CF: Linguaggio naturale/parsing/interprete

La grammatica G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

- insieme di regole di sostituzione (o produzioni)
- variabili: A, B
- terminali (simboli dell'alfabeto): $0, 1, \#$
- variabile iniziale: A

Definition

Una grammatica context-free è una quadrupla (V, Σ, R, S), dove

- V è un insieme finito di variabili
- Σ è un insieme finito di terminali disgiunto da V
- R è un insieme di regole, dove ogni regola è una variabile e una stringa di variabili e terminali
- $S \in V$ è la variabile iniziale

Una grammatica genera stringhe nel seguente modo:

1. Scrivi la variabile iniziale
2. Trova una variabile che è stata scritta e una regola che inizia con quella variabile. Sostituisci la variabile con il lato destro della regola
3. Ripeti 2 fino a quando non ci sono più variabili

Esempio per G_1:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000\#111$$

La sequenza di sostituzioni si chiama derivazione di $000\#111$

Molti linguaggi sono unione di linguaggi più semplici

- Se il linguaggio è regolare, possiamo trovare un DFA che lo riconosce

Esercizio 2

Definire una grammatica per il seguente linguaggio:

$L_1 = \{w \in \{0,1\}^* \mid w \text{ contiene un numero pari di 0 e un numero dispari di 1}\}$

$S_1 \to 1S_1 \mid S_11 \mid \varepsilon$
$S_2 \to 0S_2 \mid S_20$
$S_3 \to 1S_3 \mid S_31$
$S_4 \to 0S_4 \mid S_40 \mid \varepsilon$

Esercizio 3

Definire una grammatica per il linguaggio di tutte le espressioni regolari sull'alfabeto $\{0, 1\}$, cioè di tutte le espressioni costruite utilizzando

- le costanti di base: $\varepsilon, \emptyset, 0, 1$
- gli operatori: $+, \cdot, *$
- le parentesi

FACTOR → (EXPR) | Ø | ε | 0 | 1
(EXPR) → <FACTOR>
SUM → SUM + SUM | PRODUCT
PRODUCT → PRODUCT * PRODUCT | STAR
STAR → FACTOR* | FACTOR

Esercizio 4

Definire le grammatiche context-free che generano i seguenti linguaggi.
Salvo quando specificato diversamente, l'alfabeto è $\Sigma = \{0, 1\}$.

1. $\{w \mid w$ contiene almeno tre simboli uguali a $1\}$
2. $\{w \mid$ la lunghezza di w è dispari$\}$
3. $\{w \mid w = w^R$, cioè w è palindroma$\}$
4. $\{w \mid w$ contiene un numero maggiore di 0 che di $1\}$
5. Il complemento di $\{0^n 1^n \mid n \geq 0\}$
6. Sull'alfabeto $\Sigma = \{0, 1, \#\}$, $\{w\#x \mid w^R$ è una sottostringa di x e $w, x \in \{0, 1\}^*\}$

2.4 **(a)** $S \to R1R1R1R$
$R \to 0R \mid 1R \mid \varepsilon$

(d) $S \to 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$

Con le regole, riusciamo a costruire un albero sintattico (parsing tree):

- nodi interni sono variabili
- foglie sono variabili, simboli terminali o ϵ

Esistono delle grammatiche *ambigue*, dato che è possibile generare una stessa stringa *in modi diversi*. Occorre quindi definire un *ordine* di derivazione.

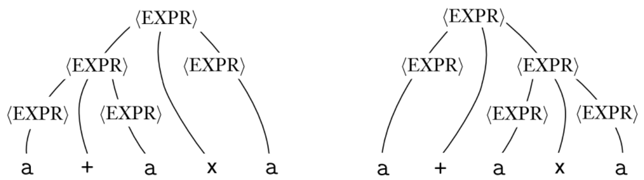
- Derivazione a sinistra (leftmost derivation): ad ogni passo, sostituisco la variabile che si trova più a sinistra.

- Una stringa w è derivata ambiguamente dalla grammatica G se esistono due o più alberi sintattici che la generano
- Equivalente: Una stringa w è derivata ambiguamente dalla grammatica G se esistono due o più derivazioni a sinistra che la generano
- Una grammatica è ambigua se genera almeno una stringa ambiguamente

Un albero fa vedere come si derivano le regole e in che ordine. Di fatto, si usa la variabile iniziale come albero. Generiamo da qui le foglie e le regole da sinistra verso destra, arrivando a terminali o ϵ

G_5
$\langle EXPR \rangle \to \langle EXPR \rangle + \langle EXPR \rangle \mid \langle EXPR \rangle \times \langle EXPR \rangle \mid (\langle EXPR \rangle) \mid a$

Questa grammatica genera la stringa a + a × a in due modi diversi!



Esercizio 1

Considerate la seguente grammatica G che definisce un frammento di un linguaggio di programmazione:

$$\begin{aligned}
\langle \text{STMT} \rangle &\to \langle \text{ASSIGN} \rangle \mid \langle \text{IF-THEN} \rangle \mid \langle \text{IF-THEN-ELSE} \rangle \\
\langle \text{IF-THEN} \rangle &\to \texttt{if condition then } \langle \text{STMT} \rangle \\
\langle \text{IF-THEN-ELSE} \rangle &\to \texttt{if condition then } \langle \text{STMT} \rangle \texttt{ else } \langle \text{STMT} \rangle \\
\langle \text{ASSIGN} \rangle &\to \texttt{a:=1}
\end{aligned}$$

Mostrate che G è ambigua

Esercizio 2

Modificate la grammatica G dell'esercizio precedente in modo da renderla non ambigua.

Esempio di grammatica ambigua (posso disambiguare semplicemente cambiando l'ordine di derivazione, chiamando prima "c"):

A → BC   A → BC → bC → bc
B → b   A → BC → Bc → bc
C → c

Possibili esempi:
S → If-then → If-cond-then-S → If-cond-then-If-then-else→If-cond-then-IF-cond-S-else-S

S → If-then-else → If-cond-then-S-else-S → If-cond-then→ If-cond-then-If-cond-else-S

Non è una parola in quanto dovrebbe essere terminale per essere considerata tale.
L'idea sintattica sarebbe:

IF(cond A){
if(cond B) statement else
statement
}

IF(cond A){
if(cond B) statement A
}
else statement B

Quindi avere un classico blocco if-then oppure avere due blocchi if-else con degli if appaiati (quindi innestati) al primo.
Per rimuovere l'ambiguità dobbiamo modificare le regole e quindi ne aggiungiamo due nuove:
Open, Closed
Open può essere con IF singolo, Closed non ha If oppure ogni If ha il suo else

L'idea è di avere degli "if" ambigui, che potrebbero collegarsi prima/dopo con altri if oppure con altri blocchi.
If ← If → else
If then S else S
If then If then else S
(qui posso aggiungere altri if, dato che questo if sarebbe interno e dunque non più ambiguo)

S → Open |Closed
Open → If-Cond-then-S|If-Cond-Then-Closed|else-Open   (qui sarebbe ambiguo perché non si saprebbe come interpretarlo)
Closed → A|If-Cond-Closed-else-Closed

Qui potrebbe rinascere l'ambiguità:

A → a:=1
S → Open→If-cond-then-S→If-cond-then-Closed
(l'idea è che Closed in sé contribuisce alla chiusura di tutti i costrutti).

È spesso conveniente avere le grammatiche in una forma semplificata. Una delle forme più semplici e utili è la Forma Normale di Chomsky.

Una grammatica context-free è in forma normale di Chomsky se ogni regola è della forma

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

dove a è un terminale, B, C non possono essere la variabile iniziale. Inoltre, ci può essere la regola $S \to \varepsilon$ per la variabile iniziale S

Theorem

Ogni linguaggio context-free è generato da una grammatica in forma normale di Chomsky

Idea: possiamo trasformare una grammatica G in forma normale di Chomsky:

1. aggiungiamo una nuova variabile iniziale $S_0 \to S$
2. eliminiamo le ε-regole $A \to \varepsilon$
3. eliminiamo le regole unitarie $A \to B$
4. trasformiamo le regole rimaste nella forma corretta

2 Eliminiamo le ε-regole $A \to \varepsilon$:

- se $A \to \varepsilon$ è una regola dove A non è la variabile iniziale
- per ogni regola del tipo $R \to uAv$, aggiungiamo la regola

$$R \to uv$$

- attenzione: nel caso di più occorrenze di A, consideriamo tutti i casi: per le regole come $R \to uAvAw$, aggiungiamo

$$R \to uvAw \mid uAvw \mid uvw$$

- se
- per

3 Eliminiamo le regole unitarie $A \to B$:

- se $A \to B$ è una regola unitaria
- per ogni regola del tipo $B \to u$, aggiungiamo la regola

$$A \to u$$

Problem 5 Convert the following CFG into Chomsky normal form.

$$\begin{aligned} A &\to BAB|B|\epsilon \\ B &\to 00|\epsilon \end{aligned}$$

1. First add a new start symbol S_0 and the rule $S_0 \to A$:

$$\begin{aligned} S_0 &\to A \\ A &\to BAB|B|\epsilon \\ B &\to 00|\epsilon \end{aligned}$$

2. Next eliminate the ϵ-rule $B \to \epsilon$, resulting in new rules corresponding to $A \to BAB$:

$$\begin{aligned} S_0 &\to A \\ A &\to BAB|BA|AB|A|B|\epsilon \\ B &\to 00 \end{aligned}$$

3. Now eliminate the redundant rule $A \to A$ and the ϵ-rule $A \to \epsilon$:

$$\begin{aligned} S_0 &\to A|\epsilon \\ A &\to BAB|BA|AB|BB|B \\ B &\to 00 \end{aligned}$$

4. Now remove the unit rule $A \to B$:

$$\begin{aligned} S_0 &\to A|\epsilon \\ A &\to BAB|BA|AB|BB|00 \\ B &\to 00 \end{aligned}$$

5. Then remove the unit rule $S_0 \to A$:

$$\begin{aligned} S_0 &\to BAB|BA|AB|BB|00|\epsilon \\ A &\to BAB|BA|AB|BB|00 \\ B &\to 00 \end{aligned}$$

6. Finally convert the 00 and BAB rules:

$$\begin{aligned} S_0 &\to BA_1|BA|AB|BB|N_0N_0|\epsilon \\ A &\to BA_1|BA|AB|BB|N_0N_0 \\ A_1 &\to AB \\ B &\to N_0N_0 \\ N_0 &\to 0 \end{aligned}$$

This grammar satisfies all the requirements for Chomsky Normal Form.

Consider the CFG:

$S \to \mathtt{a}X\mathtt{b}X$
$X \to \mathtt{a}Y \mid \mathtt{b}Y \mid \varepsilon$
$Y \to X \mid \mathtt{c}$

The variable X is nullable; and so therefore is Y. After elimination of ε, we obtain:

$S \to \mathtt{a}X\mathtt{b}X \mid \mathtt{ab}X \mid \mathtt{a}X\mathtt{b} \mid \mathtt{ab}$
$X \to \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b}$
$Y \to X \mid \mathtt{c}$

Example: Step 2

After elimination of the unit production $Y \to X$, we obtain:

$S \to \mathtt{a}X\mathtt{b}X \mid \mathtt{ab}X \mid \mathtt{a}X\mathtt{b} \mid \mathtt{ab}$
$X \to \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b}$
$Y \to \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$

Example: Steps 3 & 4

Now, break up the RHSs of S; and replace $\mathtt{a}$ by A, $\mathtt{b}$ by B and $\mathtt{c}$ by C wherever not units:

$S \to EF \mid AF \mid EB \mid AB$
$X \to AY \mid BY \mid \mathtt{a} \mid \mathtt{b}$
$Y \to AY \mid BY \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$
$E \to AX$
$F \to BX$
$A \to \mathtt{a}$
$B \to \mathtt{b}$
$C \to \mathtt{c}$

CFG/PDA/Chomsky

2. **(8 punti)** *Per ogni linguaggio L, sia* $\text{prefix}(L) = \{u \mid uv \in L$ *per qualche stringa* $v\}$. *Dimostra che se L è un linguaggio context-free, allora anche* $\text{prefix}(L)$ *è un linguaggio context-free.*

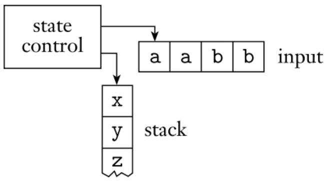
Se L è un linguaggio context-free, allora esiste una grammatica G *in forma normale di Chomski* che lo genera. Possiamo costruire una grammatica G' che genera il linguaggio $\text{prefix}(L)$ in questo modo:

- per ogni variabile V di G, G' contiene sia la variabile V che una nuova variabile V'. La variabile V' viene usata per generare i prefissi delle parole che sono generate da V;
- tutte le regole di G sono anche regole di G';
- per ogni variabile V di G, le regole $V' \to V$ e $V' \to \varepsilon$ appartengono a G;
- per ogni regola $V \to AB$ di G, le regole $V' \to AB'$ e $V' \to A'$ appartengono a G';
- se S è la variabile iniziale di G, allora S' è la variabile iniziale di G'.

Sarebbe possibile farlo anche in un altro modo; dobbiamo introdurre gli automi a pila per fare questo.

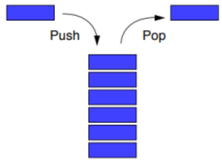
. agli Automi a Pila (PDA)

- Input: stringa di caratteri dell'alfabeto
- Memoria: stati + pila
- Funzione di transizione: dato lo stato corrente, un simbolo di input ed il simbolo in cima alla pila, stabilisce quali possono essere gli stati successivi e i simboli da scrivere sulla pila



La pila è un dispositivo di memoria last in, first out (LIFO):

- Push: scrivi un nuovo simbolo in cima alla pila e "spingi giù" gli altri
- Pop: leggi e rimuovi il simbolo in cima alla pila (top)



La pila permette di avere memoria infinita (ad accesso limitato)

Un automa a pila (o Pushdown Automata, PDA) è una sestupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

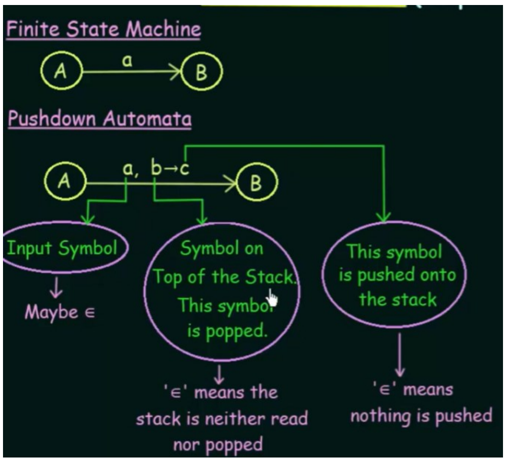
- Q è l'insieme finito di stati
- Σ è l'alfabeto di input
- Γ è l'alfabeto della pila
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \mapsto 2^{Q \times \Gamma_\varepsilon}$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati accettanti

(dove $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$)

Accettazione per pila vuota

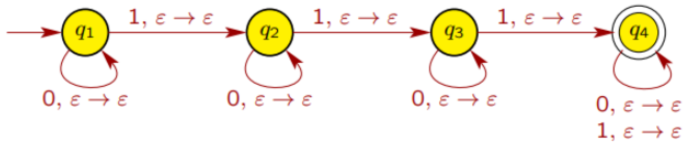
Un PDA accetta la parola w per pila vuota se esiste una computazione che

- consuma tutto l'input
- termina con la pila vuota ($s_m = \varepsilon$)



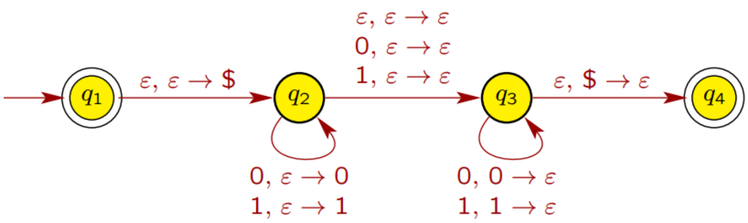
(a) $A = \{ w \in \{0,1\}^* \mid w \text{ contains at least three 1s} \}$

Answer:



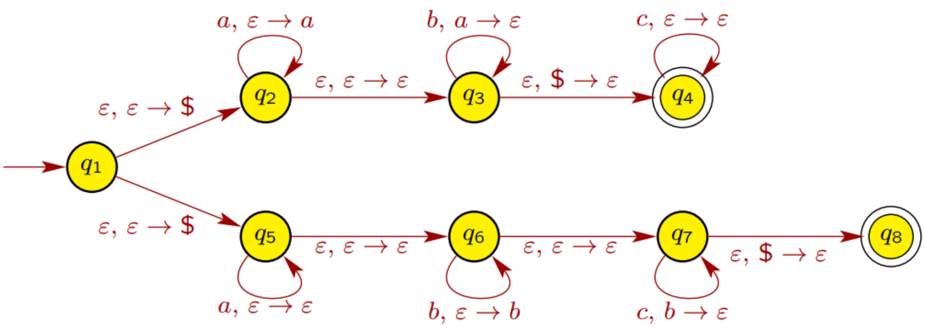
(c) $C = \{ w \in \{0,1\}^* \mid w = w^{\mathcal{R}} \}$

Answer:



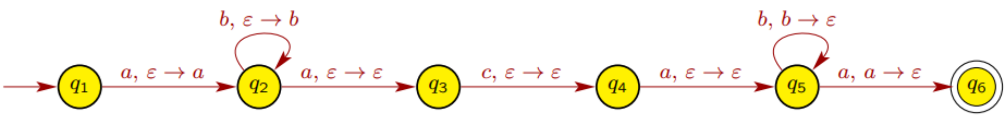
(d) $D = \{ a^i b^j c^k \mid i,j,k \geq 0, \text{ and } i = j \text{ or } j = k \}$

Answer:



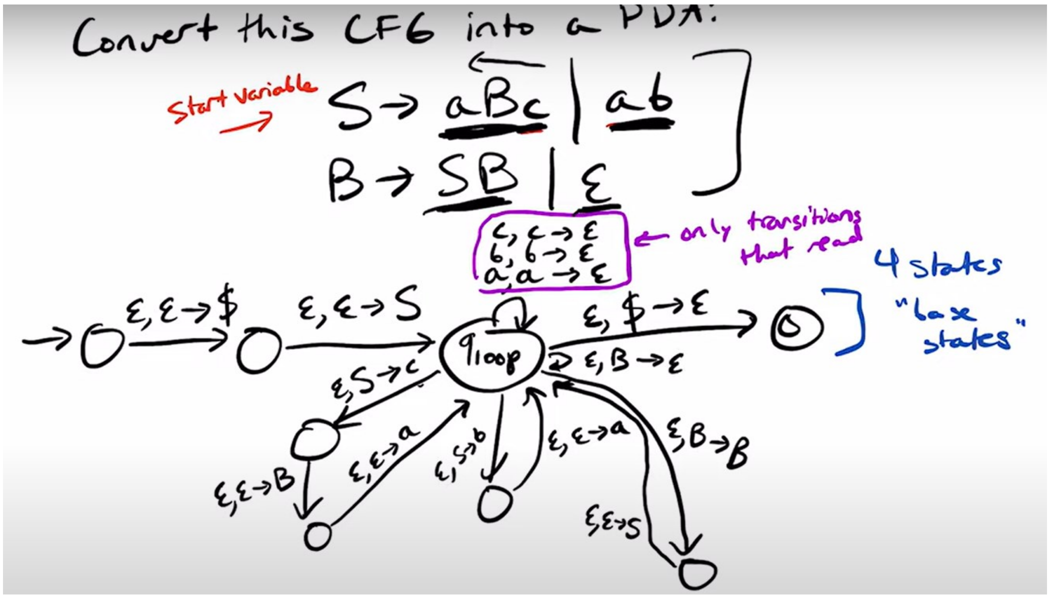
(h) $L = \{ ab^n acab^n a \mid n \geq 0 \}$.

Answer: A PDA M for L is as follows:



È possibile convertire PDA in CFG e viceversa:

- PDA simula i passi della grammatica
    - Stati iniziali
    - Regole
    - Simboli terminali
- Inserisce simbolo iniziale
- Sceglie regola e seleziona prossimo input (ogni simbolo aggiunge stati intermedi)
    - Se uguali = avanti
    - Se diversi = rifiuta



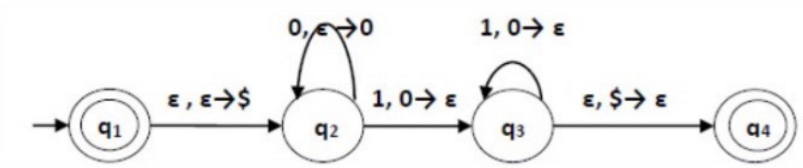
Lemma

Se un è riconosciuto da un PDA, allora è un linguaggio context-free

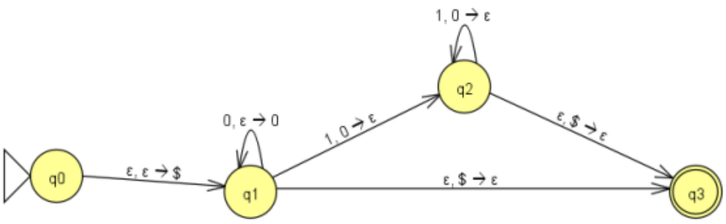
DI base:

- genera tutte le stringhe terminando
- unico stato accettante e svuota la pila prima di accettare
- ogni transizione fa push (inserisce unico simbolo = regola) e fa pop fa push (toglie unico simbolo = regola vuota = simbolo eliminato)
- si ottiene forma CF
    - regola iniziale
    - regole intermedie
    - simboli terminali

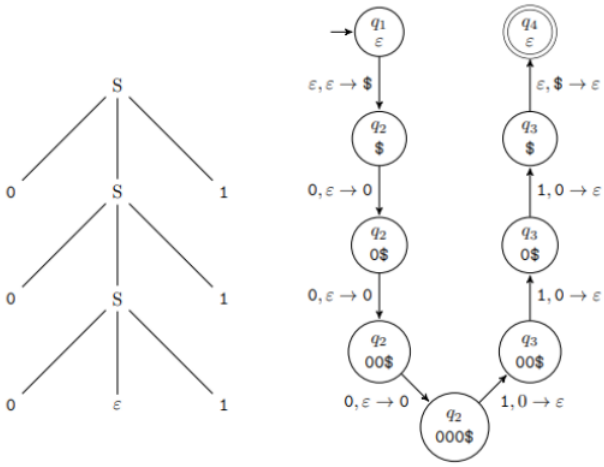
Trasformiamo il PDA per il linguaggio $\{0^n1^n \mid n>=0\}$ in grammatica:



In poche parole mette la transizione che da q2 va a q4 inserendo ε, $ → ε perché potrebbe non esserci nessun simbolo:



Quindi potremmo avere concretamente una situazione di questo tipo:

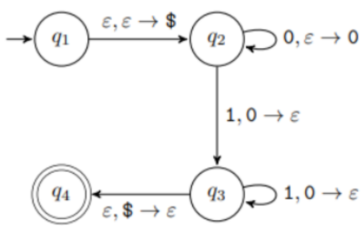


CFG derivation PDA run

La grammatica prodotta è (riporto l'automa identico al nostro, ma comunque per chiarire col discorso lettere a fianco della CFG):

- si parte considerando i 4 stati che vanno ad ε
- si inseriscono poi tutti gli stati che vanno, per combinazione, tutti gli uni con gli altri (letteralmente è una proprietà commutativa, l'immagine sotto chiarisce)
- nel caso di A_{23} abbiamo l'unione degli stati uscenti
- come ultimo abbiamo la regola dello stato finale

$A_{11} \to \varepsilon$
$A_{22} \to \varepsilon$
$A_{33} \to \varepsilon$
$A_{44} \to \varepsilon$
$A_{11} \to A_{11}A_{11} \mid A_{12}A_{21} \mid A_{13}A_{31} \mid A_{14}A_{41}$
$A_{12} \to A_{11}A_{12} \mid A_{12}A_{22} \mid A_{13}A_{32} \mid A_{14}A_{42}$
$A_{13} \to A_{11}A_{13} \mid A_{12}A_{23} \mid A_{13}A_{33} \mid A_{14}A_{43}$
$\dots$
$A_{42} \to A_{41}A_{12} \mid A_{42}A_{22} \mid A_{43}A_{32} \mid A_{44}A_{42}$
$A_{43} \to A_{41}A_{13} \mid A_{42}A_{23} \mid A_{43}A_{33} \mid A_{44}A_{43}$
$A_{44} \to A_{41}A_{14} \mid A_{42}A_{24} \mid A_{43}A_{34} \mid A_{44}A_{44}$
$A_{23} \to 0A_{22}1 \mid 0A_{23}1$
$A_{14} \to \varepsilon A_{23} \varepsilon$
$S \to A_{14}$



(Senza entrare nel dettaglio: accenni di come si ragiona)

2.25 For any language A, let SUFFIX(A) $= \{v|uv \in A$ for some string $u\}$. Show that the class of context-free languages is closed under the SUFFIX operation

Let A be a context-free language recognized by the PDA M_1. We are going to construct a PDA M recognizing the language SUFFIX(A) to show that the language SUFFIX(A) is also context-free.

Let M_2 be identical to M_1. We update the transitions of M_2 so that the transitions are performed without consuming any input symbol. That is, any transition of the form $a, b \to c$ is replaced with $\varepsilon, b \to c$. We add the transition $\varepsilon, \varepsilon \to \varepsilon$ from each state in M_2 to the corresponding state in M_1. M_1 and M_2 together form the PDA M. Start state of M is the start state of M_2.

When M starts reading a string v, it will construct the stack as it was reading u without reading u and without consuming any input symbol. This is possible since we updated all transitions in M_2 accordingly. Then at some point, by following the transition $\varepsilon, \varepsilon \to \varepsilon$, the computation continues in M_1 by reading the string v. So, the only accepted strings are those which are the suffixes of the strings in A.

We conclude that the set of context-free languages is closed under the SUFFIX operation.