁ = a₂

OPPURE

b) R₂ non è completa e a₁ segue immediatamente il punto in R₂

Se DK1-TEST ha successo \Leftrightarrow ogni stato di accettazione NON contiene due dotted rule consistenti.

Esercizio:

Determinare se la seguente grammatica G è una LR(1) oppure no:

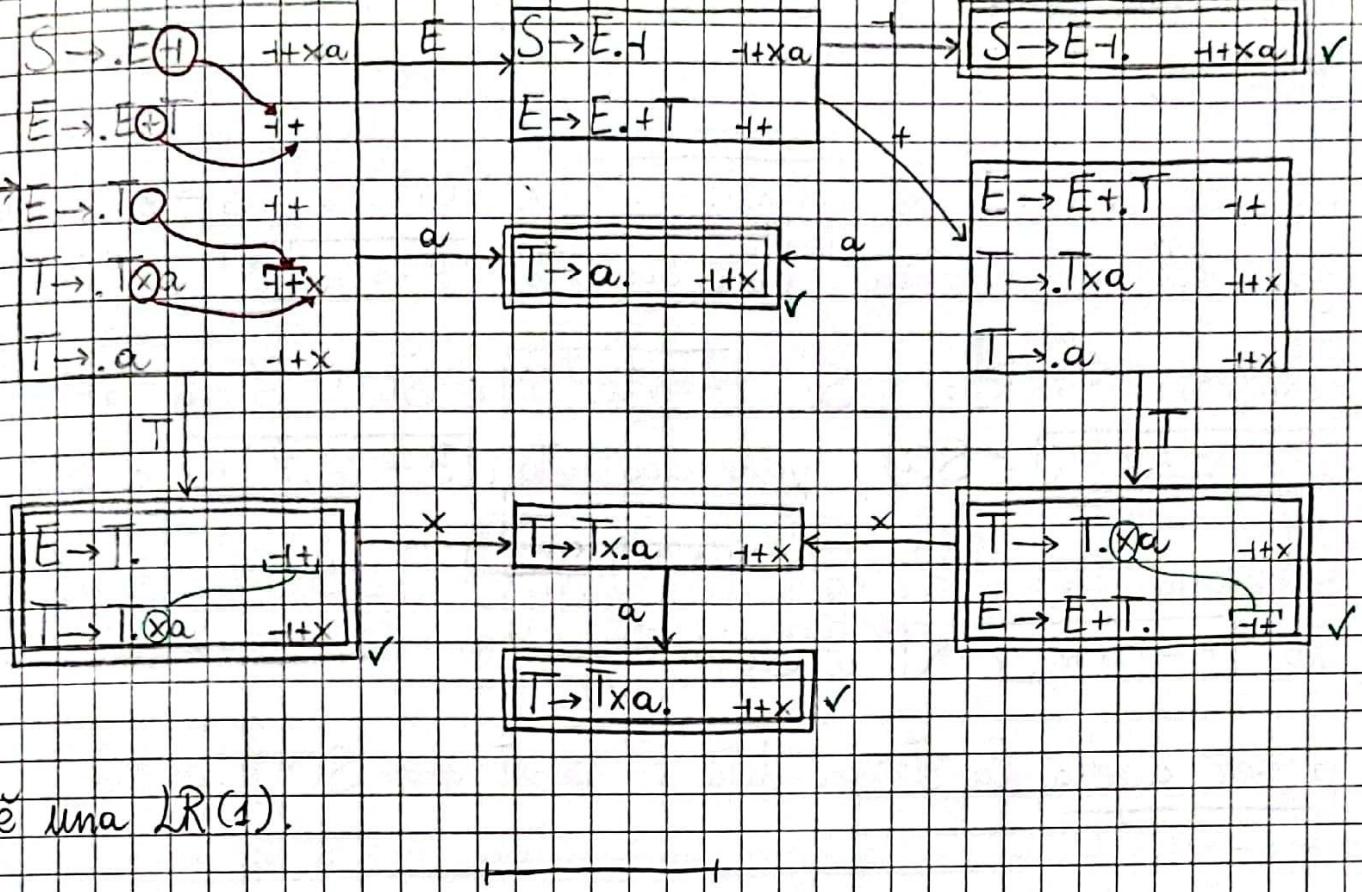
$$S \rightarrow E \dashv$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \alpha \mid a$$

Nella pagina seguente verrà riportato il DFA DK1 corrispondente alla grammatica G.

S → A	01
A → BB	01
B → CB	01
C → DC	01
D → 0	01
D → 1	01



G è una LR(1).

PARSER = algoritmo che prende in input una stringa e stabilisce se essa è generata da una data grammatica oppure no; inoltre è in grado di costruire un albero sintattico di derivazione di tale stringa.

Earley Parser:

È un algoritmo molto importante poiché funziona per tutte le grammatiche CFG.

COMPLESSITÀ TEMPORALE SU INPUT DI LUNGHEZZA n :

- $O(n^3)$ per grammatiche ambigue
- $O(n^2)$ per grammatiche NON ambigue
- $O(n)$ per DCFG
- $O(n)$ per LR(K)

$\Rightarrow \forall$ DCFL \exists grammatica G che è riconosciuta dall'Earley Parser in un tempo lineare.

FUNZIONAMENTO:

Dato una grammatica $G = (N, \Sigma, R, R_0)$ e una stringa $w = w_1 \dots w_n \in \Sigma^*$, l'Earley Parser costruisce una sequenza di sottinsiemi di stati: $S(0), S(1), \dots, S(n)$ (sottinsieme di fatted rule) secondo 4 regole (R_0, R_1, R_2, R_3):

→ INIZIALIZZAZIONE • R_0 : $S(0)$ include $R_0 \rightarrow u \mid 0$ per ogni regola $R_0 \rightarrow u$.

→ PREDISSIONE R_1 : Se $A \rightarrow u.Bv \mid i \in S(j)$ e $B \rightarrow z$ è una regola della grammatica, viene aggiunto $B \rightarrow z \mid j$ a $S(j)$.

→ COMPLETAMENTO R_2 : Se $A \rightarrow u. \mid i \in S(j)$ allora, per ogni $B \rightarrow v.Az \mid k \in S(i)$, viene aggiunto $B \rightarrow vA.z \mid k$ a $S(j)$.

→ SCANSIONE R_3 : Per ogni $A \rightarrow u.w_j v \mid k \in S(j-1)$ viene aggiunto $A \rightarrow uw_j.v \mid k$ a $S(j)$.

Il parser accetta la stringa data in input $\Leftrightarrow S(m)$ include $R_0 \rightarrow u \mid 0$.

Esercizio:

Verificare che la stringa "ataxa" appartiene al linguaggio $L(G)$, dove

$$E \rightarrow E + T \mid T$$

$$G : \quad T \rightarrow T \times F \mid F$$

$$F \rightarrow (E) \mid a$$

$S(0) : \textcircled{1} \quad \text{simbolo letto dall'input}$

$$\textcircled{1} \quad E \rightarrow E + T \mid 0 \quad R_0$$

$$\textcircled{4} \quad \overline{T} \rightarrow F \mid 0 \quad R_1 \textcircled{2}$$

$$\textcircled{2} \quad E \rightarrow .T \mid 0 \quad R_0$$

$$\textcircled{5} \quad F \rightarrow .(E) \mid 0 \quad R_1 \textcircled{4}$$

$$\textcircled{3} \quad \overline{T} \rightarrow .T \times F \mid 0 \quad R_1 \textcircled{2}$$

$$\textcircled{6} \quad \overline{T} \rightarrow .a \mid 0 \quad R_1 \textcircled{4}$$

$S(1) : a$

$$\textcircled{7} \quad F \rightarrow a. \mid 0 \quad R_3 \textcircled{6}$$

$$\textcircled{10} \quad \overline{T} \rightarrow T. \times F \mid 0 \quad R_2 \textcircled{8} \textcircled{3}$$

$$\textcircled{8} \quad \overline{T} \rightarrow F. \mid 0 \quad R_2 \textcircled{7} \textcircled{4}$$

$$\textcircled{11} \quad E \rightarrow E. + T \mid 0 \quad R_2 \textcircled{9} \textcircled{1}$$

$$\textcircled{9} \quad E \rightarrow T. \mid 0 \quad R_2 \textcircled{8} \textcircled{2}$$

$S(2) : +$

$$\textcircled{12} \quad E \rightarrow E. + T \mid 0 \quad R_3 \textcircled{11}$$

$$\textcircled{14} \quad \overline{T} \rightarrow F \mid 2 \quad R_1 \textcircled{12}$$

$$\textcircled{13} \quad \overline{T} \rightarrow .T \times F \mid 2 \quad R_2 \textcircled{12}$$

$$\textcircled{15} \quad F \rightarrow .(E) \mid 2 \quad R_1 \textcircled{14}$$

$$\textcircled{16} \quad F \rightarrow .a \mid 2 \quad R_1 \textcircled{14}$$

S(3): a

$$\boxed{17} \quad F \rightarrow a. \quad | \quad 2 \quad R_3 \quad \boxed{15}$$

$$\boxed{18} \quad T \rightarrow F. \quad | \quad 2 \quad R_2 \quad \boxed{17} \quad \boxed{19}$$

$$\boxed{19} \quad E \rightarrow E + T. \quad | \quad 0 \quad R_2 \quad \boxed{18} \quad \boxed{12}$$

$$\boxed{20} \quad T \rightarrow T. \times F \quad | \quad 2 \quad R_2 \quad \boxed{18} \quad \boxed{13}$$

$$\boxed{21} \quad E \rightarrow E. + T \quad | \quad 0 \quad R_2 \quad \boxed{19} \quad \boxed{1}$$

S(4): x

$$\boxed{22} \quad T \rightarrow T x. F \quad | \quad 2 \quad R_3 \quad \boxed{20}$$

$$\boxed{23} \quad F \rightarrow .(E) \quad | \quad 4 \quad R_2 \quad \boxed{22}$$

$$\boxed{24} \quad F \rightarrow a. \quad | \quad 4 \quad R_2 \quad \boxed{22}$$

S(5): a

$$\boxed{25} \quad F \rightarrow a. \quad | \quad 4 \quad R_3 \quad \boxed{24}$$

$$\boxed{26} \quad T \rightarrow T x. F. \quad | \quad 2 \quad R_2 \quad \boxed{25} \quad \boxed{22}$$

$$\boxed{27} \quad E \rightarrow E + T. \quad | \quad 0 \quad R_2 \quad \boxed{26} \quad \boxed{12}$$

$$\boxed{28} \quad T \rightarrow T. \times F \quad | \quad 2 \quad R_2 \quad \boxed{25} \quad \boxed{13}$$

$$\boxed{29} \quad E \rightarrow E. + T \quad | \quad 0 \quad R_2 \quad \boxed{27} \quad \boxed{1}$$

a+a+a ∈ L(G)

Come generare un albero sintattico dalla computazione dell'Earley Parser:

1) Costruire un albero tale che:

1.a) La sua radice è lo stato di accettazione di S(n).

1.b) Ciascun nodo ha come figli gli stati usati per aggiungerlo (il figlio sinistro è lo stato preceduto prima).

1.c) Le foglie sono delle rule senza variabili a sinistra del punto.

2) Potare l'albero rimuovendo i nodi i cui stati NON rappresentano regole complete.

3) Ri-disegnare l'albero scrivendo bene la regola in ogni nodo.

Per l'esempio precedente, la generazione dell'albero sintattico avviene così:

