Riassunto Traduttori 1

# Linguaggi formali

Un linguaggio formale (o rigoroso) è un linguaggio in cui vi è una semantica non ambigua, deve perciò essere comprensibile al compilatore.

Esempio: “La vecchia porta a casa del ragazzo cigola”.

Qui il soggetto è “la porta” e il verbo è “cigola”

Esempio 2:”La vecchia porta a casa del ragazzo un regalo”.

In questo caso “la vecchia è il soggetto e “porta” è il predicato.

Da notare come, data la similitudine delle due frasi, esse intendono due cose differenti, per questo utilizzare il linguaggio naturale coi compilatori è impossibile.

I linguaggi di per sé sono infiniti, per gestirli si trovano modi finiti per rappresentarli con precisione, dalla generazione al riconoscimento.

## Generazione

La generazione indica le regole che definiscono il “come” vengono generate tutte le frasi possibili in quel linguaggio e come sia possibile rappresentarle. Questi linguaggi appartengono a delle classi:

-Regolari: insieme di stringhe costituito da un alfabeto finito.

-Context free: generato da una grammatica non contestuale, ovvero le regole agiscono sui simboli non terminali indipendentemente dal contesto.

-Altri

Le classi di linguaggio definiscono concetti generali che vengono in seguito specificati nei singoli linguaggi.

## Riconoscimento

Il riconoscimento di un linguaggio definisce le regole su “come” vengono riconosciuti e a quale insieme appartengono dati linguaggi. Tutto questo avviene tramite l’ausilio di appositi algoritmi detti automi; per una determinata classe quegli stessi automi non possono riconoscere linguaggi di classi superiori.

Il punto di partenza che definisce tutte le caratteristiche di un linguaggio è detto Parsing o Analisi sintattica.

## Vocabolario comune

L'alfabeto di un linguaggio si indica con Σ, e all'interno delle graffe si indicano i caratteri. Esempio:

codice binario Σ {0,1}

numeri naturali Σ {0,1,...,9}

|...| indicano la cardinalità ovvero il numero di caratteri in un alfabeto/ stringa o il numero di stringhe in un alfabeto.

Esempio: Σ{1,2} , |Σ|=2

Una stringa è una sequenza di caratteri, un linguaggio invece è un insieme di queste ultime (in questo caso le stringhe vengono dette frasi).

### Operazioni tra stringhe/linguaggi

-) Concatenamento ( . ): indica l'unione di due stringhe o di tutte le combinazioni dei linguaggi;

Esempio:

S1 =”a”, S2 = “bc” S2.S1 = “bca”

Esempio:

L1 ={“a”,”b”}, L2 = {“c”, “d”} L1.L2 = {“ac”,”ad”,”bc”,”cd”}

NOTA: nella concatenazione tra linguaggi non vengono contati i duplicati, quindi la cardinalità totale potrebbe non essere il prodotto tra cardinalità.

Il concatenamento con una stringa nulla (ε) dà come risultato la stessa stringa (S.ε = S) essa tuttavia conta come frase nei linguaggi (se L = {ε}, |L| = 1).

* Riflessione: possibilità di capovolgere una stringa o tutte le frasi di un linguaggio (SR o LR).
* Potenza ennesima: concatenamento ennesimo della stringa o di tutte le frasi di un linguaggio

Esempio:

S=”ab” S3 = “ababab”

Esempio:

L = {“a”,”b”}

L3 = {“aaa”,“aab”,“aba”,“baa”,“abb”,“bab”,“bba”,“bbb”}

La potenza ennesima segue il seguente ragionamento:

* Prefisso/suffisso: parte iniziale/finale di una stringa.

Esempio: s= “abbb”

Prefissi (s) = {ε, “a”, ab, abb, abbb}

Suffissi (s) = {b, bb, bbb, abbb, ε}

* Annichilatore: riduzione del linguaggio, esso determina un elemento assorbente:

L.Φ = Φ con |Φ| = 0.

* Chiusura di Kleene: generazione della potenza ennesima partendo dalla stringa vuota (\*) o dal primo elemento del linguaggio(+). Esse sono descritte nel seguente modo:

L\*= U∞i=0 = L0 U L1 U … L+= U∞i=1 = L1 U L2 U …

La chiusura di Kleene ha la seguente proprietà:

(L\*)\* = L\* (L+)+ = L+

NOTA: I linguaggi possono anche utilizzare le operazioni insiemistiche.

### Espressioni regolari

Le espressioni regolari sono un linguaggio che permette di specificare altri linguaggi, fanno parte di questo gruppo i caratteri, ε, Φ, e le operazioni viste in precedenza.

Esempio: rappresentazione dei numeri interi

Σ(+,-,d) d=0,1,...,9

(+ U - U ε).(d).(d\*) oppure (+ U - U ε).d\*

L'unione determina un'alternativa, mentre il punto, come già detto, rappresenta la concatenazione. In questo caso vi può essere un + o un - all'inizio, seguito da almeno un numero da 0 a 9 rappresentando così tutti i numeri interi.

Per poter rappresentare anche lo 0: (+ U - U ε).(((0,1,...,9).(d\*)) U 0)

### Linguaggio denotato: implicazione(==>)

L'implicazione definisce 4 regole date n espressioni regolari(e):

-)(e1 U e2 U … U en) => ei con 1<= i <= n

-)e\* => e U e }k con k>=0

-)e+ => e U }k con k>0

-)en => e U e }n

Tutte le regole sono strettamente contenute una dentro l'altra, le uniche eccezioni sono la prima e la quarta le quali possono coincidere. Quando si opera su un percorso regolare, si sceglie un percorso che permette di operare su una parte più piccola del linguaggio .

### Linguaggio denotato: derivazione(-->)

Date due espressioni regolari L1 e L2, L1 deriva da L2 se L1 è uguale a ed L2 equivale a con che deriva da **:**

L1 ->L2 se [ L1 = ] con **->**

[ L2 = ]

La derivazione è ennesima (->n) quando svolta in N passi, è incerta(->\*) quando il numero di passi non è indicato, è sinistra (->s) quando viene completata partendo dal percorso regolare più a sinistra dell'espressione.

Esempio: (a)\*.(b U c).(d)+.(e uf)\*

Passo 1: aa.(b U c).(d)+.(e uf)\*

Passo 2:aac..(d)+.(e uf)\*

Passo 3:aacdddd.(e uf)\*

Passo 4:aacddddefeef

NOTA: (e U f) \* Viene derivato risolvendo prima la chiusura di Kleene e poi l'unione.

l'ordine di derivazione nelle espressioni non ha alcuna importanza.

il linguaggio denotato su un alfabeto è la formula sviluppata in tutti i modi possibili ,questa definizione vale per tutte le espressioni regolari.

I linguaggi regolari sono quindi linguaggi derivati da espressioni regolari, se il linguaggio è finito allora è un'unione finita di concatenamenti.

un linguaggio denotato si indica con L(e), quindi per ogni espressione regolare e:

* se e = Φ allora L (Φ) = Φ
* se e = ε allora L(ε) = L{ε}
* se e = a allora L(a) = L{a}
* se e = e1.e2 allora L(e) = L(e1).L(e2)
* se e = e1 U e2 allora L(e) = L(e1) U L(e2)
* se e = e\* allora L(e\*) = L(e\*)

### Proprietà di chiusura rispetto all’operatore

Data un'espressione regolare “e” o più stringhe che la rispettano, le operazioni su di esse appartengono alla stessa espressione regolare

Esercizio:

Somma/ differenza con numeri naturali e derivazione sinistra con 0 + 15 - 7

Soluzione:

Σ(+,-,0,1,…..,9) con d = (0 U (1 U … U 9) ( 0 U … U 9)\*)

d.((+ U -).d)\*

#### Derivazione :

0.((+ U -).d)\*

0.((+ U -).d).((+ U -).d)

0 + d.((+ U -).d)

0 + 15.((+ U -).d\*

0 + 15 - d

0 + 15 - 7

### Grammatiche context-free

Una grammatica è context free quando è una quadrupla <V, Σ, P, S>, in cui:

-)V è l'alfabeto non terminale, ovvero quello che serve per la struttura della frase;

-)Σ è il linguaggio terminale, quello che si utilizza per la costruzione effettiva di una frase:

-)P è l'insieme di regole di produzione, è generalmente indicato come P ⊆V x ( V u Σ) dove V x ( V u Σ) è la forma di frase, ogni regola è una corrispondenza tra un non terminale è un insieme misto di terminali e non.

NOTA: alla sinistra della forma di frase deve esserci uno e un solo operatore, ad esempio X -> a, X -> ab, oppure X -> z. [aX -> d] è sbagliato perché dipende dal terminale a.

-) S è uno start symbol, il punto di partenza della frase

Esempio: lista di palindromi

V: <l>, <pal>

Σ: {a,b,-}

P: <pal> <- a.<pal>.a

<pal> <- b.<pal>.b

<pal> <- ε

<l> <- <pal>.-.<l>

<l> <- ε

S: <l>

La scrittura di questi linguaggi avviene tramite BNF che, a differenza di quello sopra, utilizza “ ::= ” al posto di “<-” e “l” al posto di “U”

#### Derivazione (->)

Da due forme di frasi β e ૪ ∈ ( Σ u V)\* , β -> ૪ quando β = ղAδ , ૪ = ղ∝δ con ղδ ∈ (Σ u V)\* e (A-> ∝) ∈ P con A ∈ V.

Le derivazioni: Nesima, incerta e sinistra, valgono anche nella derivazione context free

### Linguaggio generato

Data una grammatica G e un terminale S, un linguaggio è detto generato da tale grammatica se è l'insieme di tutte le stringhe con caratteri terminali a partire da S.

L (G) = {x ∈ Σ\* \ S ->\* X}

La classe dei linguaggi context free é quindi la classe contenente tutti i linguaggi generati da grammatiche context free. una grammatica è ridotta quando non presenta cose che non hanno senso, queste ultime sono:

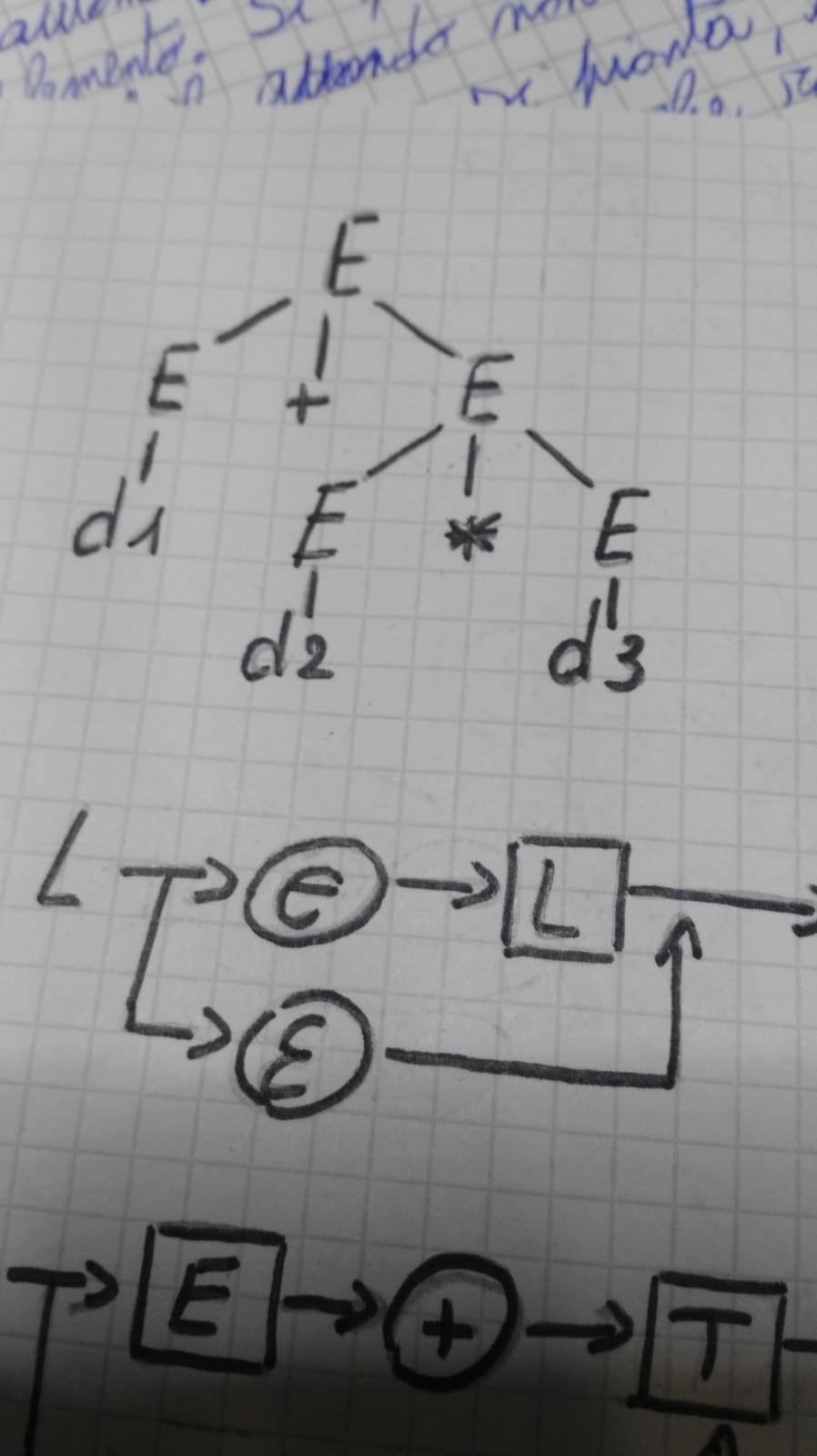
-) le frasi che in almeno una derivazione ritornano allo Start symbol (S->+ S);

-)Le frasi non raggiungibili da S, ovvero che non rispettano la seguente regola:

∀A, S->\*∝Aβ con ∝,β ∈ (V u Σ)

una regola è ricorsiva quando lo stesso terminale è presente sia a destra che a sinistra di “<- “ 0 “::=”, Una grammatica è ricorsiva quando presenta regole ricorsive o ci si arriva tramite n derivazioni.

### Albero di derivazione

Dato uno Start symbol E e la regola E -> E+E | E\*E | d, l’albero di derivazione è una rappresentazione ad albero delle derivazioni in cui la root è lo start symbol e andando avanto coi nodi si applicano le regole date, esso indica anche la struttura di una frase.

Esempio: d1 + d2 \* d3

E -> E+E

E-> d1+E

E-> d1+E\*E

E-> d1 + d2 \* E

E-> d1 + d2 \* d3

### Ambiguità nelle grammatiche e nei linguaggi

Data una grammatica è una qualunque frase generata da essa, questa si dice ambigua se esistono più alberi di derivazione per la stessa frase. un linguaggio è ambiguo quando ogni grammatica con cui lo costruisco è ambiguo:

Esempio:

L1 {a\*.bn.cn}

L2 {an.bn.c\*}

L1 U L2 permette la generazione di frasi da entrambi gli Start symbol, scegliendo l'alternativa che vuole a causa dell' intersezione. Le grammatiche equivalenti, quindi, sono un problema.

### Forme normali delle grammatiche

Le grammatiche context free possono essere ridotte in una forma normale, senza perdere il loro significato, esse sono:

-) la forma di Chomsky: un non terminale è derivabile con due non terminali oppure un terminale

A-> BC con A,B,C ∈ V

A-> a con a ∈ Σ u {ε}

-) la forma di Greyback: un terminale è derivabile con un terminale e un non terminale

A -> ∝a con ∝ ∈ V\* e a ∈ Σ

NOTA: la forma di Greyback non vale con la stringa vuota ;

le forme normali servono per confrontare le grammatiche tra di loro. Le forme normali servono per confrontare le grammatiche tra di loro.

### Dimostrazione

Per dimostrare che è possibile trasformare una grammatica in un'altra equivalente, si deve togliere la ricorsione sinistra immediata, dato che non è necessaria.

Esempio: A -> aA| b A-> ba\*

Quindi per generalizzare:

A->A **1|...|** A**k|** B1 | … | Bn con **1 ...k ∊** (V u Σ)\*  e B1 … Bn ∈ (V u Σ)\*  con il primo B diverso da A. B1 … Bn sono forme di frase in cui non sono presenti A a sinistra, cosa invece presente in A**1 ...k .** La formula senza ricorsioni a sinistra può essere in due modi:

-) si sostituiscono tutte le A con BA’ e A’ con ∝A’ oppure ε;

A -> B1A’ |...| BnA’

A’ -> 1 A’ |… |k A’ | ε

-) si sostituiscono tutte le A con B1 … Bn o con B1A’ … BnA’ e A’ con 1 A’ |… |k A’ oppure 1 |… |k

A -> B1 |...| Bn | B1A’ |...| BnA’

A’ -> 1 A’ |… |k A’ | 1 |… |k

### Linguaggi lineari

L (gram CF) ⊃ L (gram lin) ⊃ L (gram DX) ☰ L (gram strettamente DX) ☰ L(e.r.)

I linguaggi context free contengono i linguaggi lineari che a loro volta contengono i linguaggi destri ( o sinistri). Questi ultimi coincidono con i linguaggi strettamente destri e con quelli

denotati da espressioni regolari.Le grammatiche context free sono grammatiche nella forma A-> con A ∈ V e ∈ (V U Σ)\*. Una grammatica lineare è una grammatica context free nella forma A-> con A ∈ V e ∈ Σ\* ( X U ε)\* con X ∈ V.

I linguaggi lineari sono quindi linguaggi context free aventi un solo asse di crescita nell' albero di derivazione. La cosa è dimostrabile grazie al fatto che una grammatica lineare è context free per definizione. Per dimostrare l'inclusione stretta, bisogna fare un esempio:

L = { a1n b1n a2m b2m … akh bkh}

In questo esempio bisogna avere un albero di derivazione che oltre a produrre A1 deve produrre anche B1 e così via, quindi ho bisogno di più assi di crescita e di conseguenza non è lineare.

### Grammatiche lineari destre o sinistre

Una grammatica lineare destra o sinistra è una grammatica lineare in cui il non terminale si trova rispettivamente a destra o sinistra dei terminali.

A -> con ∝ ∈ (Σ\* ( X U ε)) per L DX

con ∝ ∈ (( X U ε) Σ\* ) per L SX

i linguaggi destri/ sinistri sono contenuti nei linguaggi lineari per definizione, per dimostrare l'inclusione stretta si utilizza il seguente esempio: L { an bn \ n >= 0 }

come faccio a dire che il numero di a deve essere uguale a quello di b? Ogni volta che produco a, produco anche b, questo è possibile solamente in questo modo: S-> aSb .

Da notare che alla destra/ sinistra del non terminale S vi sono dei terminali, quindi è in contrasto con la definizione di linguaggio destro/ sinistro e quindi è dimostrata l'inclusione stretta. La tecnica utilizzata nella dimostrazione è detta bilanciamento, essi sono importanti perché permettono di riconoscere una grammatica lineare da una destra/ sinistra.

### Grammatiche lineari strettamente destre/sinistre

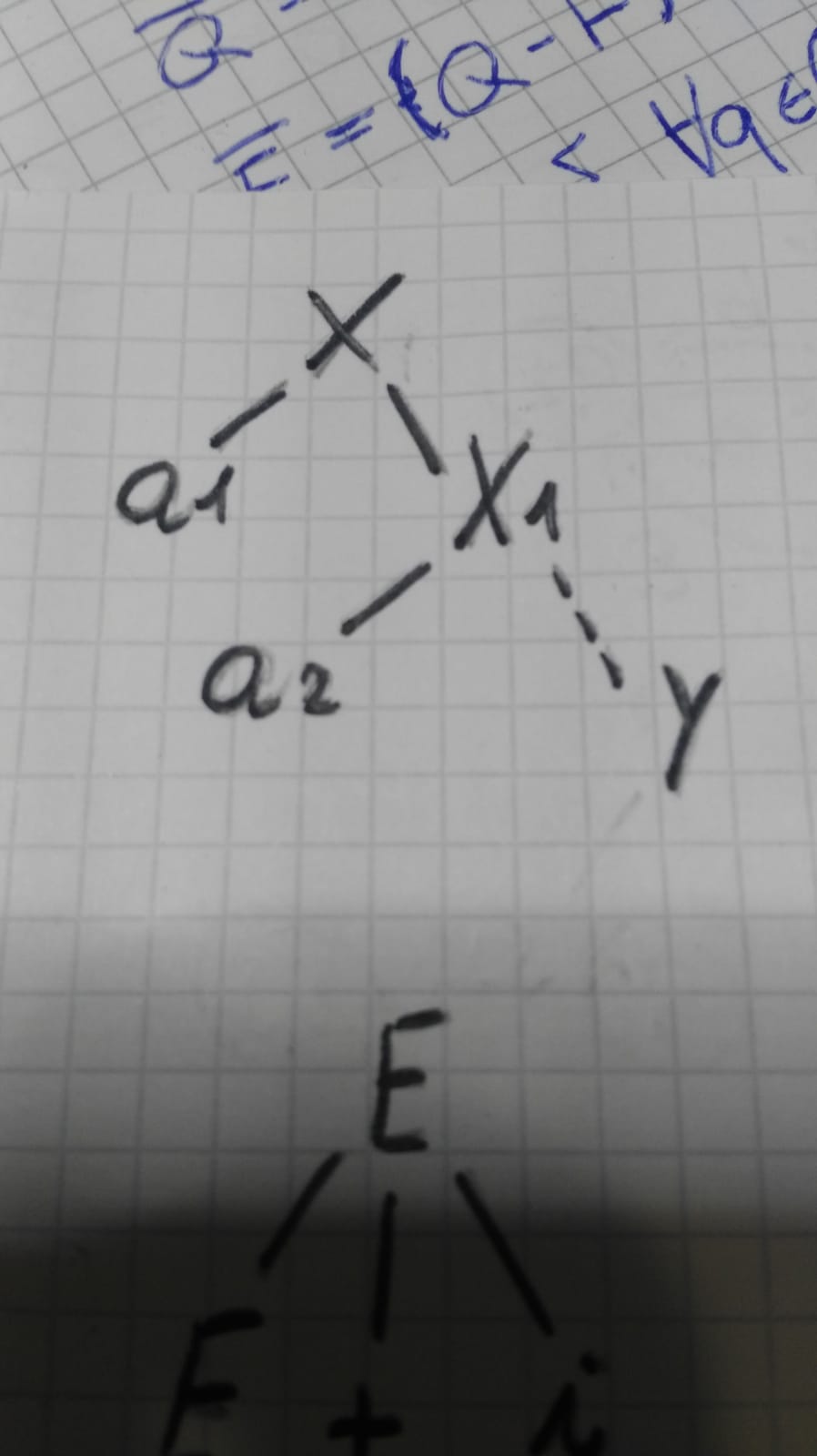
Una grammatica strettamente destra/ sinistra è una grammatica destra/ sinistra in cui vi sono un solo terminale e un solo non terminale (oppure niente).

A -> con ∝ ∈ (Σ u ε)( X u ε) per L SDX

con ∝ ∈ ( X u ε)(Σ u ε) per L SSX

la dimostrazione di contenimento avviene inserendo non terminali.

Esempio:

X -> a1 … ak Y 

X1 -> a1 X1,

X2 -> a2 X2 ,

X3 -> a3 X3 ,

Xk -> ak Y

### Grammatiche regolari

Una grammatica regolare è una grammatica destra/ sinistra denotato da un'espressione regolare, la dimostrazione avviene attraverso più passaggi; dati X -> **1|...|k:**

1)Si trasformano tutte le ” | ” in “ U ”;

Esempio: X-> abX | bX | cA | fB | g | hd

X = abX U bX U cA U fB U g U hd

2)Si raccoglie a fattor comune (se possibile);

X = X (ab U b) U cA U fB U g U hd

3) Eliminazione delle reazioni dirette scrivendole dalla forma “uXxv” a “u\*v”;

X = (ab U b)\* ( cA U fB U g U hd)

4) Risoluzione mediante Gauss, ovvero si sostituiscono tutti gli altri non terminali (se possibile)

5) se vi sono altre ricorsioni dirette, estendere l'espressione e ripartire dal punto 2. Oltre alla dimostrazione vi è anche una regola generale: una grammatica è regolare quando l'alfabeto è unitario ovvero è formato da una sola lettera, di conseguenza non vi sono bilanciamenti e quindi si può anche ricavare la grammatica destra/sinistra.

Esempio: S-> aSa | ε

L'albero di derivazione ha un asse centrale che termina con la stringa vuota, la sua espressione regolare è quindi (aa)\* e la sua grammatica destra è S -> aaS | ε

#### Osservazioni

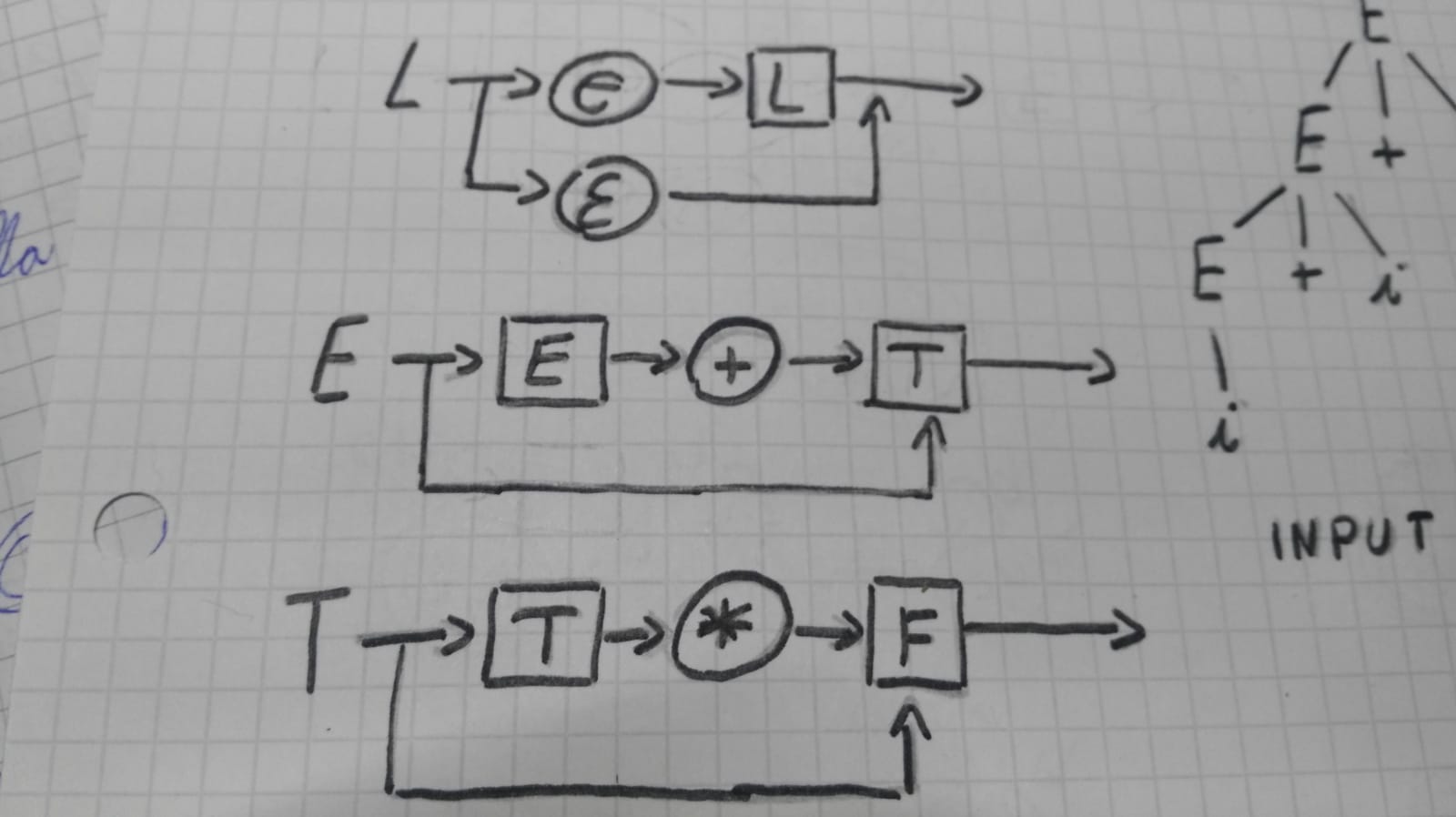
Le forme di bilanciamento si traducono come ricorsioni auto inclusive, ovvero che ci sono terminali sia a destra che a sinistra del non terminale. Il salto da context-free a regolare avviene” a livelli”: da context-free a lineare utilizzando un solo asse di crescita e da lineare a destro/ sinistro quei bilanciamenti.

E’ possibile utilizzare anche notazioni più compatte per la rappresentazione delle grammatiche, quali formule ristrette o grafi. In quest’ultimo caso, i caratteri terminali sono rappresentati tramite cerchi mentre i quadrati lo fanno coi non terminali.

Esempio: L → eL|𝜺 = e\*

Esempio: E→ E+T|T

T→ T\*F|F



### Passaggio da un linguaggio a una grammatica

Consideriamo una grammatica che implementa una lista di elementi e:

L→ ε|eL oppure L→ ε|Le

La differenza principale tra le due grammatiche è l’ordine in cui vengono generati gli elementi e, nella prima infatti viene prima generato “e” e poi si va in profondità mentre nella seconda avviene il contrario, il risultato in questo caso non cambia. Qui vale il discorso di composizionalità: non ho bisogno di ridefinire tutta la lista, basta solo ridefinire l’elemento e, infatti nel caso di una lista di numeri binari:

L → ε|eL con e-->0|1 oppure L-->ε|0L|1L

La prima grammatica è più astratta rispetto alla seconda, quindi la si preferisce a quest’ultima.

Diverso è il caso in cui vi sono liste con separatori, la grammatica L-->ε|e,L infatti può fare danno dato che è possibile avere altri elemento oltre l’ultima virgola, quindi si fa un ragionamento più complesso:

L→ e|e,L oppure L→ e|L,e

L’implementazione della stringa vuota si ragiona in un modo più complesso, ovvero aggiungendo un livello in più alla grammatica:

L→ ε|N N→ e|e,N

#### Gestione delle priorità degli operatori

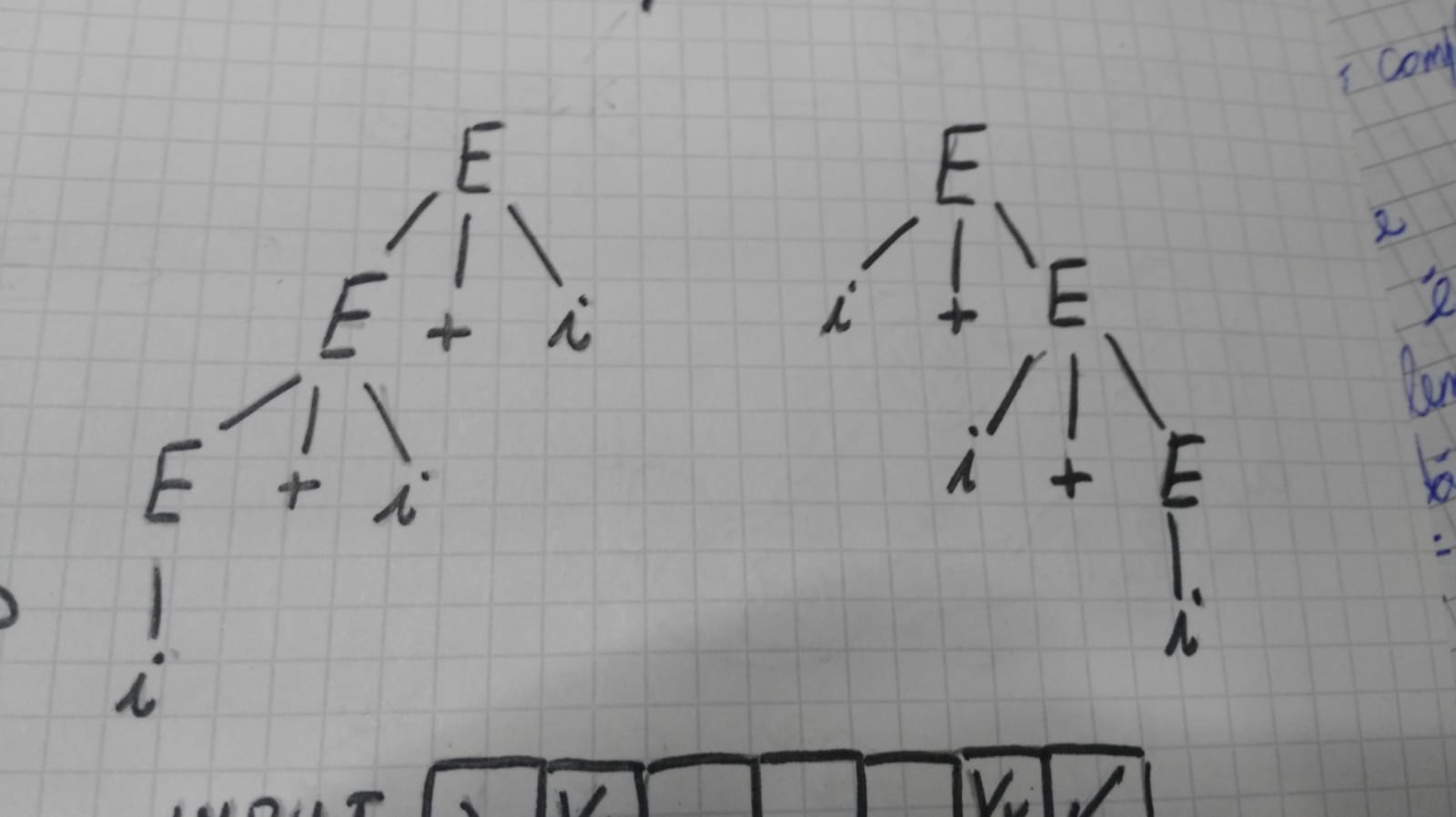
Consideriamo il seguente esempio:

E→ E + E|E \* E|d dove d è un numero tra 0 e 9

Questo esempio può essere visto come una lista di separatori a livello di lettura, tuttavia non è così quando i separatori si riferiscono a operatori dato che in tal caso entra in gioco la priorità degli stessi. Quindi bisogna sapere quando la precedenza degli operatori e la loro associatività, ad esempio a + b + c può essere letto come (a + b) + c oppure a + (b + c). I linguaggi degli operatori utilizzano quindi le parentesi per identificare quale operazione ha priorità e quale meno. Una soluzione semplice è quindi l’utilizzo delle parentesi, tuttavia esse ci costringono a utilizzare un linguaggio più complesso, quindi si deve imporre tale operazione mettendola più in profondità, ad esempio:

E→ E+i|i

E→ i+E|i



la differenza principale tra le due grammatiche sta nell’espansione dell’albero, infatti se voglio sviluppare il ramo a sinistra, basta mettere una ricorsione appunto a sinistra. Un ragionamento simile è quindi applicabile anche con gli operatori, tenendo conto della loro priorità, come nell’esempio: se voglio generare x + y \* z

E→ E + T|T

T→ T \* F|F

F→ i |(E)

In questo modo si evitano le ambiguità e si dà priorità alla moltiplicazione dato che è più in profondità. Nel precedente esempio è possibile aggiungere la sottrazione e la divisione, quindi dato che hanno rispettivamente le stesse priorità dell’addizione e della moltiplicazione, si agisce nel seguente modo:

E→ E + T|E - T|T

T→ T \* F|T \ F|F

F→ i |(E)

Supponiamo ora di voler aggiungere la sommatoria, essa è una somma n-aria e quindi si può rappresentare come una lista, tuttavia c’è bisogno di una sentinella per indicarne la fine così da evitare ambiguità, quindi:

E→ E + T|E - T|T

T→ T \* F|T \ F|F

F → -G|G

G→ L |H

H→ i,(E)

L→ E,L|E

Utilizzando i due per indicare l’inizio e la fine della lista è possibile implementare queste operazioni nelle grammatiche, com’è possibile notare dall’esempio questi operatori devono essere parentesizzati, dal momento che non si sa da quanti operatori è composto.

Il metodo visto in precedenza è detto notazione infissa, tuttavia vi è un’altro metodo per rappresentare le priorità tra operatori, la notazione postfissa. Essa permette di evitare l’utilizzo di ulteriori caratteri non terminali, Ogni operatori infatti prende i due terminali antecedenti a esso, ad esempio:

a+b\*c si rappresenta come abc\*+, il “\*” forma (b\*c) mentre il “+” prende quest’ultimo e la “a” restante per completare la frase.

Questa notazione tuttavia non funziona con gli operatori variadici e in più possono esserci ambiguità tra la sottrazione e la negazione. Per quest’ultimo caso una possibile soluzione è il cambio di uno degli operatori.

## Linguaggi context-sensitive

I linguaggi context-sensitive sono linguaggi che includono i context-free e, come dice il nome, sono dipendenti da un contesto, per esempio:

aA→ X1….Xn

bA→ Y1….Ym

A seconda del terminale che precede A, si sceglie la regola da utilizzare, un esempio di linguaggio context-sensitive è la lingua italiana, in essa infatti non posso mettere nomi plurali quando l’articolo è singolare e viceversa, un discorso simile vale anche per i tempi verbali. La gestione delle grammatiche context-sensitive è complicata e richiede molta potenza di calcolo, per gestirle infatti si utilizzano apposite tecniche. I linguaggi di programmazione fanno parte di questo insieme, essi però non hanno bisogno di tutte queste tecniche dato che bisogna solo gestire funzioni e variabili creati da esso. Negli esempi precedenti la parte context-free viene gestita ad-hoc attraverso una tabella di simboli, essa rappresenta un modo efficiente per gestire questo tipo di linguaggi.

Esempio: generazione di un linguaggio con la grammatica S→ YcY con Y appartenente ai terminali (c escluso). in questo esempio devono essere due gruppi di caratteri identici separati da c.

| Si hanno le seguenti regole:  S→ X↙ //genero la stringa X  X→ aXA // genero palindrome in cui A eB sono prestanomi di a e b  X→ bXB  XA→ XA’ //cambio gli oggetti A e B vicino a X in A’ e B’  XB→ XB’  A’A→ AA’//li sposto a destra  B’B→ BB’  A’B→ BA’  B’A→ AB’  A’↙ → a //sostituisco  B’↙ → b  A’a→ aa  B’b→ bb  A’b→ ab  B’a→ ba  Xa→ ca  Xb→ cb | Esempio: abbcabb  S→ X  X → aXA  X→ abXBA  X→ abbXBBA  X→ abbXB’BA  X→ abbXBB’A  X→ abbXB’AB’  X→ abbXAB’b  X→ abbXA’bb  X→ abbXabb  X→ abbcabb  La derivazione delle frasi dipende dall’ordine, di conseguenza bisogna permutare tutti gli ordini possibili. |
| --- | --- |

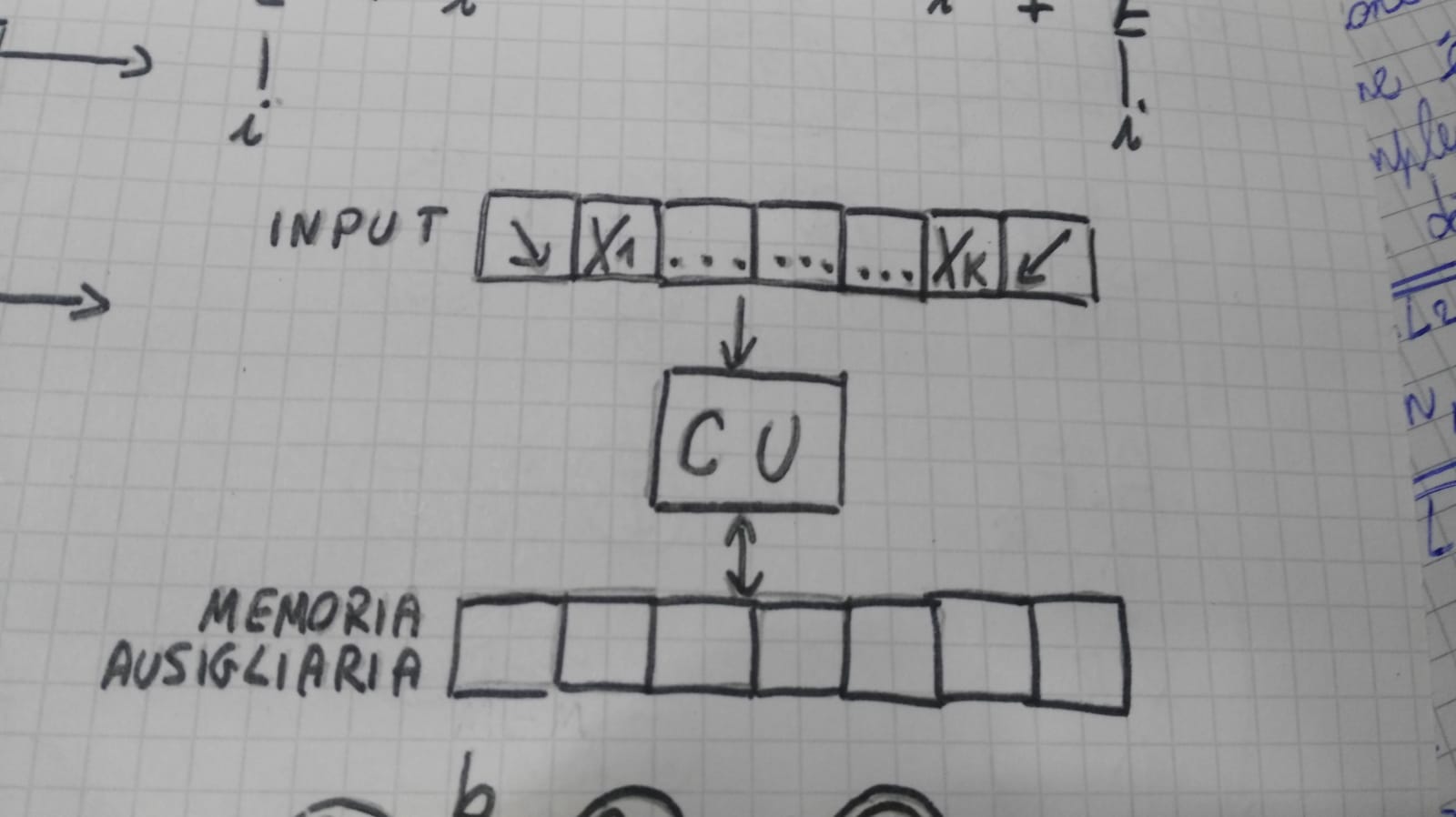
Come già detto in precedenza, gestire le cose a questo livello richiede molta potenza di calcolo, per questo motivo si utilizzano i linguaggi context-free con l’utilizzo di qualche trucchetto. I vari linguaggio sono divisi in classi in base alla grammatica che li genera e all’automa che li riconosce.

| Classe | Regole | Classe del linguaggio generato | Classe dell’automa riconoscitore |
| --- | --- | --- | --- |
| Tipo 3  (Pattern matching) | lineare destra  A-->xB con X appartenente a ∑\* | Linguaggi regolari | automa a stati finiti |
| Tipo 2  (Linguaggi di programmazione | context-free  A → α  con α ∈ (∑ ∪ V)\* | Linguaggi context-free | automa a pila non deterministico |
| Tipo 1 | Context-sensitive  β → α  α, β ∈(∑ ∪ V) \*  |β| ≤ |α| | Linguaggi context-sensitive | macchina di Turing con nastro proporzionale all’input |
| Tipo 0  (Silvally) | Senza restrizioni | Linguaggi ricorsivamente enumerabili | Macchina di Turing |

# Riconoscimento dei linguaggi

Oltre alla generazione dei linguaggi (vista in precedenza), i è anche il loro riconoscimento, esso è possibile grazie ai cosiddetti automi che, dato in input una frase, esso indica se appartiene o meno a un dato linguaggio restituendo un booleano o valori più complessi. In questo contesto vengono utilizzati gli automi al posto degli algoritmi dato che sono standardizzabili e più semplici da dimostrare.

## Introduzione generale alla macchina di Turing

Per capire il concetto di automa e di riconoscimento di un linguaggio, introduciamo la macchina di Turing: essa è una macchina che computa tutto il computabile ed è formata un’unità di controllo in cui vi è un programma codificato in modo semplice (può essere una tabella che indica cosa fare in una data situazione), l’input in cui vi è una testina che si muove a piacere e la memoria ausiliaria, considerata infinita. La macchina di Turing si trova in un certo stato indicato dall’unità di controllo e a seconda di esso opera in modo differente: da un dato stato e un dato input sotto la testina è possibile finire in un altro stato, scrive in memoria oppure muovere la testina a destra o a sinistra.

q, a → q', a', s/d

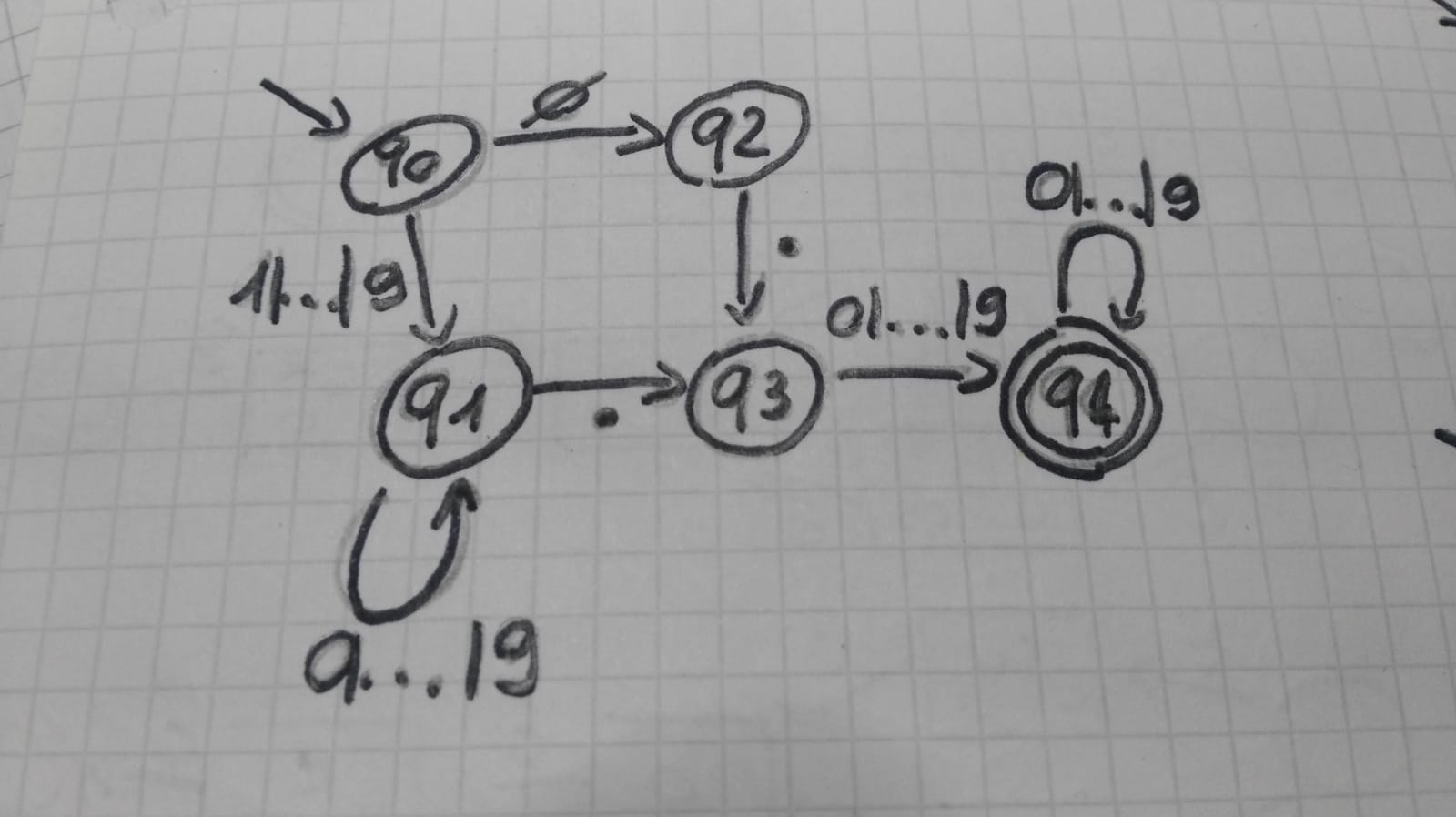
L’espressione sopra è una transizione e può essere rappresentata con una tabella le cui righe sono gli stati e le colonne i valori della testina dell’input, le celle della tabella indicano quindi il risultato della cosiddetta funzione di transizione δ, essa è deterministica quando vi è un solo valore e non deterministica quando ce ne sono molteplici, una cella vuota indica invece un fail. Una transizione deve avere una configurazione iniziale e una finale, le mosse sono operazioni molto piccole che permettono il passaggio da una configurazione a un’altra. Una configurazione è detta finale quando l’input è finito e la macchina si trova in una configurazione definita come tale, essa contiene lo stato e i vari input.

## Fasi di riconoscimento

Esempio: dato in input abcab bisogna indicare se esso è contenuto nella grammatica YcY, dove c è la marca di centro. Consideriamo che l’input ricevuto venga scritto nella memoria ausiliaria:

1. Memorizzo a nella tabella dettata in precedenza e lo sovrascrivo, si ha quindi “Nbcab”;
2. La testina va avanti fino a quando non trova la marca di centro c, quindi va avanti e controlla se il carattere a cui punta la testina è uguale a quello memorizzato, se è così lo sovrascrive e torna all’inizio (si ha quindi “NbcNb”), altrimenti va in fail;
3. Esegue i primi due punti fino a quando la prima metà non viene totalmente sovrascritta;
4. Se non ci sono più caratteri da esaminare dopo c, l’automa termina con successo, altrimenti va in fail.

## Automa a stati finiti

L’automa a stati finiti è un caso particolare della macchina di Turing in cui non vi è la memoria ausiliaria e l’input deve essere letto da sinistra verso destra, esso serve per il riconoscimento dei linguaggi regolari. La rappresentazione più comune è quella a grafo in cui i nodi rappresentano gli stati e gli archi indicano i valori della testina dell’input. Uno stato particolare, rappresentato tramite due cerchi concentrici, indica lo stato finale, in questo tipo di automa possono essere presenti molteplici stati di questo tipo. Le celle della tabella delle transizioni rappresentano quindi gli stati di arrivo dato quello iniziale e il valore della testina dell’input. 

Esempio: riconoscimento dei numeri con floating point

Il numero 102.01 fa parte del linguaggio? sì perché:

q0,1 → q1

q1,0→ q1

q1,2 → q1

q1,. → q3

q3,0 → q4

q4,1 → q4

A questo punto l’automa controlla se si trova in uno stato finale o meno, in tal caso è così e quindi il numero appartiene a tale linguaggio.

Nel caso di 102. la frase finisce in q3 che non è uno stato finale, quindi va in fail.

L’automa a stati finiti è definito da 5 oggetti:

* L’insieme degli stati Q;
* L’alfabeto ∑;
* La funzione di transizione ઠ la cui tipatura è ઠ:Qx∑→ Q per gli automi deterministici, essi vanno in un’altro stato oppure falliscono;
* L’insieme degli stati finali F contenuto in Q;
* lo stato iniziale q0;

Per definire un linguaggio accettato dall’automa, serve la nozione di stringa attraverso ઠ\* ovvero una funzione di transizione che opera su di esse: ઠ\*: Q,∑\* → Q

la funzione ઠ\* é dimostrabile per induzione, infatti data una stringa Y, un carattere a e uno stato q, si ha che:

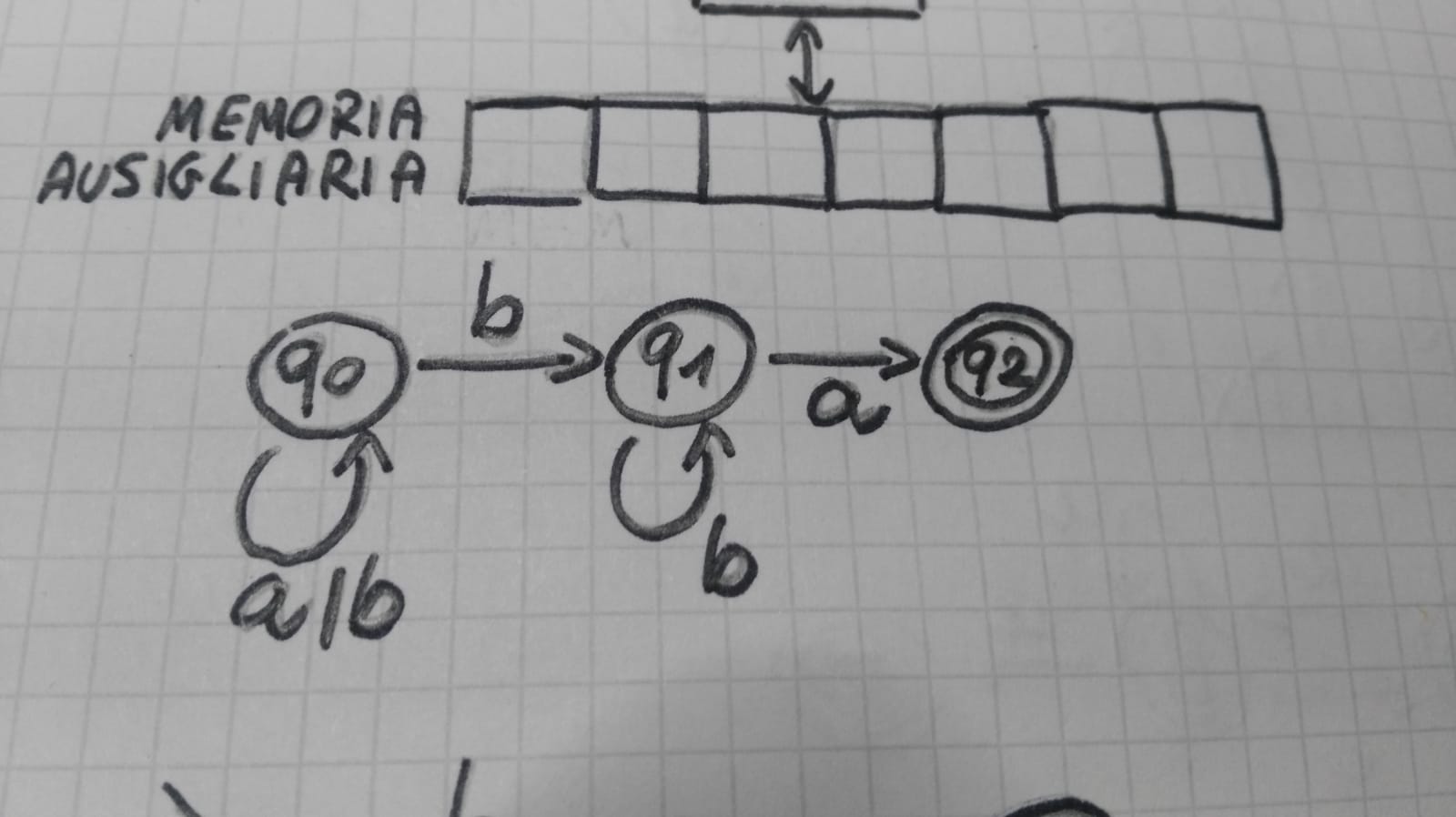
* ઠ\*(q,a)=ઠ(q,a) ovvero il caso base dell’ipotesi induttiva;
* ઠ\*(q,Y\*a)=ઠ\*(ઠ\*(q,Y),a)=ઠ(ઠ\*(q,Y),a)

Quindi, dato un automa a stati finiti M, il linguaggio accettato da M è l’insieme di stringhe in cui a partire da uno stato iniziale si arriva a quello finale analizzando la stringa stessa:

L(M)={ x ∊ ∑\* con ઠ\*(q0,x) ∊ F}

## Automi a stati finiti non deterministici

Consideriamo il seguente esempio:



L’automa permette di riconoscere frasi nella forma (a U b)\*bb\*a, fino a quando la macchina si trova nello stato q0 e sceglie a, non vi è alcun problema, tuttavia ce ne sono quando dalla stato q0 si sceglie b. Dal momento che ho due vie disponibili, quella che fa restare la macchina in q0 e quella che la porta in q1, quale devo scegliere?

L’automa dell’esempio è non deterministico, ovvero dato uno stato e il valore della testina dell’input, la funzione di transizione restituisce un insieme di stati.

ઠ:QxΣ→

Questi automi, data la loro semplicità di scrittura, sembrano più potenti di quelli deterministici, tuttavia è facilmente dimostrabile che non è così e che esistono tecniche per convertirli.

L’automa non deterministico è una quintupla N=<Q,Σ,ઠ,q0,F> tutti i campi tranne ઠ hanno lo stesso significato di quelli deterministici, la tipatura di quest’ultimo è infatti scritta come sopra. Come per gli automi deterministici, è possibile definire ઠ\*:QxΣ\* → dimostrabile per induzione nel seguente modo:

Base: ઠ\*(q,ε)={q}

Ipotesi: ઠ\*(q,Ya)=ઠ\*(ઠ\*(q,Y),a)=ઠ(ઠ\*(q,Y),a) con Y∈∑\* e a∈∑

Che risultato dà ઠ\*(q,Y)? negli automi deterministici la macchina finisce in un solo stato, con quelli non deterministici invece è possibile finire in più stati, quindi:

ઠ\*(q,Ya)={p \ ∃ r ∈ δ\*(q,y) ˄ p ∈ δ(r,a)}

ઠ\*(q,Ya) é quindi l’insieme di tutti gli stati p tale che esiste uno stato r appartenente all’insieme ઠ\*(q,Y) e p deve appartenere all’insieme ઠ\*(r,a).

Dato un automa non deterministico N, il linguaggio accettato da N è l’insieme di tutte le stringhe che a partire da q0 fino a un insieme di stati in cui è presente almeno uno stato finale.

L(N)={x∈∑\* \ ઠ\*(q0,x) ∩F≠Ø}

Esempio: dato l’automa del precedente esempio, riconoscere la frase abba.

La prima lettera è una a, quindi seguendo la freccia la macchina rimane nello stato iniziale q0:

1. q0,a → q0;

Con la lettera b ci sono due percorsi differenti, essi verranno presi in contemporanea:

1. q0,b -->q0
2. q0,b -->q1

Con la seconda b si hanno 3 alternativa: andare da q0 a q1, rimanere in q1 oppure rimanere in q0:

1. q0,b → q0
2. q0,b → q1
3. q1,b → q1

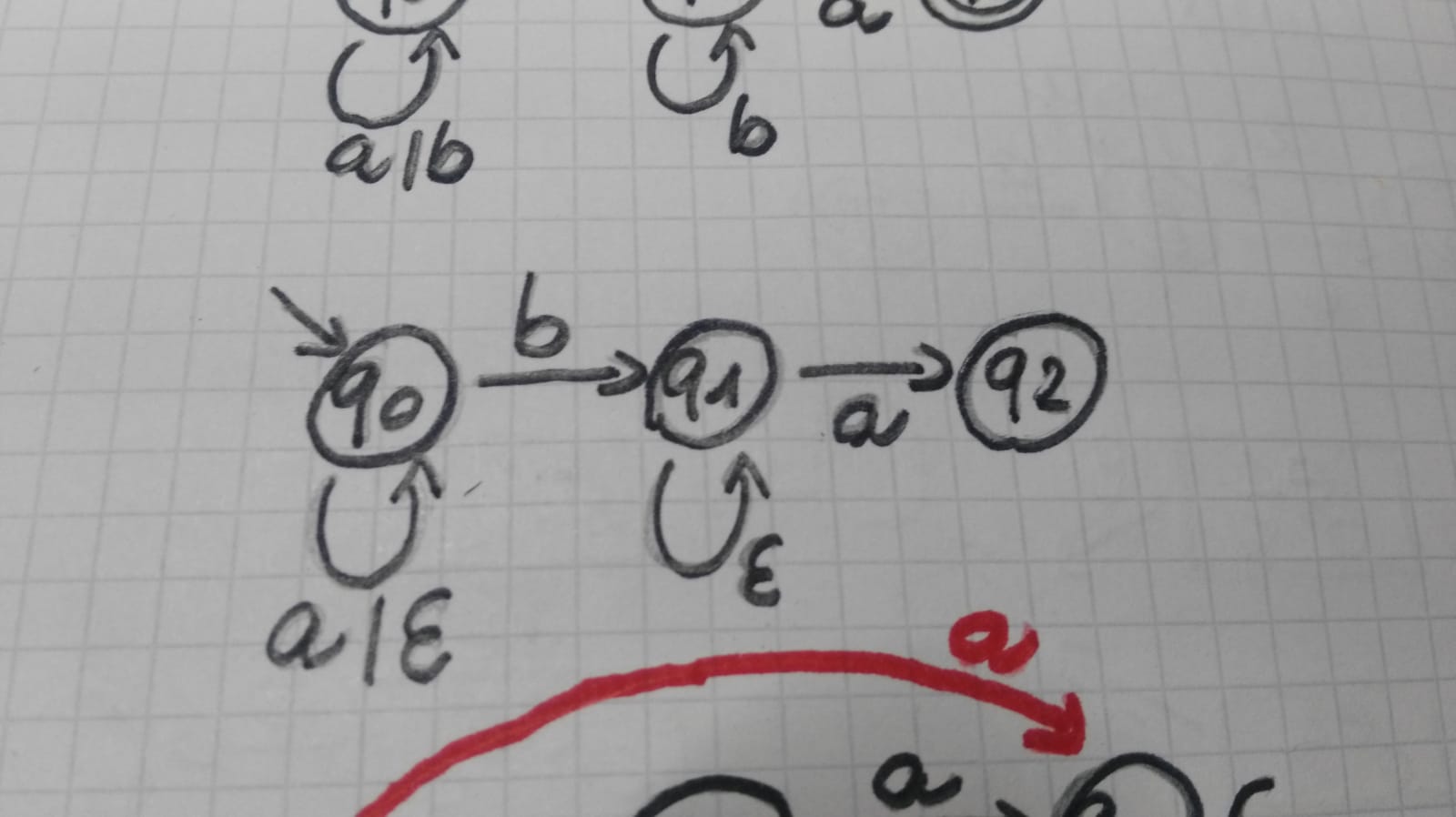
Con la seconda a si può rimanere in q0 oppure da q1 si può andare in q2, lo stato finale:

1. q0,a → q0
2. q1,a → q2
3. q1,a → q2

Conclusioni: Il primo modo va in fail, il secondo e il terzo vengono invece accettati dato che si trovano nello stato finale. abba viene quindi riconosciuto perchè l’insieme di stati ce n’è almeno uno che porta a quello finale.

## Automa non deterministico con ε mosse

Consideriamo ora i seguenti automi



la ε mossa permette di andare da uno stato a un altro senza leggere l’input, nel primo automa non vi è quindi nessuno problema, cosa invece presente nel secondo: Quale via scelgo? Se ho una b in input, non ho problemi, ma nel caso avessi una a la macchina va in entrambi gli stati. Questo tipo di automa non deterministico è detto con ε mosse ed è una quintupla <Q,Σ,ઠ,q0,F>, come nell’automa precedente, la differenza rispetto agli altri sta in ઠ, la sua tipatura infatti è ઠ:Qx(Σ U {ε}) → .In questo tipo di automa v è la funzione ઠ^, ,dal momento che ઠ^(q,ε) non dà più q come risultato, bisogna definire il concetto di ε-chiusura di uno stato: esso è l’insieme di tutti gli stati raggiungibili da q utilizzando solo le ε mosse. Essa è una chiusura transitiva, quindi se da q e ε vado in t e da qui (sempre con ε) vado in s, la ε-chiusura sarà l’insieme {q, t, s}, quindi:

q ∈ ε-chiusura(q)

t ∈ ε-chiusura(q) se ∃ s ∈ ε-chiusura(q) ˄ δ(s, ε) = t

Di conseguenza è possibile definire ઠ^ come:

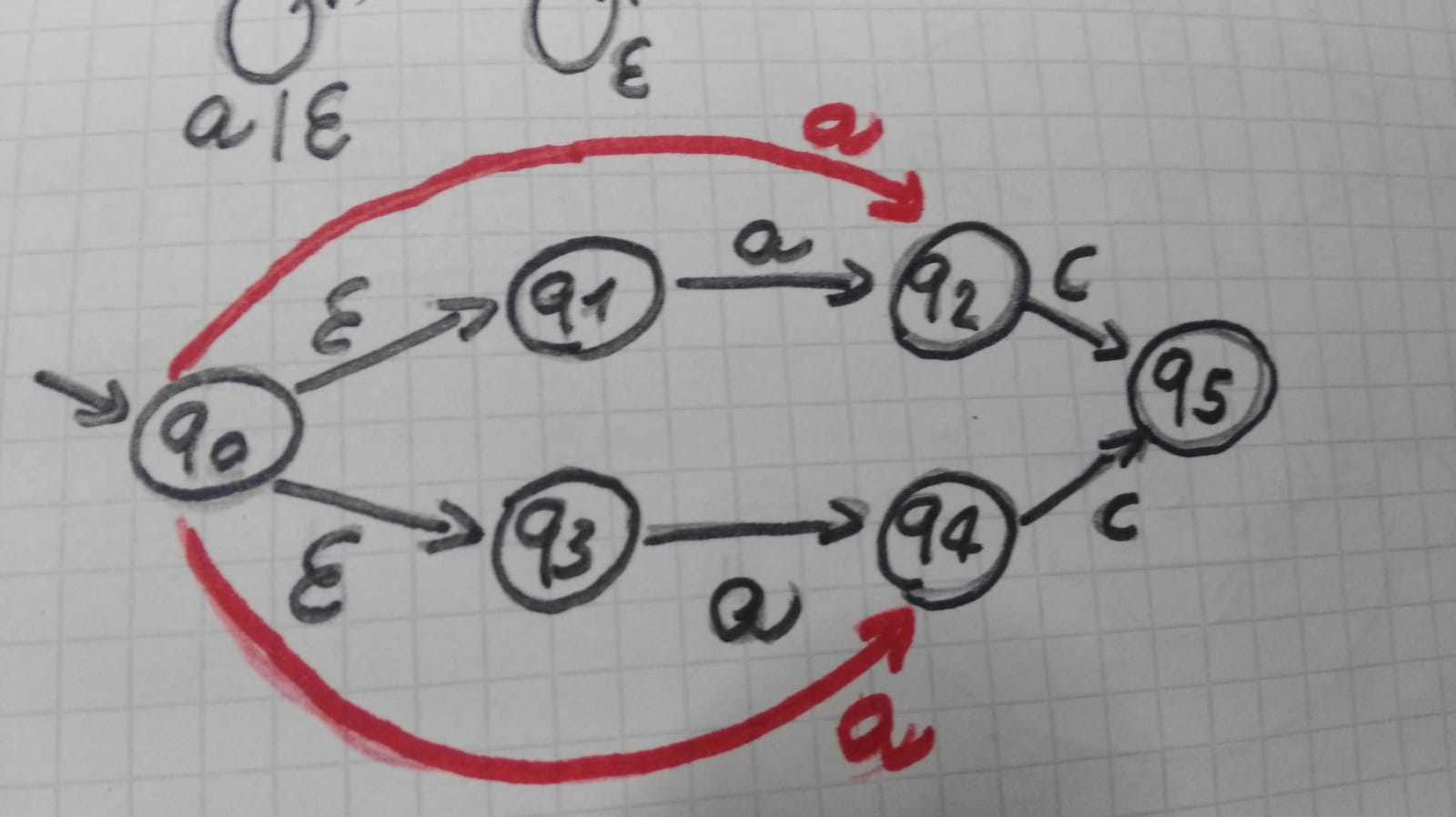
ઠ^(q,ε)=ε-chiusura(q)

ઠ^(q,Ya)=ε-chiusura(P) con P={p \ ∃ r ∈ ઠ^(q,Y) ˄ ઠ(r,a)=p}

In poche parole: a partire da uno stato p si va negli stati a cui porta Y, poi a quelli in cui porta a e infine a quelli in cui porta l’ε-chiusura.

## Rimozione delle ε-mosse

Per rimuovere le ε mosse dall’automa, l’idea è dire le stesse cose senza utilizzarle, quindi dato un automa N=<Q,Σ,δ,q0,F> non deterministico e con ε mosse, si deve costruire un automa N’=<Q’,Σ’,δ’,q0’,F’> senza le ε mosse. Per farlo si controllano tutti i cammini con ε mosse che, per ogni stato q e per ogni simbolo a dell’alfabeto, portano a un cammino con a a partire da q, quindi si tolgono tutti gli archi con ε e si aggiunge un arco a che parte da q e arriva allo stato a cui porta il cammino con a.



Di conseguenza bisogna definire una funzione di transizione ઠ’ che faccia le stesse cose di ઠ utilizzando lo stesso insieme di stati e lo stesso alfabeto ma senza ε mosse, quindi in base al precedente esempio:

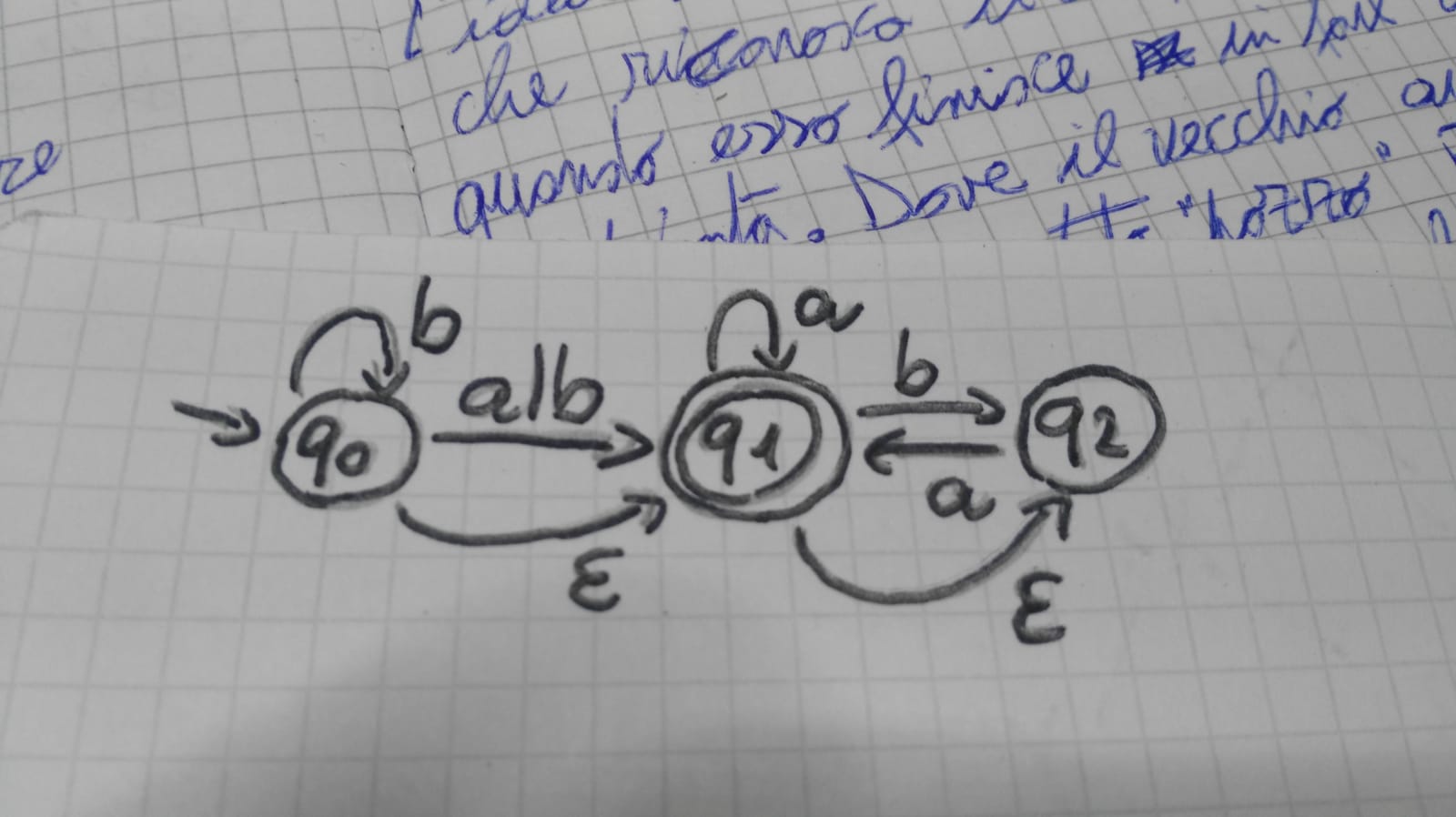
∀a∈Σ e ∀q∈Q ઠ’(q,a)=ઠ^(q,a)=ε-chiusura(ઠ(ε-chiusura(q),a))

Bisogna quindi implementare un algoritmo che calcoli il risultato di ઠ’.

Per quanto riguarda l’insieme di stati finali del nuovo automa F’, si ha che:

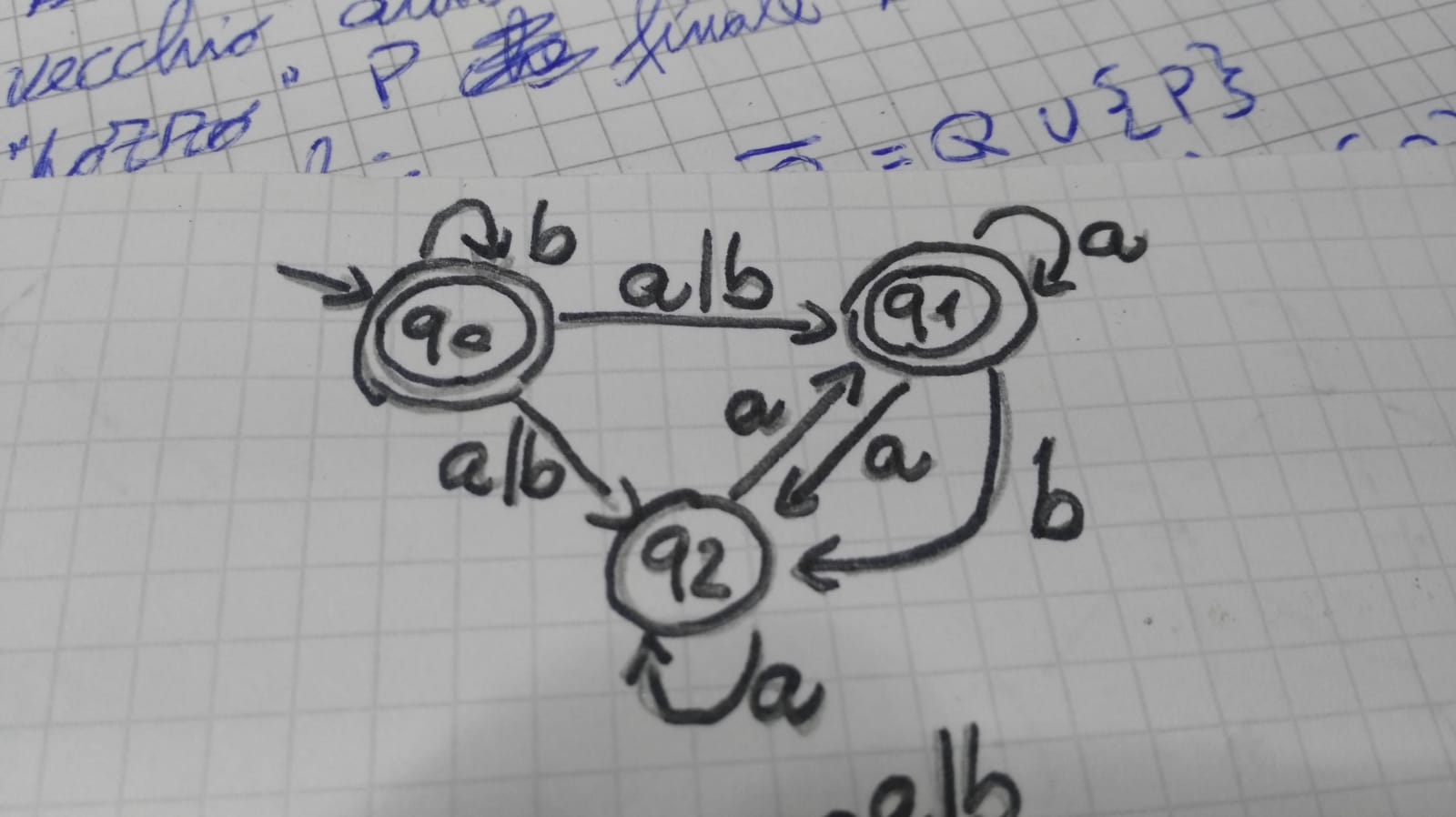
Questo perchè, supponendo che a partire da q0 vi sono delle ε mosse che portano a uno stato finale s, dal momento che non vi sono più ε mosse, per raggiungere s bisogna rendere q0 esso stesso uno stato finale. In questo modo l’automa non rischia il fail e non cambia il suo modo di operare.

Esempio: Dato un automa con ε mosse, si deve convertire in un automa non deterministico normale, per farlo si utilizza una tabella in cui si indicano i risultati delle operazioni di ઠ’ su ogni stato.



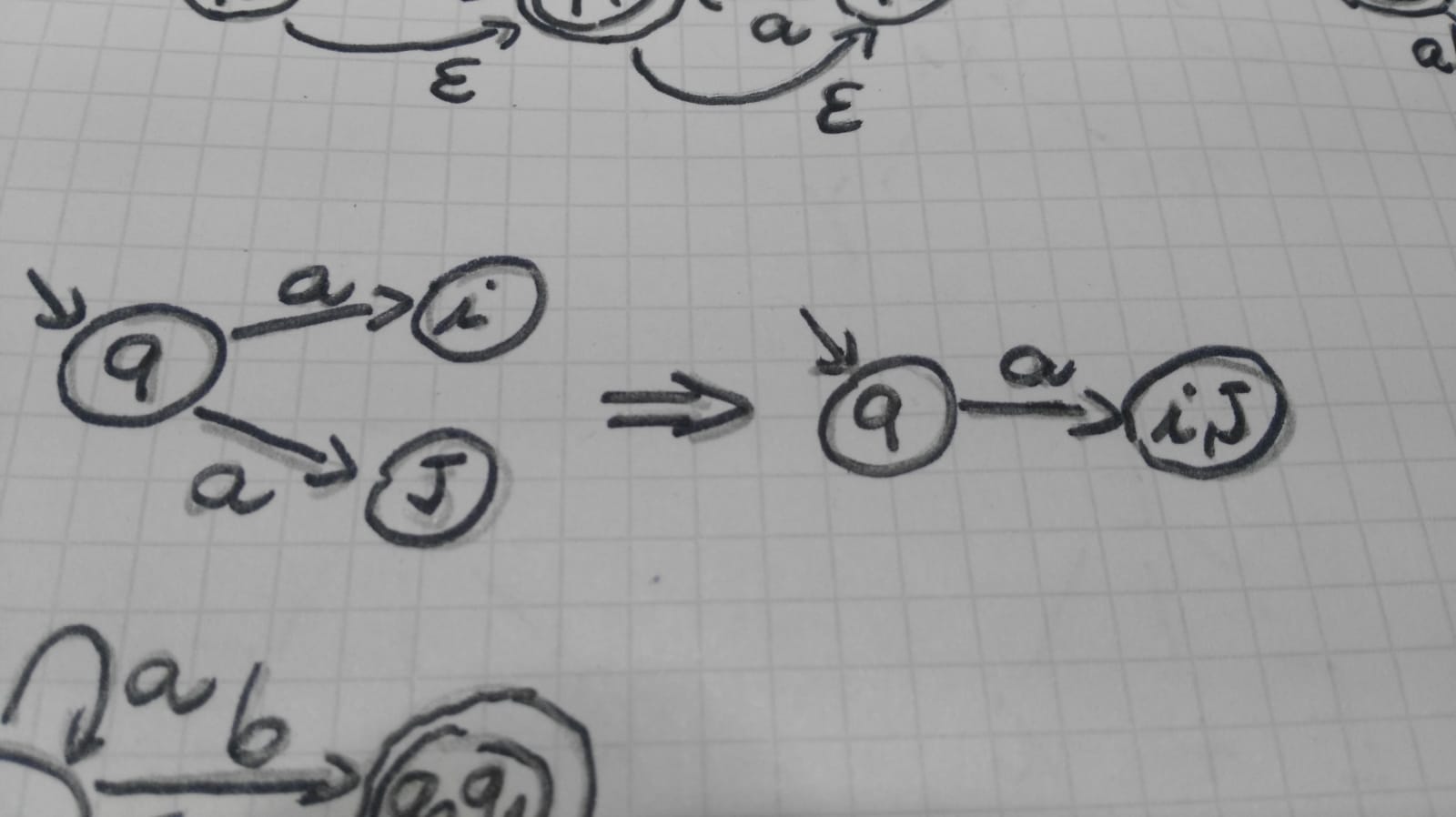
|  | q0 | q1 | q2 |
| --- | --- | --- | --- |
| r=ε-chiusura(qi) | q0, q1, q2 | q1, q2 | q2 |
| qa=ઠ(r,a) | q1 | q1 | q1 |
| ε-chiusura(qa) | q1, q2 | q1, q2 | q1, q2 |
| qb=ઠ(r,b) | q0, q1, q2 | q2 | Nessuno |
| ε-chiusura(qb) | q0, q1, q2 | q2 | Nessuno |

q0 ha un percorso verso lo stato finale q1, quindi è anch’esso uno stato finale. L’automa finale si ricava considerando le righe riguardanti le ultime ε-chiusure, il questo caso ε-chiusura(qa) e ε-chiusura(qb)



## Rimozione del non determinismo

Per rimuovere il non determinismo da un automa devo visitare tutto l’automa, ogni passo consuma un simbolo in input e quindi la complessità in tempo è lineare. Nel caso peggiore però vi è un tempo esponenziale dato che la macchina ha bisogno di esplorare tutto lo spazio quando si ha a che fare con automi deterministici. Come evitare ciò? se in un automa non deterministico da uno stato q posso andare in i e j con un carattere a, per renderlo deterministico si può dire che la transizione ઠ(q,a) porta a i+j, uno stato rappresentante un insieme di stati dell’automa di partenza.



L’idea funzione a livello teorico, in pratica invece bisogna tenere conto di tutte le combinazioni di stati possibili, di conseguenza si ha un tempo lineare al costo di uno spazio esponenziale.

Dato un automa N=<Q, ∑, δ, q0, F>, si deve ottenere un automa M=<Q’, ∑, δ’, q0’, F’> deterministico. Dato che lo spazio diventa esponenziale durante il passaggio, l’insieme di stati Q’ è l’insieme delle parti di Q ().

Lo stato iniziale q0’ rimane lo stesso (q0’=q0), per quanto riguarda quelli finali, tutti gli insiemi di stati che contengono almeno uno stato finale sono considerati anch’essi finali:

F’={[qi,...,qj] \ {{[qi,...,qj]} ⋂ F ≠ Ø}

La funzione di transizione δ’ lavora sui nuovi stati dell’automa, quindi:

∀ [qi ... qj] ∈ Q', ∀ a ∈ ∑, δ’([qi,...,qj],a)=[p1,...,pk] ↔️ ∀ q ∈ {qi ... qj},

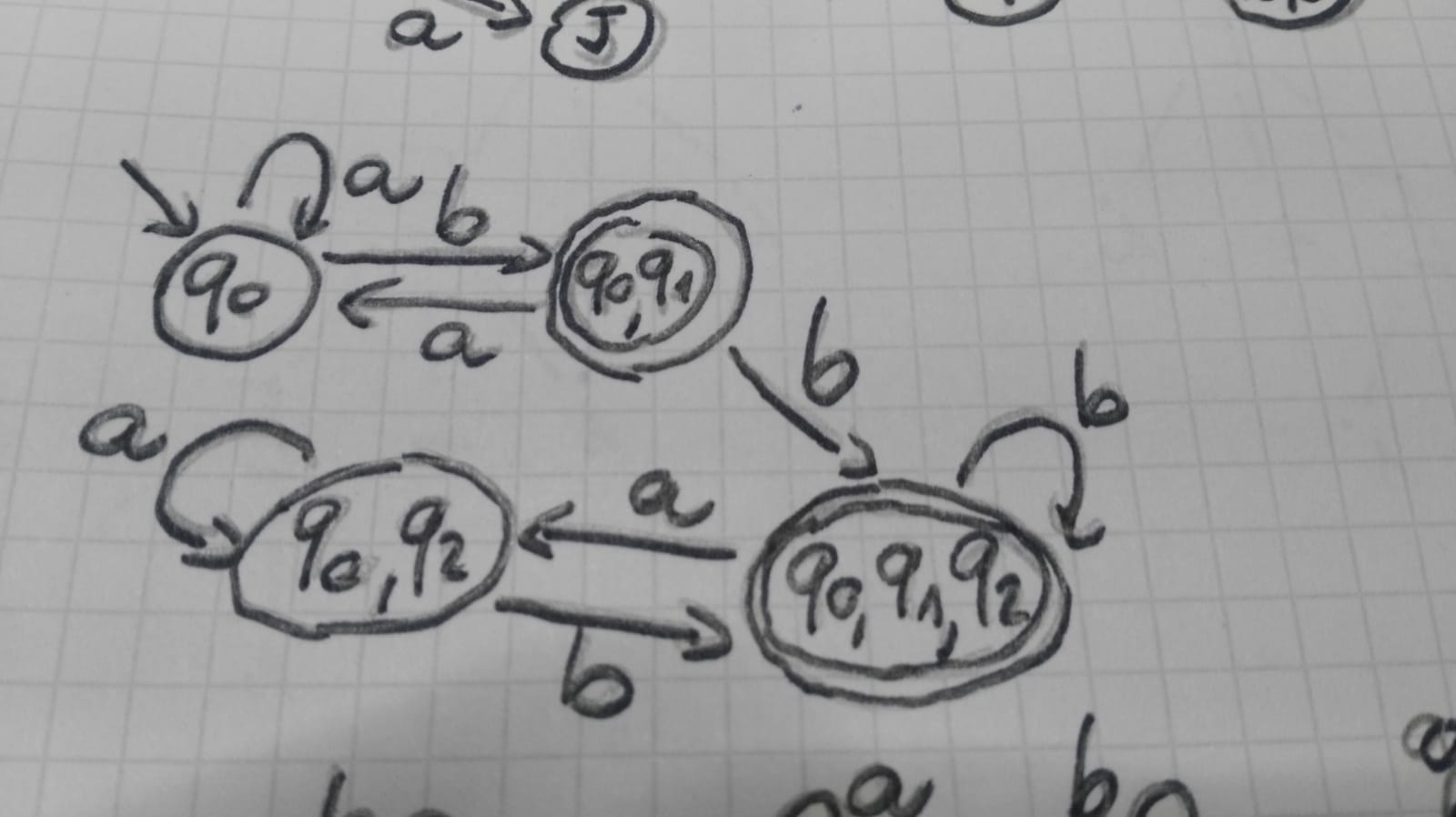
Come nel caso precedente, anche questa rimozione utilizza una tabella per la ricostruzione dell’automa, in essa vi è una particolarità: i campi degli stati “singoli” (vi è un solo stato all’interno dell’insieme di stati) ha le celle uguali alle ultime ε-chiusura della tabella precedente, tutte le altre sono le loro unioni.

Esempio: dato l’automa non deterministico senza ε-mosse, si ha la seguente tabella:



|  | q0 | q1 | q2 | q0,q1 | q0,q2 | q1,q2 | q0,q1,q2 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| (qi,a) | q0 | q0 | q0,q2 | q0 | q0,q2 | q0,q2 | q0,q2 |
| (qi,b) | q0,q1 | q2 | q2 | q0,q1,q2 | q0,q1,q2 | q2 | q0,q1,q2 |

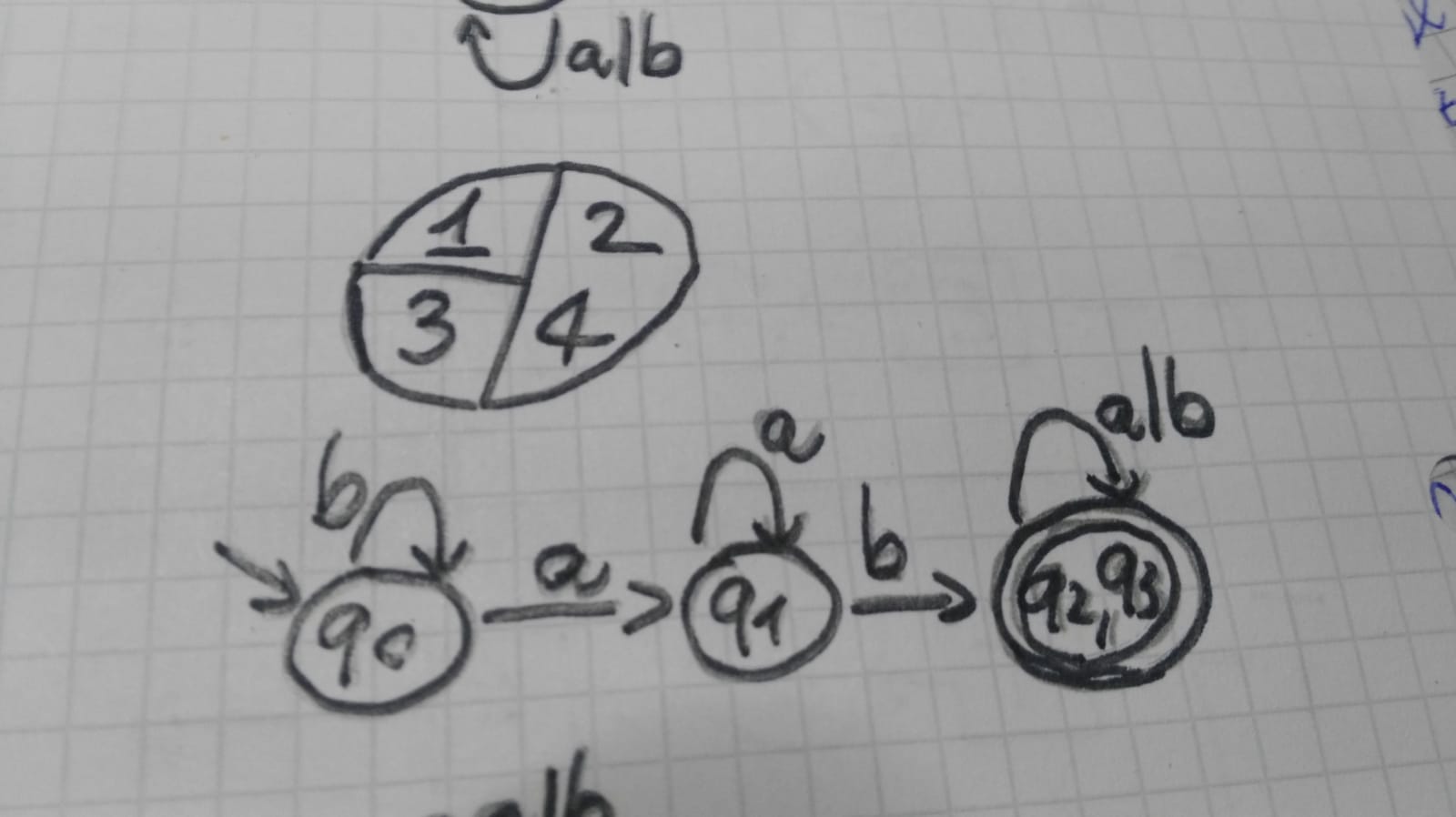
Dalla tabella sopra si ricava il seguente automa:



Lo stato di partenza non cambia mentre quelli finali devono contenere almeno uno stato finale dell’automa originale.

## Minimizzazione

Un automa è minimizzabile quando esiste una versione minima che svolge la stessa funzione, esso è anche unico tranne nei casi in cui vi sono isomorfismi. La minimizzazione si basa sulla fusione di stati indistinguibili, ovvero si uniscono uno o più stati per ottenere il numero minimo di stati possibile. La Indistinguibilità è data dal comportamento degli stati stessi, infatti due o più stati sono indistinguibili quando si comportano allo stesso modo per ogni stringa in input. La minimizzazione è una relazione di equivalenza esaustiva e mutualmente esclusiva, ovvero che l’insieme totale degli stati viene suddiviso in sottoinsiemi ognuno rappresentante uno stato, ogni stato non resta fuori dai sottoinsiemi (mutua esclusione) e ogni sottoinsieme ne contiene almeno uno (esaustione), essa è anche transitiva (se q è indistinguibile da p e p lo è per t, allora q lo è per t), simmetrica (se uno stato q è indistinguibile da p, allora anche p lo è per q) e riflessiva (ogni stato è indistinguibile da sè stesso). Consideriamo il seguente esempio:



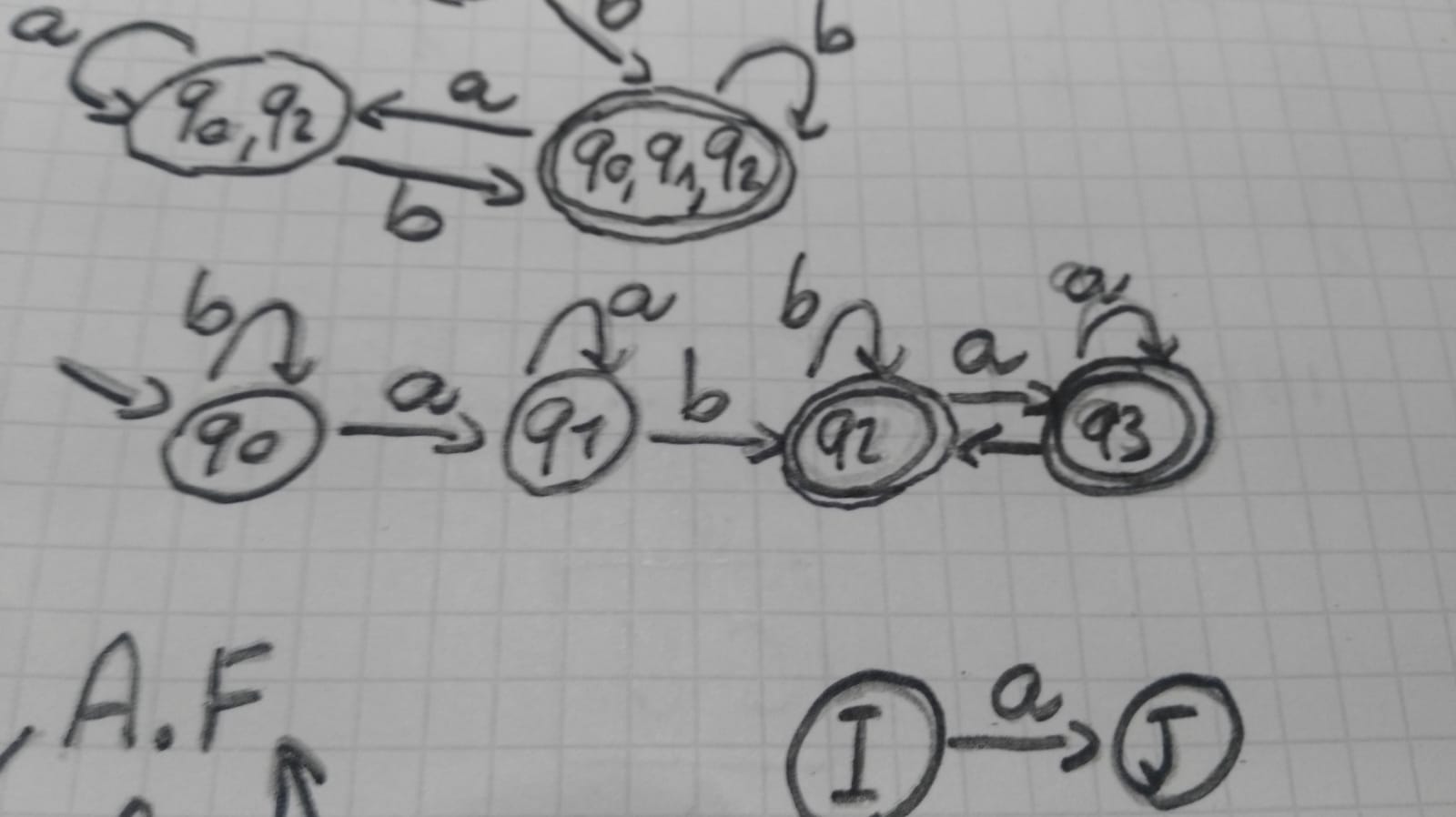
Com’è possibile notare, gli stati sono stati suddivisi in 4 partizioni, quindi l’automa minimizzato avrà 4 stati.

L’algoritmo di minimizzazione deve quindi prendere un automa M=<Q, ∑, δ, q0, F> e creare M’=<, ∑, δ’, {q0}, F’>, per farlo si compila una tabella di indistinguibilità, essa viene compilata solo a metà dato che la minimizzazione è riflessiva e simmetrica.

L’insieme degli stati di M’ è l’insieme delle parti di Q, F’ è l’insieme di partizioni in cui è presente lo stato finale e la funzione δ’ è descritta nel seguente modo:

δ’({qi,...,qk},a)={p1,...,pk} ↔️ ∀q∊{qi,...,qk}, (Uq δ(q,a))={p1,...,pk}

Esempio: Dato un automa deterministica come in figura:



Nella tabella si segnano con una X le coppie di stati distinguibili, per trovarle si eseguono i seguenti passaggi:

1. si segnano tutte le coppie (p,q) in cui p è finale e q no;
2. per ogni coppia non marcata e per ogni simbolo dell’alfabeto si controlla se la funzione di transizione porta allo stesso stato.

| q1 | 1,1X |  |  |  |
| --- | --- | --- | --- | --- |
| q2 | X | X |  |  |
| q3 | X | X | 2,2  3,3 |  |
|  | q0 | q1 | q2 | q3 |

Per quanto riguarda il punto 2, le coppie rimaste sono (q0,q1) e (q2,q3), quindi si è svolto il seguente ragionamento:

q0,a → q1

q1,a → q1

Com’è possibile notare, entrambi gli stati portano a uno stato non ancora segnato, ovvero q1, quando c’è il carattere a, quindi q0 e q1 potrebbero essere indistinguibili.

q0,b → q0

q1,b → q2

Qui possiamo concludere che q0 e q1 sono indistinguibili, dato che (q1,b) porta a uno stato segnato nella tabella, ovvero q2.

Per quanto riguarda la coppia (q3,q2), invece:

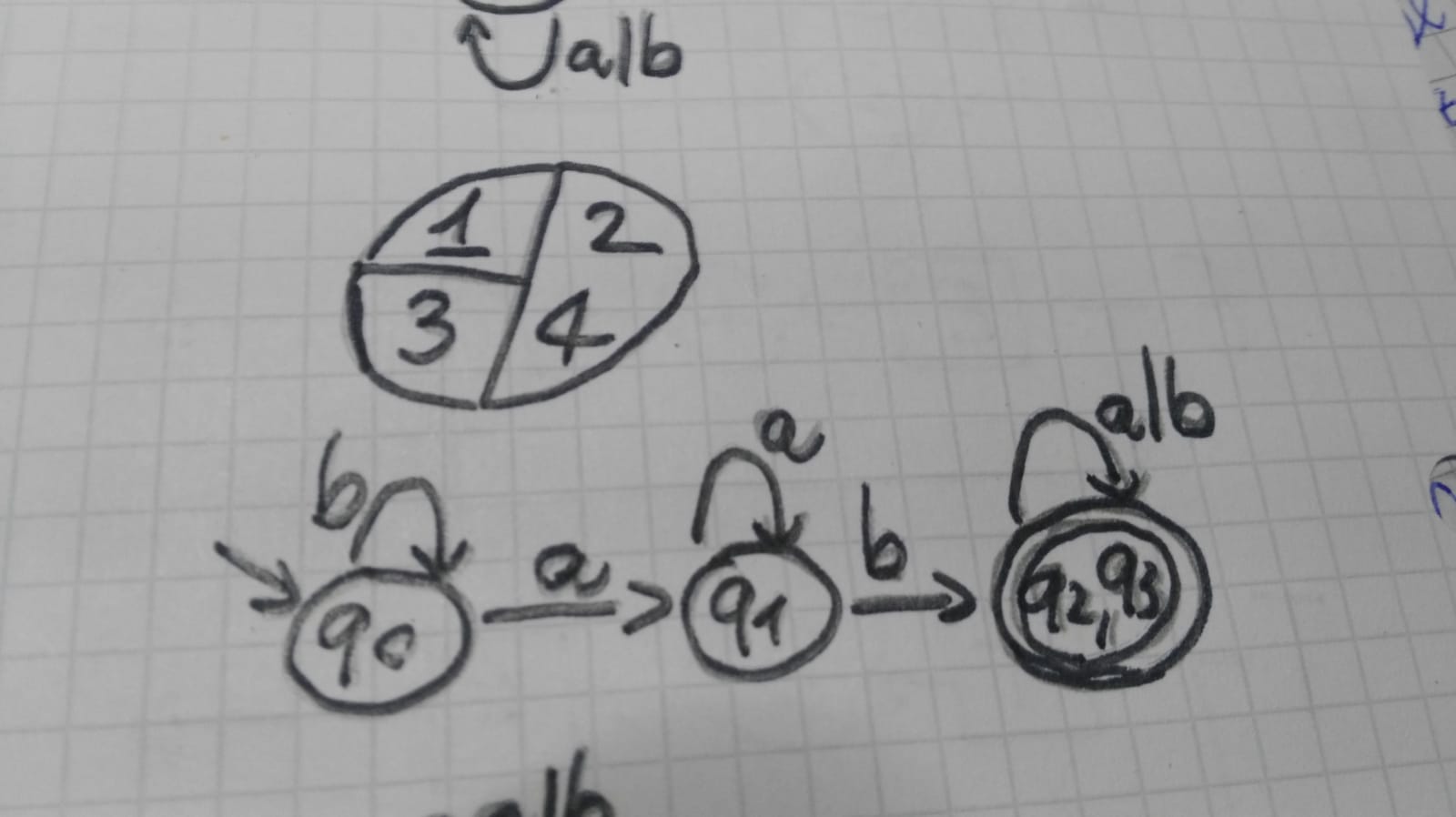
q2,a → q3

q3,a → q3

q2,b → q2

q3,b → q2

Sia q2 che q3 portano a stati non segnati sia con a che con b, quindi sono indistinguibili e si possono unire, l’automa minimizzato sarà quindi il seguente:



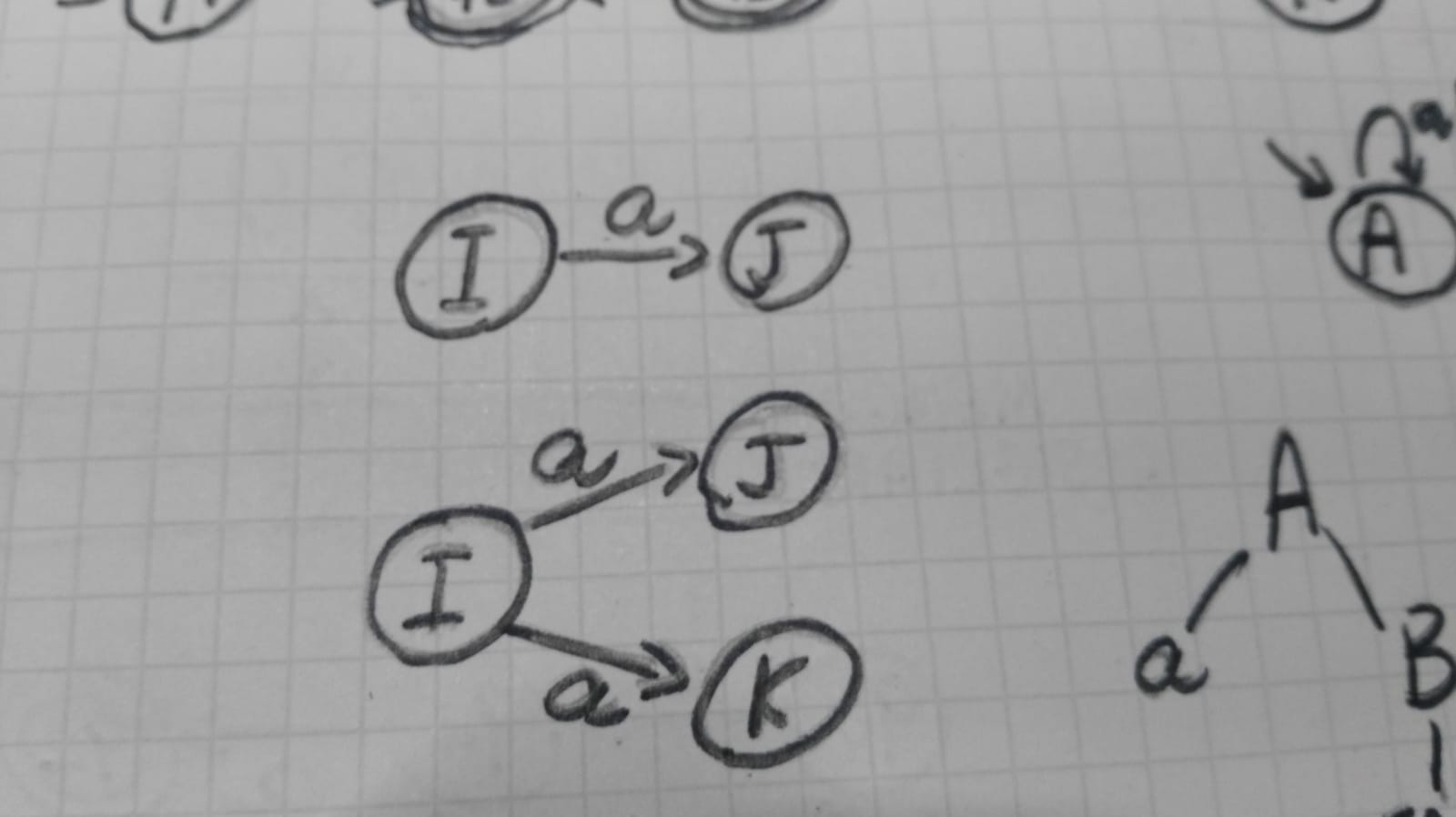
## Dimostrazione di contenimento tra automi a stati finiti e linguaggi regolari

Come si può evincere dall’immagine, parlare di linguaggi destri, regolari e di automi a stati finiti è la stessa cosa, infatti per ogni grammatica destra esiste un’espressione regolare equivalente, può costruire un automa a stati finiti o essere riconosciuto da esso.

### Contenimento Automa ← Linguaggio destro

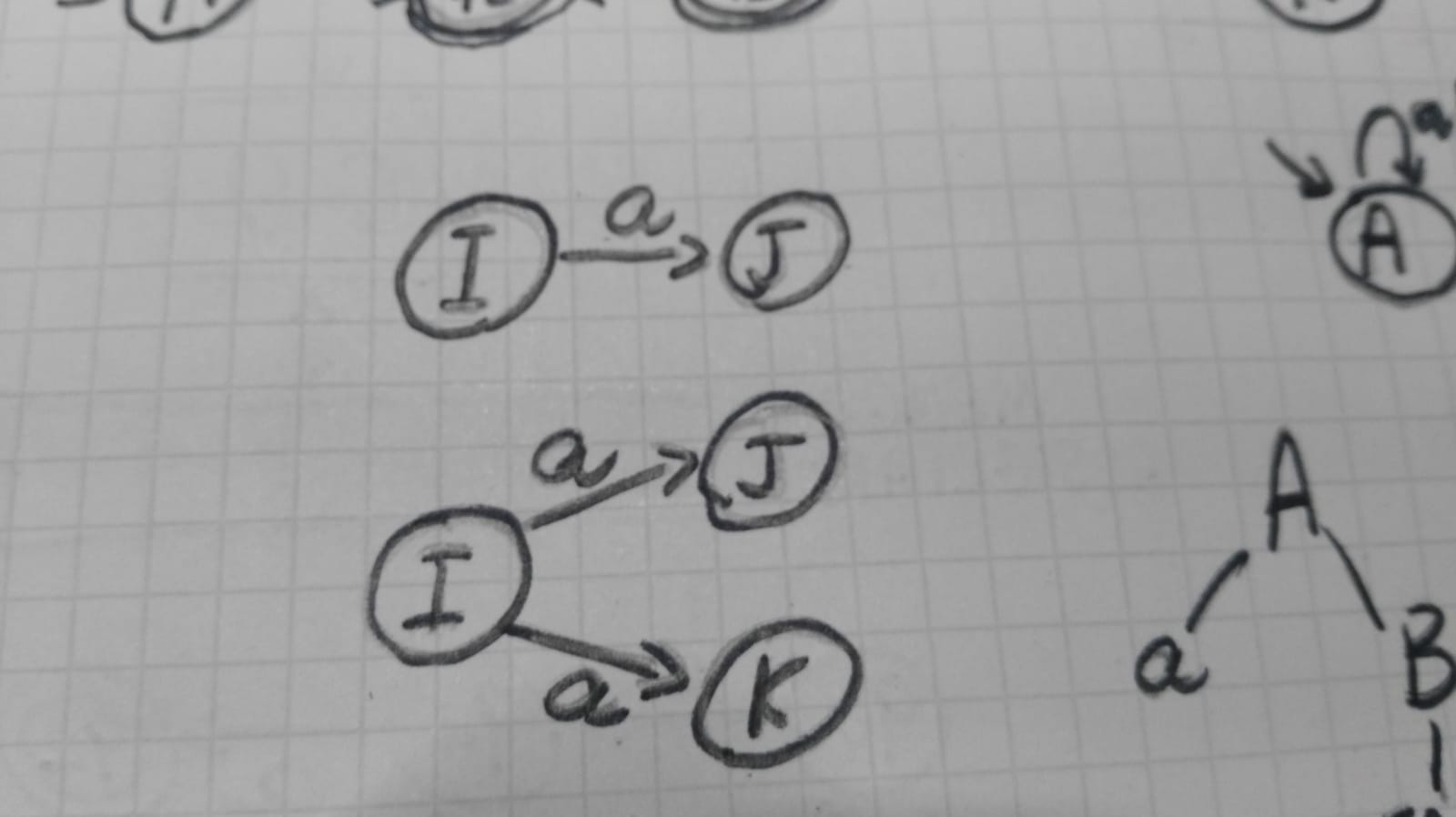
Dato un automa a stati finiti N=<Q, ∑, δ, q0, F> bisogna costruire una grammatica G=<V,∑,P,S> equivalente. L’idea è che ogni passo di transizione dell’automa corrisponde a un passo di generazione nella grammatica, quindi ogni stato corrisponde a un simbolo non terminale mentre ogni transizione a un terminale.

Esempio: dato il seguente automa



La grammatica equivalente è I→ aJ

Esempio: dato il seguente automa non deterministico



La grammatica equivalente sarà I→ aJ|aK

Per ogni stato finale si aggiunge q’ → ε, nel caso degli esempi precedenti;

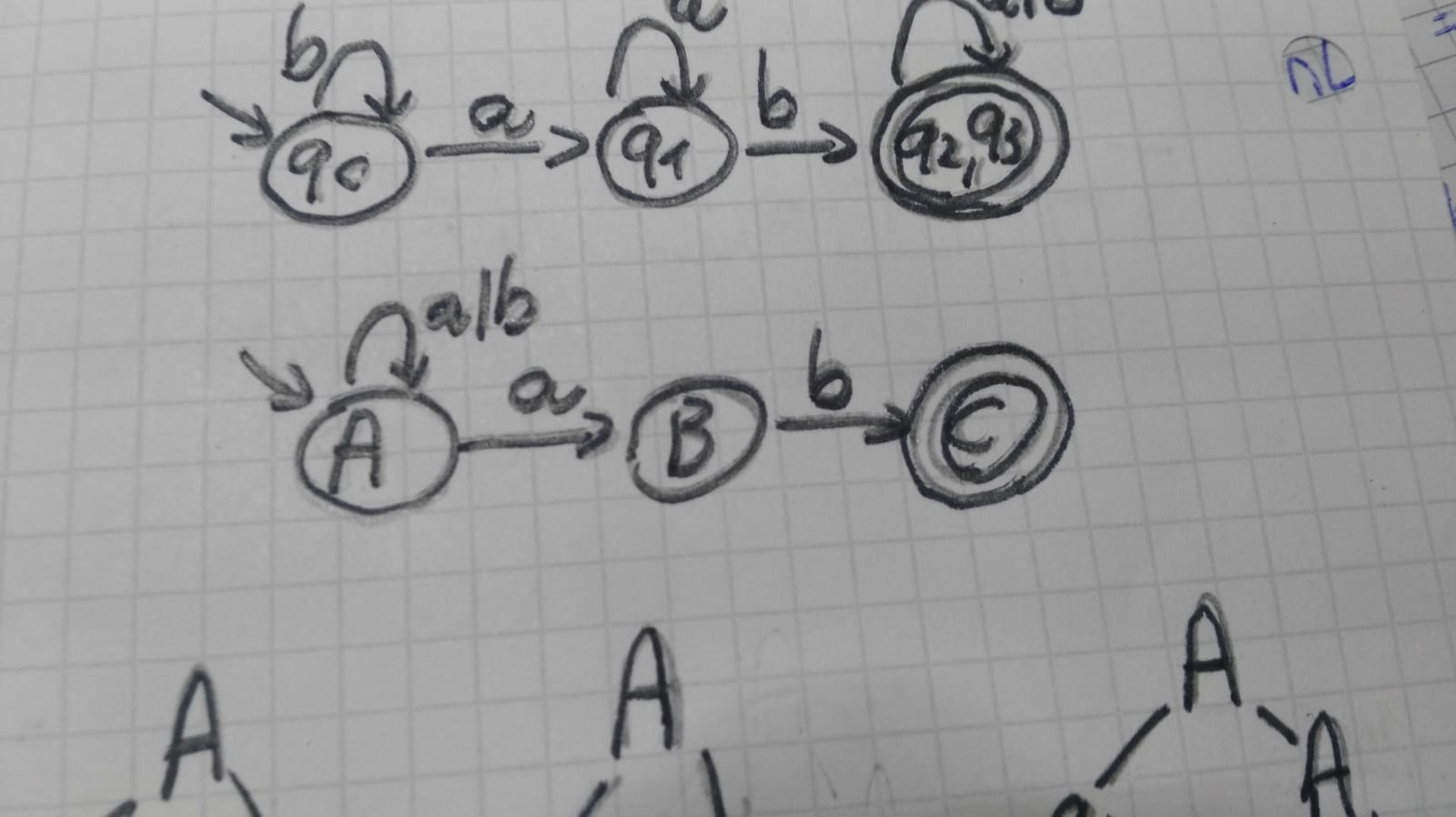
J→ε K→ε

Lo start symbol S è dato dallo stato iniziale dell’automa mentre l’insieme di regole P è generato nel seguente modo: Per ogni simbolo dell’alfabeto e per ogni stato q e q’, q’ appartiene alla transizione (q,a) e q genera il carattere “a” e il non terminale q’. In più per ogni stato q finale, q deve generare la stringa vuota.

∀a∊∑, ∀q,q’∊Q, q’∊δ(q,a), q→ aq’∊P

∀q∊F, q→ε ∊ P

Esempio: Dato il seguente automa, ricavare la grammatica equivalente



S=A

A→ aA|bA|aB|bC

C→ ε

La grammatica e l’automa possono generare la stringa “aab”?

I passi da fare sono:

* A,a→ A A→ aA

A,a→ B A→ aB

* A,a→ A A→ aA

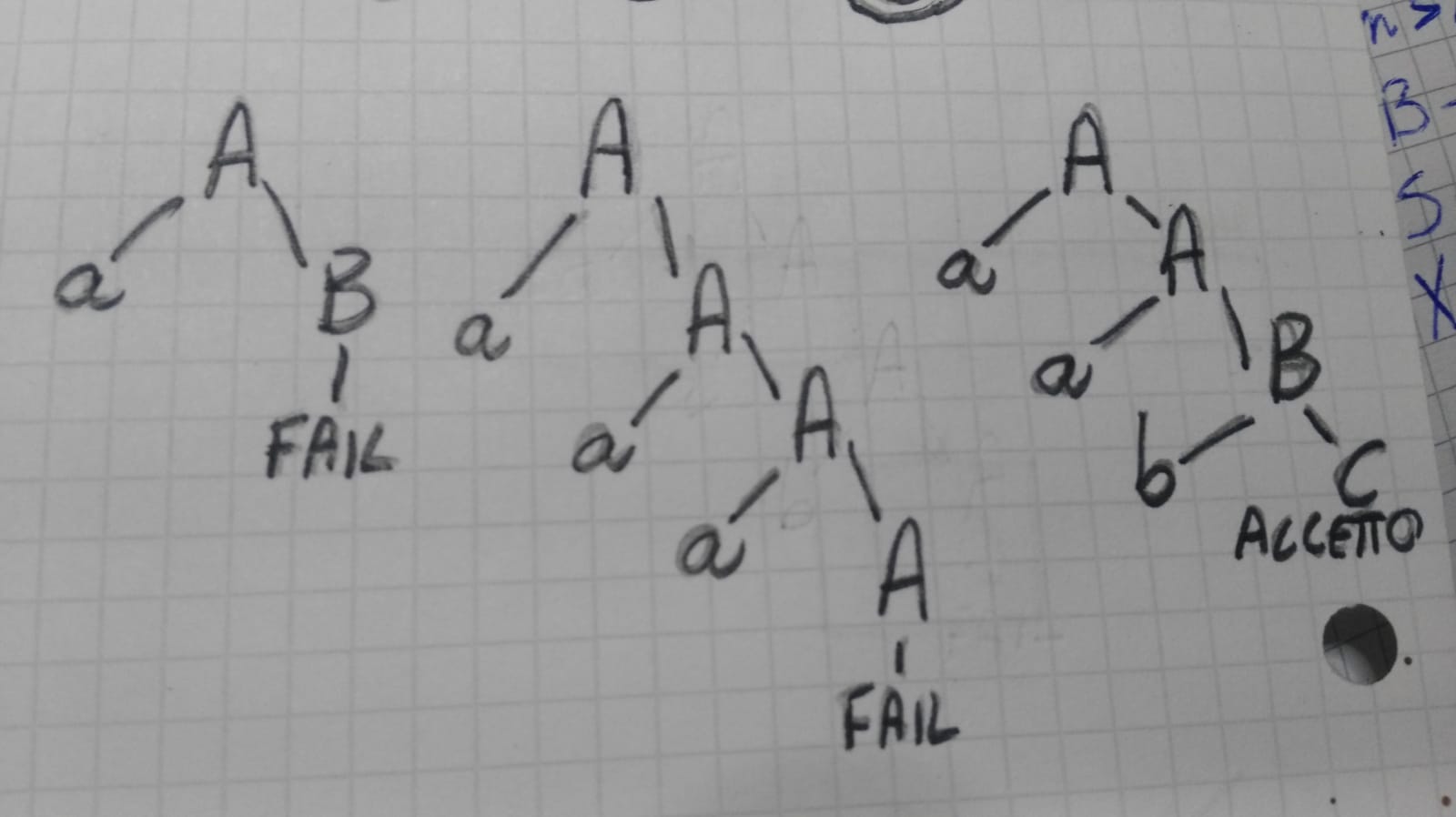
A,a→ B A→ aB

B,a→ FAIL B→ FAIL

* A,b→ A A→ bA

B,b→ C B→ bC

Attraverso gli alberi di derivazione si può vedere che vi è un caso in cui la stringa “aab” viene accettata.



### Contenimento Linguaggio destro ← Automa

Data una grammatica destra G=<V, ∑, P, S> si vuole costruire un automa M=<Q, ∑, δ, q0, F> equivalente. In questo caso si fa il ragionamento inverso a quello precedente: ogni regola nell’insieme P equivale a una transizione tra stati:

∀ regola A→ aB, B∊δ(A,a)

La grammatica smette di generare in due sole situazioni: quando genera una stringa vuota (A → ε) oppure un terminale (A → a). Un non terminale che genera ε nella grammatica diventa uno stato finale nell’ automa, come si gestisce invece la seconda situazione? Si genera un nuovo stato T finale e si arriva a esso tramite una a-mossa. Di conseguenza l’insieme di stati Q equivale all’insieme di non terminali V più lo stato T mentre F è formato da T e da tutti i non terminali che generano ε.

Q=V U {T} q0=S

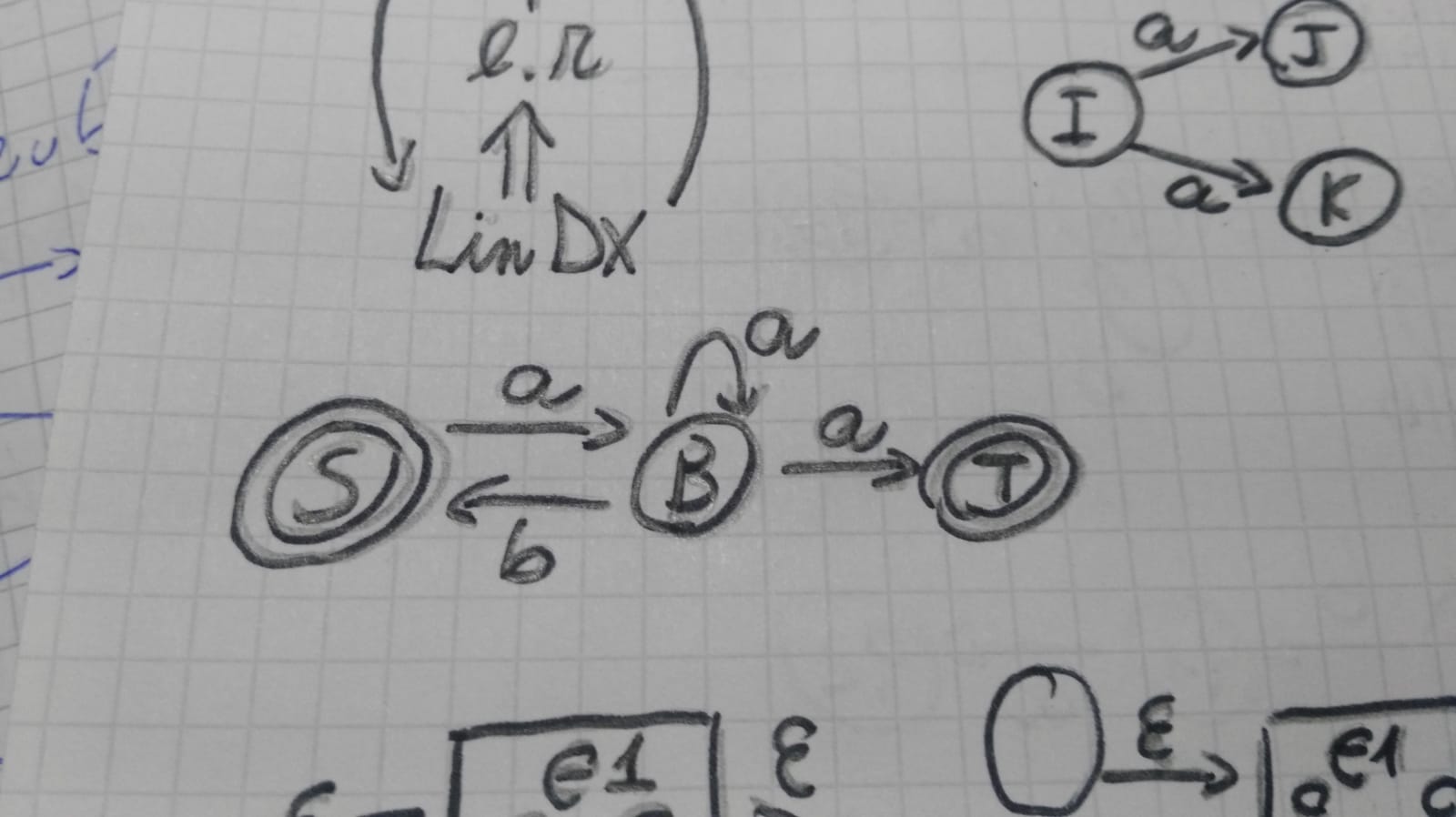
F={T} U {∀A,A→ ε}

Esempio: La seguente grammatica:

S→ aB|ε

B→ aB|bS|a

L’automa equivalente è il seguente:



#### 

#### Costruzione dell’automa

L’idea di base è mettere in mezzo allo stato iniziale e quello finale tutte le regole della grammatica, per farlo si utilizza la seguente tabella:

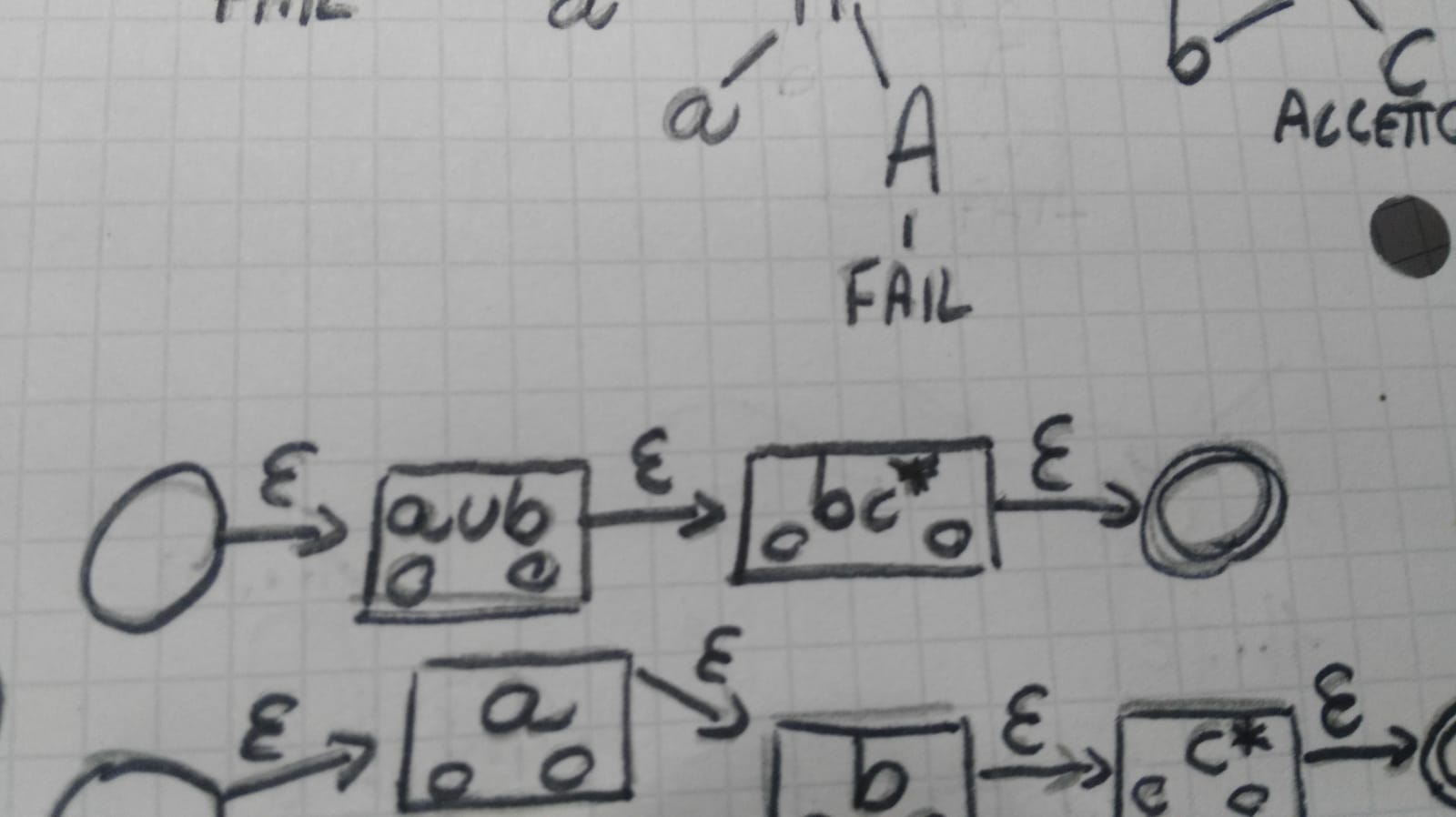
Data un’espressione regolare e, essa si rappresenta con:

<Automa espressione regolare>

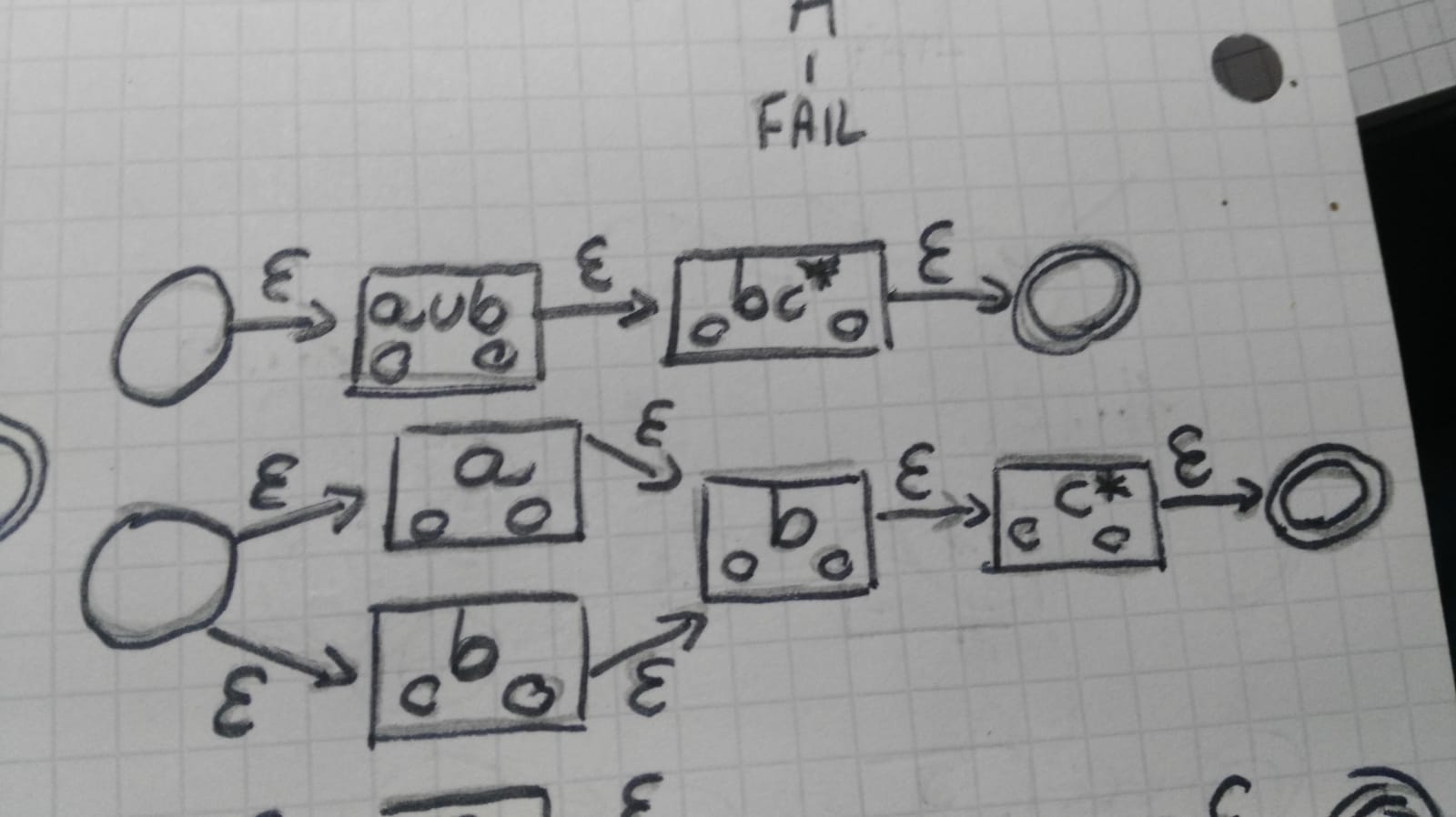
| Insieme vuoto | NIENTE | Se non ci sono regole, non si aggiunge niente. |
| --- | --- | --- |
| Stringa vuota | ε-mossa | Se A→ ε si aggiunge una ε-mossa verso lo stato finale. |
| ∀a∊∑ | a-mossa | Se A→a si aggiunge una a-mossa. |
| e1 U e2 |  | Automa dell’unione tra espressioni regolari. |
| e1.e2 |  | Automa del concatenamento tra espressioni regolari. |
| e\* |  | Automa della chiusura di Kleene. |

Esempio: data la grammatica (a U b).b.c\*, costruire l’automa equivalente:

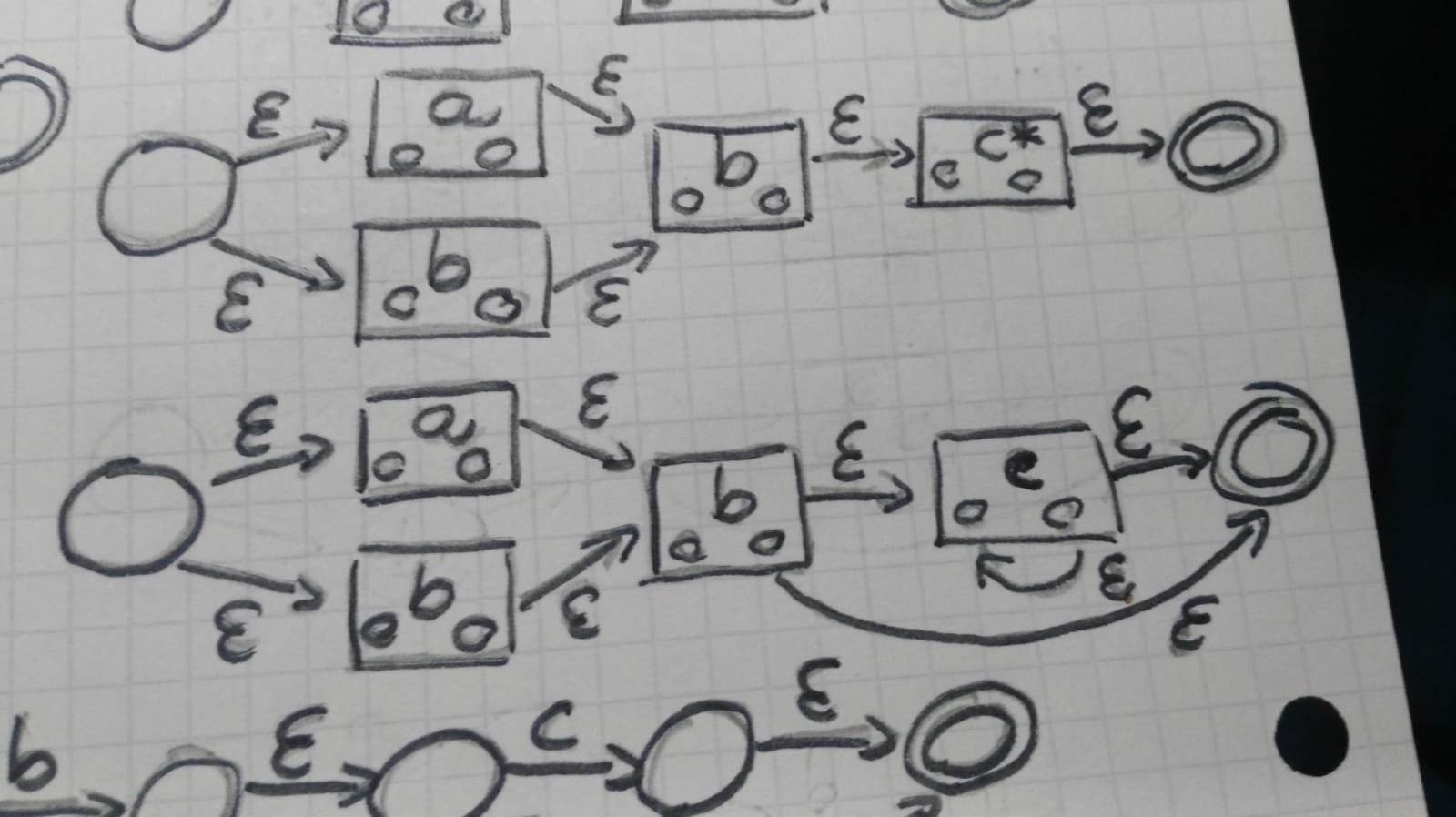
Passo 1: si divide (a U b) da b.c\*



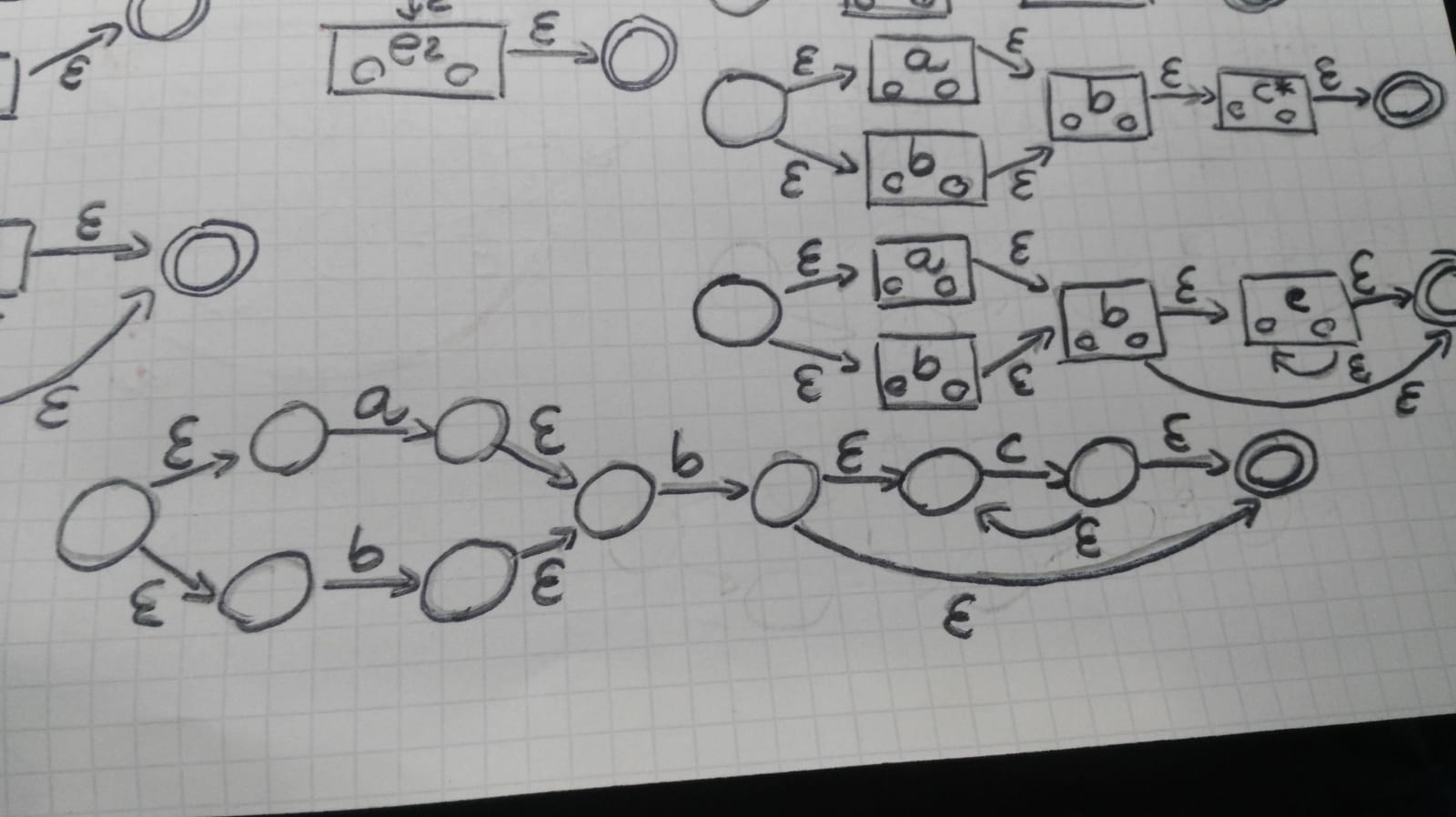
Passo 2: si divide (a U b) in a e b e b.c\* in b e c\*



Passo 3: si risolve c



Passo 4: si sostituiscono le espressioni regolari con le posse opportune.



## Chiusura rispetto al complemento

Come abbiamo visto in precedenza, una proprietà dei linguaggi regolari è la chiusura rispetto all’operatore, ovvero che un’operazione tra linguaggi regolari genera un linguaggio anch’esso regolare. L’unica dimostrazione fatta è la chiusura rispetto all’unione, tuttavia questa regola vale anche per il complemento: Dato un automa M=<Q, ∑, δ, q0, F>, bisogna ottenere -M=<-Q, ∑, -δ, q0, -F>. Un’idea semplice è quella di rendere finali gli stati che non lo sono e viceversa, tuttavia c’è il rischio che l’automa si pianti o che non sia in uno stato finale, di conseguenza bisogna raccogliere tutte queste situazioni e indirizzarle verso uno stato pozzo P finale, il quale accetterà (dove l’automa iniziale si pianta, quello nuovo accetta). Una volta arrivato a P, ogni simbolo in input mi fa ritornare a P. Possiamo quindi dire che:

L’insieme di stati del nuovo automa è uguale a quello precedente con l’aggiunta di P.

-Q = Q U {P}

L’insieme di stati finali del nuovo automa è complementare a quello precedente con l’aggiunta di P.

-F = (Q - F) U {P}

Per ogni stato q e per ogni simbolo a dell’alfabeto, se vi è una transizione (q,a) definita, allora la mantengo nel nuovo automa, altrimenti la indirizzo verso P.

∀ q ∈ Q, ∀ a ∈ ∑, se δ(q, a) è definita, allora -δ(q, a)=δ(q, a), altrimenti -δ(q, a)=P

Ultima ma non meno importante, per ogni simbolo a dell’alfabeto, la transizione (P,a) resterà sempre nello stato P.

∀ a ∈ ∑, -δ(P, a)=P

In conclusione, dato un linguaggio regolare, il suo complemento è anch’esso regolare.

## Chiusura rispetto all’intersezione

Come per il complemento, la chiusura rispetto all’operatore vale anche per l’intersezione, in questo caso però è possibile ragionare in modo logico utilizzando le formule di De Morgan:

Il complemento del complemento di un’intersezione tra linguaggi è uguale all’intersezione stessa:

L1 ∩ L2 = -(-(L1 ∩ L2))

Una delle formule di De Morgan indica che, dati due booleani A e B, -(A ៱ B) = -A v -B, quindi applicando questa formula ai linguaggi:

L1 ∩ L2 = -(-(L1 ∩ L2)) = -(-L1 U -L2)

In conclusione, dati due linguaggi regolari, l’intersezione è anch’essa regolare dato che il complemento e l’unione lo sono.

## 

## Automi per riconoscere i linguaggi context-free

Dal momento che sappiamo generare i linguaggi context-free, li vogliamo anche riconoscere, l’automa a stati finiti però non riesce a riconoscerli a causa delle dipendenze.

Esempio: data la grammatica UcU(r) con U ∊ (a U b)\*, riconoscere la stringa “abbcbba”.

Con l’automa a stati finiti non è possibile riconoscerlo, se ci fosse un limite n su U riuscirei a farlo, ma non è così e quindi si ha bisogno di un nuovo automa: esso è simile a quello a stati finiti con la differenza che vi è una memoria infinita e semplificata, uno stack. Questo tipo di automa è detto a pila e permette di fare i bilanciamenti nei linguaggi context-free, in “si possono riconoscere le frasi facendo la Push ogni volta che viene generata una a e un Pop quando viene generato b.

Esempio: Riconoscere la stringa “abcba” generata dalla grammatica UcU(r) con U ∊ (a U b)\*

[palindrome con marca di centro]

| PILA | INPUT | STATO | TRANSIZIONE | Spiegazione |
| --- | --- | --- | --- | --- |
| Z0 | abcba↙ | q0 [push(A)] | δ(q0,a,Z0)=(q0,Z0A) | Quando si è nello stato q0 e si è alla sinistra della marca di centro c, i caratteri vengono inseriti nella pila. |
| Z0A | bcba↙ | q0 [push(B)] | δ(q0,b,A)=(q0,AB) |
| Z0AB | cba↙ | q0 | δ(q0,c,B)=(q1,ε) | L’automa si trova sulla marca di centro, quindi si passa allo stato q1 e non si aggiunge niente alla pila. |
| Z0AB | ba↙ | q1 | δ(q1,b,B)=(q1,ε) | A questo punto si controlla se la cima della pila corrisponde al carattere che c’è sulla testina dell’input, se non è così l’automa si pianta. |
| Z0A | a↙ | q1 | δ(q1,a,A)=(q1,ε) |
| Z0 | ↙ | q1 | δ(q1,↙,Z0)=(q2,Z0) | Quando la pila è vuota e si ha il simbolo ↙, si passa a q2, lo stato di accettazione. |

Z0 indica la stringa vuota. L’automa a ogni passo esegue una Pop automatica prima della Push, quando bisogna inserire il carattere tolto insieme a quello nuovo. Nel caso delle palindrome senza marca di centro, la situazione si complica, l’automa infatti non sa in quale parte del frase si trova, di conseguenza prova tutte le opzioni possibili in quel momento.

Un automa a pila non deterministico è un <Q,Σ,Г,δ,q0,Z0,F>, dove Q è l’insieme di stati dell’automa, Σ e Г sono rispettivamente gli alfabeti della frase e della pila, Z0 indica la pila vuota e F l’insieme degli stati finali. La funzione di transizione δ, dato uno stato, un carattere e la cima della pila, restituisce una o più coppie (Q,Г\*)

δ: QxΣxГ →

Per descrivere il linguaggio accettato dall'automa a pile, serve il concetto di configurazione, essa è qualcosa da salvare per poi riprendere in un secondo momento, infatti vengono salvati lo stato, la stringa in input e il contenuto della pila. Una configurazione è descritta nel seguente modo:

C=<q,y,p> con q∊Q, y∊Σ\*, p∊Г\*

Esempio: data la configurazione C=<qi,ay,pA> e (qj,s)∊δ(qi,a,A) con s∊Г\*

una possibile mossa è <qj,y,ps> dove s è l’insieme di simboli che inserisco nella pila, con (qj,s)∊δ(qi,ε,A) invece si può ottenere <qj,ay,ps> dato che le ε-mosse non considerano l’input. Una particolare configurazione è quella iniziale, data una stringa y in input, la configurazione iniziale è (q0,y,Z0), da essa posso arrivare a un’altra configurazione, detta finale, in un certo numero di mosse, questo significa che la frase è stata riconosciuta:

<q0,y,Z0> →n\*<q,ε,p>

Il linguaggio L accettato dall’automa M è quindi l’insieme di stringhe y tali che a partire da una configurazione iniziale si arrivi a una finale con un certo numero di mosse:

L(M)={y \ <q0,y,Z0> →\*<q,ε,p>}

## Dimostrazione di contenimento tra automi a pila e linguaggi context-free

Come già accennato, gli automi a pila servono a riconoscere i linguaggi context-free, quindi a partire da un grammatica possiamo costruire l’automa e viceversa. Quindi, data una grammatica context-free G=<V,∑,P,S>, ricavare il rispettivo automa M=<Q,∑,Γ δ,q0,Z0>:

Gli automi costruiti in questo modo sono formati da un unico stato che è quello iniziale, quindi:

Q={q0}

L’idea alla base è sfruttare la pila per simulare la generazione della grammatica, quindi, se invece di mettere simboli a caso metto i terminali e i non terminali, è possibile simulare i passi della generazione. Di conseguenza, l’alfabeto della pila sarà formato dai simboli terminali e non terminali più Z0:

Γ=V U ∑ U {Z0}

Per ogni regola di P, se è nella forma A → X1,...,Xn, δ(qo,ε,A) porta a uno stato in cui X1,...,Xn sono rovesciati all’interno della pila:

∀A→∝∊P, se A→ X1,...,Xn, allora (q0,Xn,...,X1)∊δ(qo,ε,A)

Quando il primo simbolo è un terminale, esso verrà messo in cima alla pila e poi si effettua la verifica su di esso, si può comunque effettuare tutto questo in una sola passata: si inserisce una regola in cui, per ogni regola A → a∝ con a terminale, la coppia (q0,∝(r)) appartiene alla transizione δ(qo,ε,A) (in poche parole, la regola scatta quando vi è un terminale all’inizio e quindi effettua la verifica su esso senza metterlo in pila).

∀A→a∝∊P con a∊∑, (q0,∝(r))∊δ(qo,a,A) se A→ aX1,..,Xn

Vi è anche una seconda regola oltre la prima in cui, per ogni regola A→ X∝ con X non terminale, la coppia (q0,∝(r)X) appartiene alla transizione δ(qo,ε,A) (in parole povere la regola scatta e non verifica nulla, serve solo a gestire i simboli non terminali).

∀A→X∝∊P con X∊V, (q0,∝(r)X)∊δ(qo,ε,A)

Il controllo dell’input quando è presente un terminale in cima alla pila viene effettuato secondo la seguente regola: per ogni a terminale in cima alla pila, se corrisponde al terminale sulla testina dell’input si va avanti, altrimenti si pianta.

∀ a ∈ ∑, δ(q0, a, a)={(q0, ε)}

Vi sono poi regole che definiscono l’inizializzazione e la terminazione: la prima indica che si inizia mettendo lo start symbol in cima alla pila insieme a ↙ per indicare la fine della stringa.

δ(q0, ε, Z0) = {(q0, ↙S)}

La seconda invece indica che, se ho ↙ in cima alla pila e il carattere della testina è ↙, allora non inserisco più niente nella pila,di conseguenza l’accettazione avviene quando quest’ultima è vuota.

δ(q0, ↙, ↙) = {(q0, ε)}

Esempio: costruire l’automa a pila dalla seguente grammatica:

S→ aS

S→ A

A→ aAb

A→ ab

Inizializzazione: (q0,↙S)∊δ(q0, ↙,↙)

Terminazione: (q0,ε)∊δ(q0, ↙,↙)

Com’è possibile notare, la prima regola ha un non terminale all’inizio, quindi:

(q0,S)∊δ(q0, a, S)

lo stesso discorso vale anche per la terza e la quarta:

(q0,bA)∊δ(q0, a, A)

(q0,b)∊δ(q0, a, A)

Per quanto riguarda la quarta regola, invece, non vi sono terminali all’inizio e quindi non si aggiunge niente in pila:

(q0,A)∊δ(q0, ε, S)

Infine si aggiungono le regole in cui vi è corrispondenza tra la cima della pila e la testina, in tal caso non si aggiunge niente alla pila:

(q0,ε)∊δ(q0, a, a)

(q0,ε)∊δ(q0, b, b)

L’automa è pesantemente non deterministico perchè i rami con a li prendo solo quando ho a in cima alla pila mentre quelli con ε li prendo in ogni caso (sia con a che con b in questo caso), quindi è consigliabile costruire un possibile albero di derivazione della frase per decidere le regole da utilizzare e applicarle per il riconoscimento.

Esempio: “aaabb”

| Pila | Input | Stato | Transizione |
| --- | --- | --- | --- |
| Z0 | aaabb↙ | q0 | regola 1 |
| ↙S | aaabb↙ | q0 | regola 3 (4) |
| ↙S | aabb↙ | q0 | regola 4 (3) |
| ↙A | aabb↙ | q0 | regola 6 (5) |
| ↙bA | abb↙ | q0 | regola 5 (6) |
| ↙bb | bb↙ | q0 | regola 6 |
| ↙b |  |  |  |
| ↙ | ↙ |  | regola 2 |

## 

## 

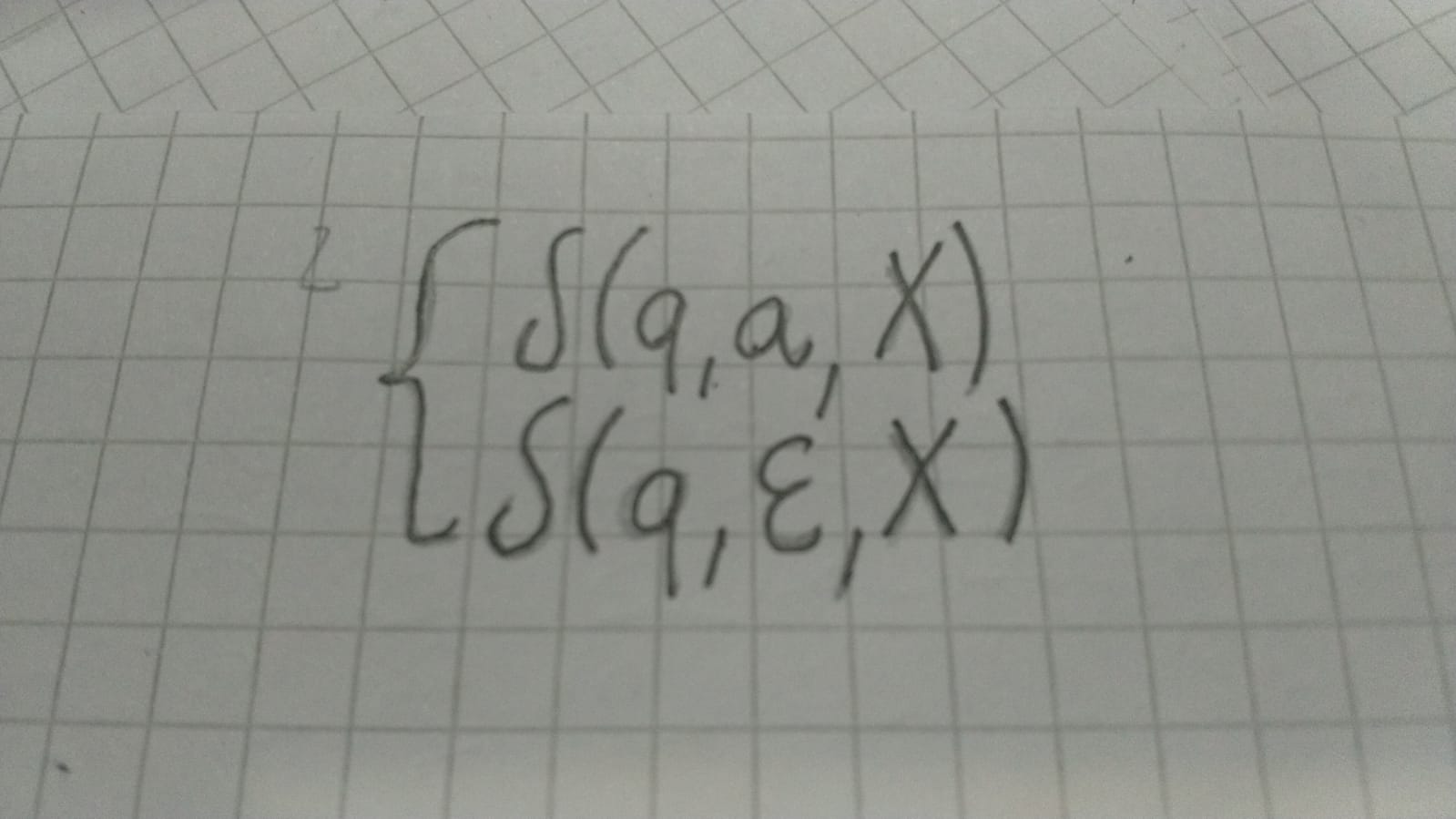
## Automi a pila deterministici

Gli automi a pila deterministici sono particolari automi che permettono il riconoscimento di linguaggi deterministici, questi ultimi sono linguaggi strettamente contenuti nei context-free.

La classe dei linguaggi deterministici è infatti la classe contenente tutti i linguaggi riconosciuti dagli automi deterministici, questi ultimi impiegano un tempo lineare per riconoscerli,fanno parte di questo insieme i linguaggi regolari dato che sono riconoscibili da un automa a stati finiti (un caso particolare dell’automa a pila). Gli automi a pila non sono come quelli a stati finiti: non esiste una forma deterministica per ogni automa non deterministico. Un automa è detto deterministico quando, per ogni stato q, simbolo a dell’alfabeto e simbolo della pila, è possibile andare in un solo stato:

∀q, ∀a, ∀X, |δ(q,a,X)|<=1

In più per uno stesso stato q, se è definito per ε, esso non è definito per i simboli a e viceversa.



Ultima regola ma non meno importante, una definizione definita per ε deve restituire un solo stato.

|δ(q,ε,X)|<=1

I linguaggi ambigui non sono deterministici dato che vi sono più possibili modi per arrivare a un risultato (andando contro la definizione stessa di determinismo). Un esempio sono le palindrome senza marca di centro perchè con 2 simboli uguali vicini devo considerare sia che quella coppia sia il centro della frase, sia il caso in cui non lo sia. Per riconoscere una palindroma, un automa deve impilare la frase fino a metà e disimpilarla poi fino alla fine, quindi si suppone che qualcosa dica che l’automa è a metà della frase: considerando la frase “abba” da riconoscere, con la prima la prima a e la prima b l’automa continua a impilare dato che non è a metà, con la seconda b però l’automa non sa se è a metà o meno, quindi deve considerare entrambe le opzioni. Le palindrome senza marca di centro sono un linguaggio non ambiguo, infatti per generarlo è necessaria una grammatica non ambigua:

S→ aSa|bSb|aa|bb|a|b

L’unico modo per capire se un linguaggio è deterministico è trovare almeno un automa deterministico che lo riconosce dato che non c’è una caratterizzazione a livello di grammatica, se invece un linguaggio è non deterministico bisogna dimostrare che non esiste alcun automa non deterministico in grado di riconoscerlo.

Consideriamo ora il seguente esempio: dato il linguaggio deterministico con n>=0, si vuole costruire l’automa deterministico che lo riconosce. Partendo dallo stato q0, ogni volta che prendo una a, devo metterla in coda, quindi:

δ(q0,a,Z0)=(q0,Z0A)

δ(q0,a,A)=(q0,AA)

Quando trovo una b, significa che sono nella seconda metà della frase, di conseguenza si passa allo stato q1 e non si inserisce niente in pila (si ricorda che a ogni iterazione c’è una pop automatica):

δ(q0,b,A)=(q1,ε)

δ(q1,b,A)=(q1,ε)

Infine, quando in input vi è ↙️ e la pila è vuota, si passa allo stato q2, lo stato di accettazione:

δ(q1,↙️,Z0)=(q2,Z0)

L’automa risultante sarà quindi il seguente:

<Immagine automa deterministico>

La proprietà di chiusura rispetto all’operatore non vale per questo tipo di automi, infatti anche se il complemento di un linguaggio è deterministico, l’unione o l’intersezione di due linguaggi deterministici potrebbe non esserlo. Tuttavia vale la seguente proprietà: dati linguaggi L e R in cui L è deterministico e R regolare, L/R è deterministico: in poche parole la classe del linguaggio non cambia si si aggiungono dei simboli all’inizio o alla fine della stringa.

Per riassumere:

| Linguaggio | Regolari | Context-free | Deterministici |
| --- | --- | --- | --- |
| Formule | espressioni regolari |  |  |
| Grammatiche | grammatiche lineari destre/sinistre | grammatiche context-free |  |
| Automa | automa a stati finiti | automa a pila non deterministico | automa a pila deterministico |

A livello di grammatica la differenza tra linguaggi regolari e context-free è la produzione ricorsiva auto inclusiva, rappresentabile come un albero con asse di crescita centrale, a livello di automa invece la pila viene utilizzata per i bilanciamenti. Tra i linguaggi context-free e deterministici non vi è alcuna differenza a livello di grammatica, tuttavia a livello di automa è il non determinismo stesso, infatti con i secondi non ho bisogno di fare delle “strade alternative” per riconoscere una frase.

## Parsing

Per tradurre un linguaggio non basta saperlo generare o riconoscere, bisogna dare anche la sua derivazione attraverso l’albero di derivazioni che ne dà la struttura. Tutte le classi di linguaggi sono parsificabili, quello che ci interessa però è effettuare il parsing su linguaggi in modo efficiente, ovvero sui linguaggi deterministici. Le strategie principali di parsing sono:

* Top-down;
* Bottom-up

### Top-down

Top-down è una tecnica in preordine in cui si deriva a partire dallo start symbol costruendo l’albero di derivazione, quest’ultimo si basa sulla grammatica e non sul linguaggio.

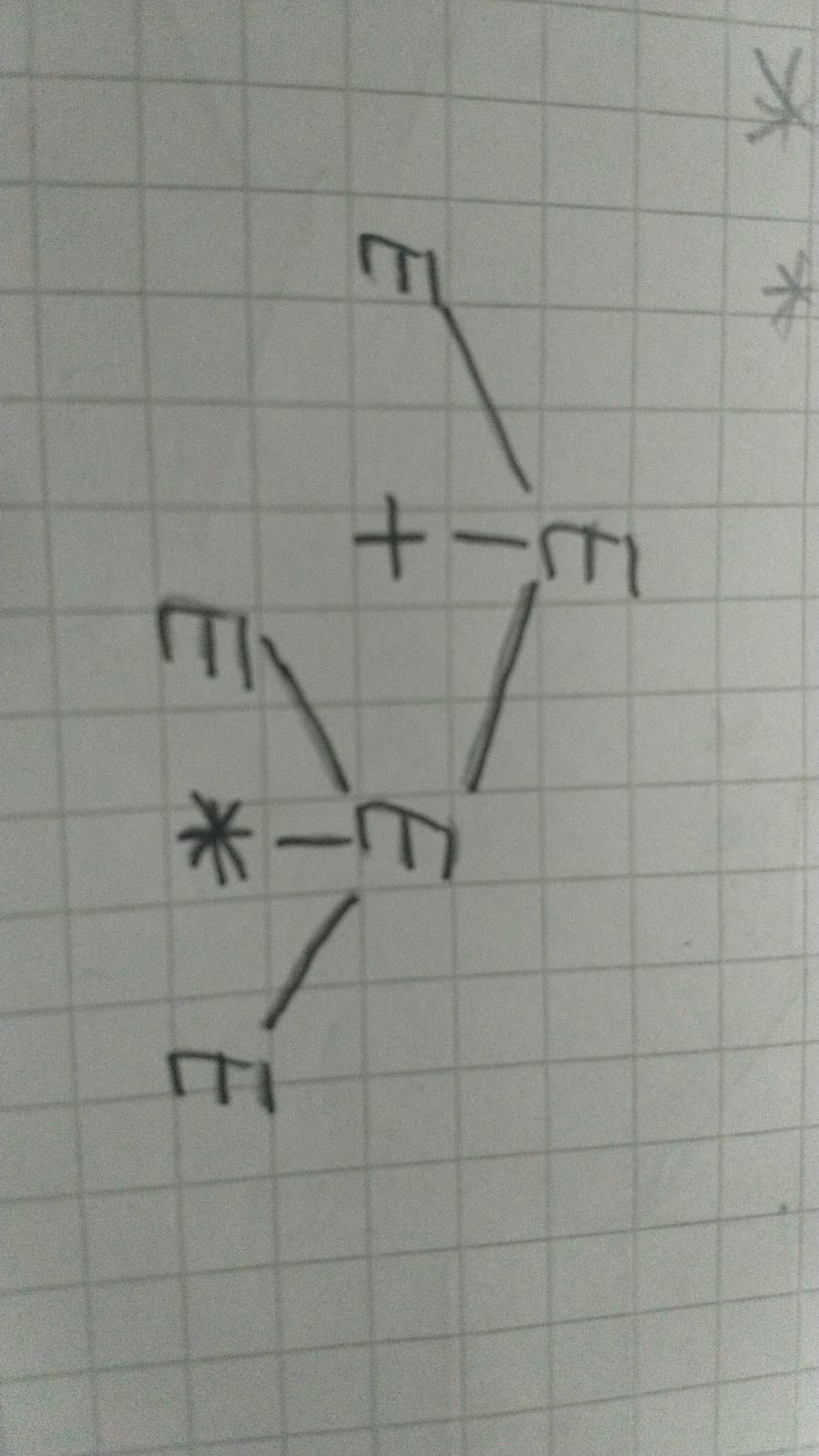
Esempio: data la grammatica E→ E+E|E\*E|(E)|id, si vuole fare il parsing di id+id\*id. Questo tipo di parsing utilizza la cosiddetta derivazione left-most, ovvero che a partire dallo start symbol si costruisce l’albero propagandosi verso il basso, quindi generandolo in preordine.

La tecnica top-down seleziona passo-passo le derivazione, in questo esempio sceglie prima E+E, poi deriva la prima E in id e la seconda in E\*E e queste ultime sfociano in id.

Questo tipo di parsing può andare avanti guardando tendenzialmente un simbolo in input oppure non considerarli affatto, questo è un problema perchè avere un solo simbolo senza sapere cosa c’è dopo rende complicato decidere cosa fare. Questo tipo di parsing è utilizzato perché facile da costruire, tuttavia sono meno potenti dei bottom-up dato che non si riescono a trovare tutti i linguaggi deterministici, questo perché bisogna scegliere subito la strada da prendere.

### Bottom-up

Bottom-up è una tecnica che esegue il parsing dal basso verso l’alto, quindi l’albero di derivazione viene costruito partendo dalle foglie e risalendo quando si è visualizzato l’input, essa utilizza quindi una derivazione right-most al contrario. Utilizzando il precedente esempio, l’albero di derivazione si costruisce guardando il simbolo in input corrente e le regole della grammatica: da id infatti posso risalire al simbolo E, questo passo è detto riduzione. Ora il considera il +, con esso non si può fare niente dato che non vi sono regole che corrispondono nella grammatica, quindi si passa al successivo. Una possibile passo di riduzione è E+E ma per evitare ambiguità supponiamo che \* abbia più priorità di +, quindi si vanno a ridurre tutti gli id in E. la frase ora è E+E\*E, quindi è possibile ridurre E\*E in E e il ricavato E+E anch’esso in E, l’albero di derivazione è ora concluso.



Dall’esempio precedente è possibile ricavare le seguenti azioni:

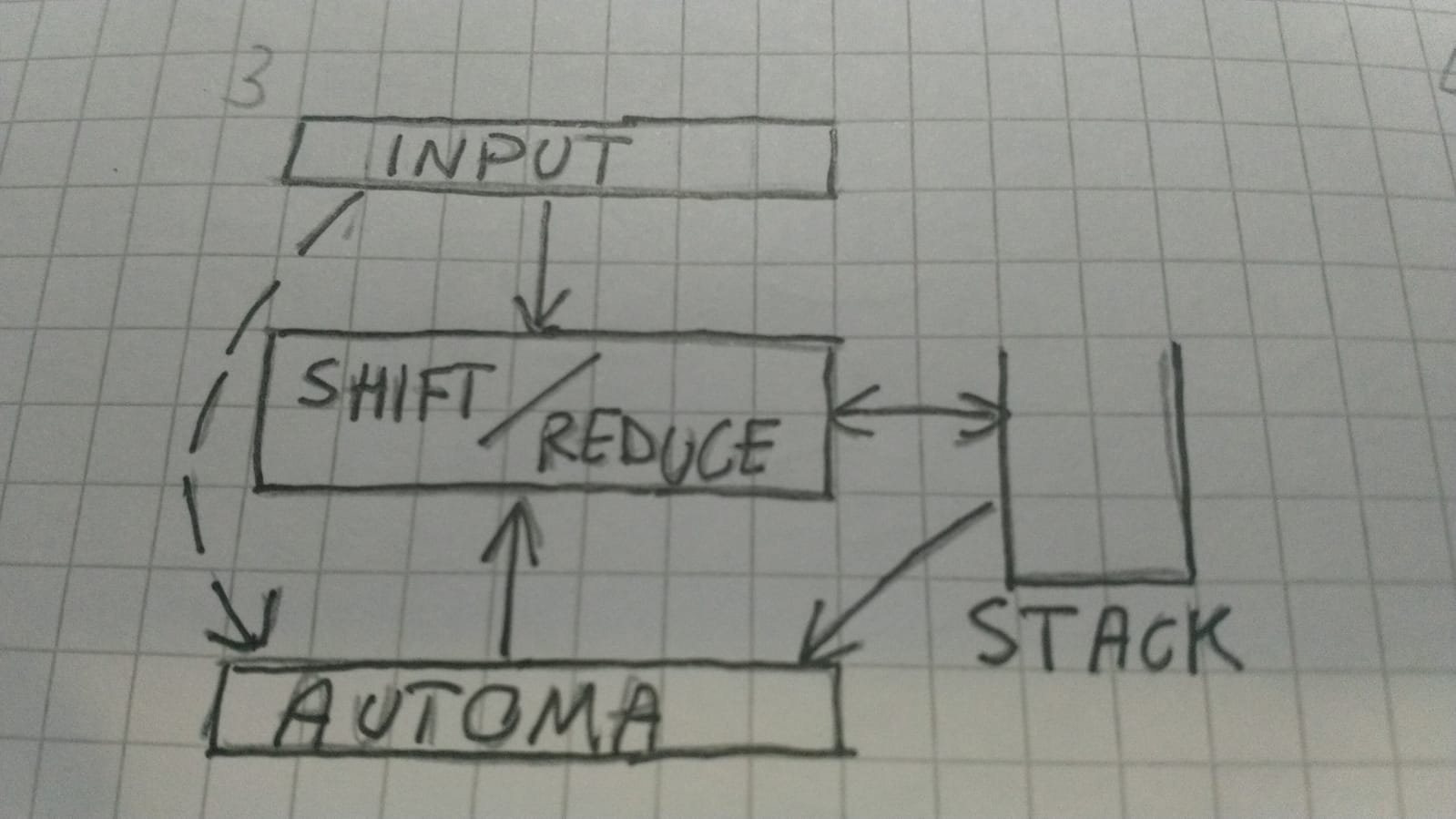
* Shift: spostamento di un simbolo al livello superiore dell’albero
* Reduce: riduzione di una grammatica al non terminale che lo genera.

Una grammatica è riducibile quando è presente almeno una regola che permette di risalire nell’albero di derivazione, tuttavia questo non basta ed è necessario sapere se il passo di riduzione va bene o meno, per esserlo infatti deve essere un passo della riduzione right-most al contrario. Una parte riducibile di una forma sentenziale destra γ è:

* una produzione A → β;
* una posizione di γ dove c’è la forma sentenziale β e la riduzione A → β produce un’ulteriore forma sentenziale destra precedente nella derivazione right-most di γ, ovvero: S --(rm)\*-->αAw--(rm)--> αβw

Una forma sentenziale destra è un misto di terminali e non terminali generabili da una derivazione right-most. L’automa che genera l’albero bottom-up è un automa a pila, esso è basato sulle operazioni di Shift e Reduce e prende appunto il nome di Parser Shift-Reduce.

Di default il parser esegue un’operazione di Shift (prende il primo simbolo in input e lo mette in pila) fino a quando non vi è una parte riducibile in cima alla pila, in tal caso di fa la reduce.



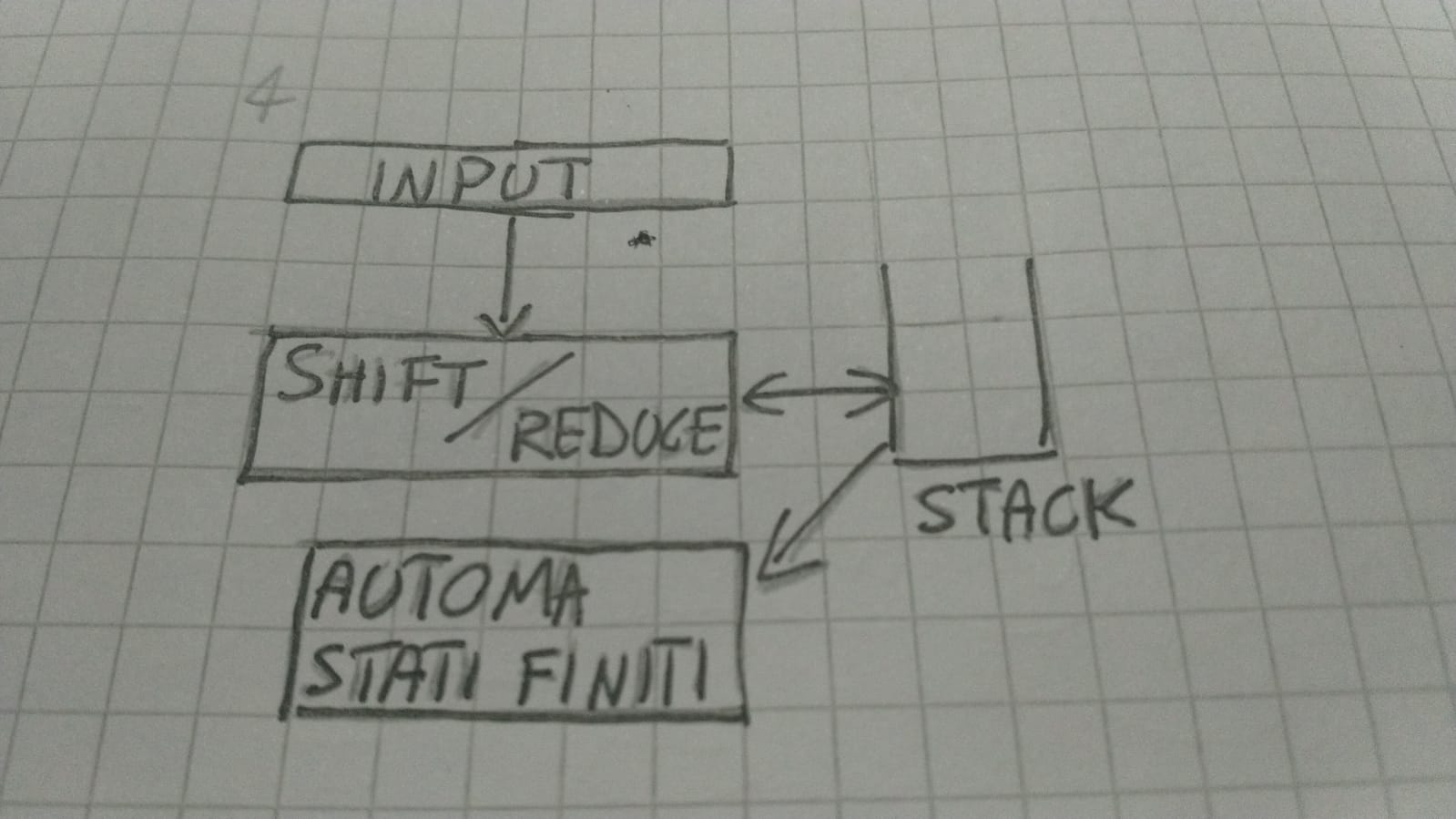
Il parser shift-reduce è anche detto LR(n), dove L indica l’input, R la derivazione right-most al contrario e n è il look ahead, ovvero il numero di simbolo da leggere dall’input prima di decidere l’azione da eseguire.

Per scegliere l’azione da eseguire, il parser utilizza un automa, esso è a stati finiti e, a seconda dello stato in cui si trova, si esegue una determinata azione (il linguaggio delle forme sentenziali destre è quindi regolare), vi possono essere però dei conflitti:

* Shift-Reduce: vi sono condizioni che permettono sia l’operazione di shift che quella di reduce;
* Reduce-Reduce: vi sono più condizione che permettono l’operazione di Reduce su differenti forme sentenziali.

### Parser LR(0)

Il parser LR(0) è un caso particolare dello shift-reduce in cui non si guarda nessun simbolo in input, l’automa può quindi fare shift sull’input e impilarlo oppure non controllarlo e fare una reduce, il linguaggio delle pila è una forma sentenziale destra.



Il linguaggio da analizzare è quello riconosciuto dall’automa a stati finiti, tuttavia non interessa riconoscerlo bensì comunicare al parser l'operazione da eseguire. Per costruirlo, l’idea è riconoscere sia una parte riducibile, sia i suoi prefissi, solo essi infatti possono stare in cima alla pila del parser.

Esempio: S→ aSa|b

In questo esempio, aSa può benissimo stare in cima alla pila, aaSaa invece no perchè ha bisogno di una riduzione.

Per descrivere i prefissi bisogna guardare la grammatica, la riduzione infatti si fa utilizzando le parti destre di quest’ultima, per farlo si utilizza il candidato LR(0), una produzione della grammatica più il simbolo ˇ il quale indica dove si è arrivati. Nel caso dell’esempio precedente, i possibili candidati sono:

* S→ ˇaSa
* S→ aˇSa
* S→ aSˇa
* S→ aSaˇ

Tutto ciò che è prima di ˇ si trova in cima alla pila. Per ricostruire le forme sentenziali destre bisogna unire la nozione di candidato con quella di prefisso ascendente, il candidato LR(0) valido per una famiglia di forme sentenziali è quindi un candidato LR(0) che permette il riconoscimento di più forme sentenziali di una data grammatica.

Esempio: S→ aSa|b

supponendo di avere 3 a in cima alla pila, è possibile proseguire aggiungendo ulteriori a ma non aggiungendo una b, dato che bisogna prima inserire S. Da questo esempio possiamo quindi concludere che, data una forma sentenziale, è possibile proseguire solamente coi candidati LR(0) validi per essa.

Un candidato LR(0) si può scrivere in generale nel seguente modo:

A→ β1 ˇ β2 con β1,β2 ∈ (V U Σ)\*, ovvero che sono forme sentenziali.

A è valida per un prefisso ascendente se e solo se esiste una derivazione del tipo:

S--(rm)\* → αAw --(rm)\* →αβ1β2w

Essendo una riduzione right-most, non possono esserci non terminali in w, se così non fosse la riduzione dovrebbe prendere in considerazione anche quelli.

Ogni stato dell’automa deve riconoscere un gruppo di prefissi ascendenti, per descriverlo si utilizzano tutti i candidati validi per essi. Sulla base di questo, dato un prefisso, è possibile ricavare tutti i candidati validi guardando la grammatica. La costruzione dell’automa avviene anche attraverso due funzione ausiliarie: la chiusura e la goto.

#### Chiusura

Dato un candidato nella forma A→ αˇBβ in cui B è non terminale e α e β sono forme sentenziali, tutte le regole di B diventano possibili candidati LR(0), questa regola viene applicata ricorsivamente perchè se il primo simbolo a destra di ˇ è non terminale, allora bisogna aggiungere anche le regole della grammatica di quest’ultimo finchè non si finisce.

Esempio: B→ Xb|c, X→ Ya|a, Y→ c|b

A partire dallo stato B, si aggiungono le regole della grammatica:

B→ ˇXb

B→ ˇc

Nella prima regola è possibile notare che dopo ˇ vi è un terminale, quindi si aggiungono le sue regole:

X→ ˇYa

X→ ˇa

anche qui vi è un non terminale dopo ˇ nella prima regola:

Y→ ˇc

Y→ ˇb

In parole semplici, la funzione di chiusura segue il seguente ragionamento:

1. ogni candidato i nella regola I→ i appartiene alla chiusura del non terminale I;

∀ candidato i ∈ I → i ∈ closure(I)

1. ogni candidato a nel candidato αˇBβ appartiene alla chiusura di I, per ogni produzione di B nella grammatica, B→ ˇγ appartiene alla chiusura di I.

∀ candidato a ∈ α ^ B β ∈ closure(I)

∀ produzione B → γ ∈ G

B → ^ γ ∈ closure(I)

### Goto

La funzione di goto(I,X) permette la transizione da uno stato all’altro. Supponendo di aver definito uno stato contenente il candidato A→ αˇXβ con X terminale o meno e α e β come forme sentenziali, esso sarà valido per un dato prefisso ascendente α'α, quindi ci può essere un caso in cui quest’ultimo sia in cima alla pila e, di conseguenza, c’è una regola A→α'αβ che permette di continuare. Si può quindi dire la seguente regola:dato uno stato I e un simbolo X, per ogni candidato LR(0) di I nella forma A→αˇXβ, la chiusura di A→αXˇβ appartiene al goto(I,X).

∀A→αˇXβ ∈ I, closure({A→αXˇβ}) ∈ goto(I,X)

### Innesco del processo

Il primo prefisso ascendente banale è ε e quindi il primo stato deve riconoscerlo, questo però non è indispensabile per innescare il processo. Per semplificare una grammatica, si aggiunge una regola in cui viene messo ↙️ alla fine, ad esempio se ho S come start symbol della grammatica, la regola da aggiungere è S’ → S↙️. Il primo stato ha quindi come primo candidato S’ → ˇS↙️ e di esso si fa la chiusura da esso partono tutti i possibili goto verso altri stati finchè è possibile:

Finchè non si possono più aggiungere stati

∀I ∈ C

∀X ∈ (V∪∑) tale che goto(I, X) =∅

C=C ∪ {goto(I, X)}

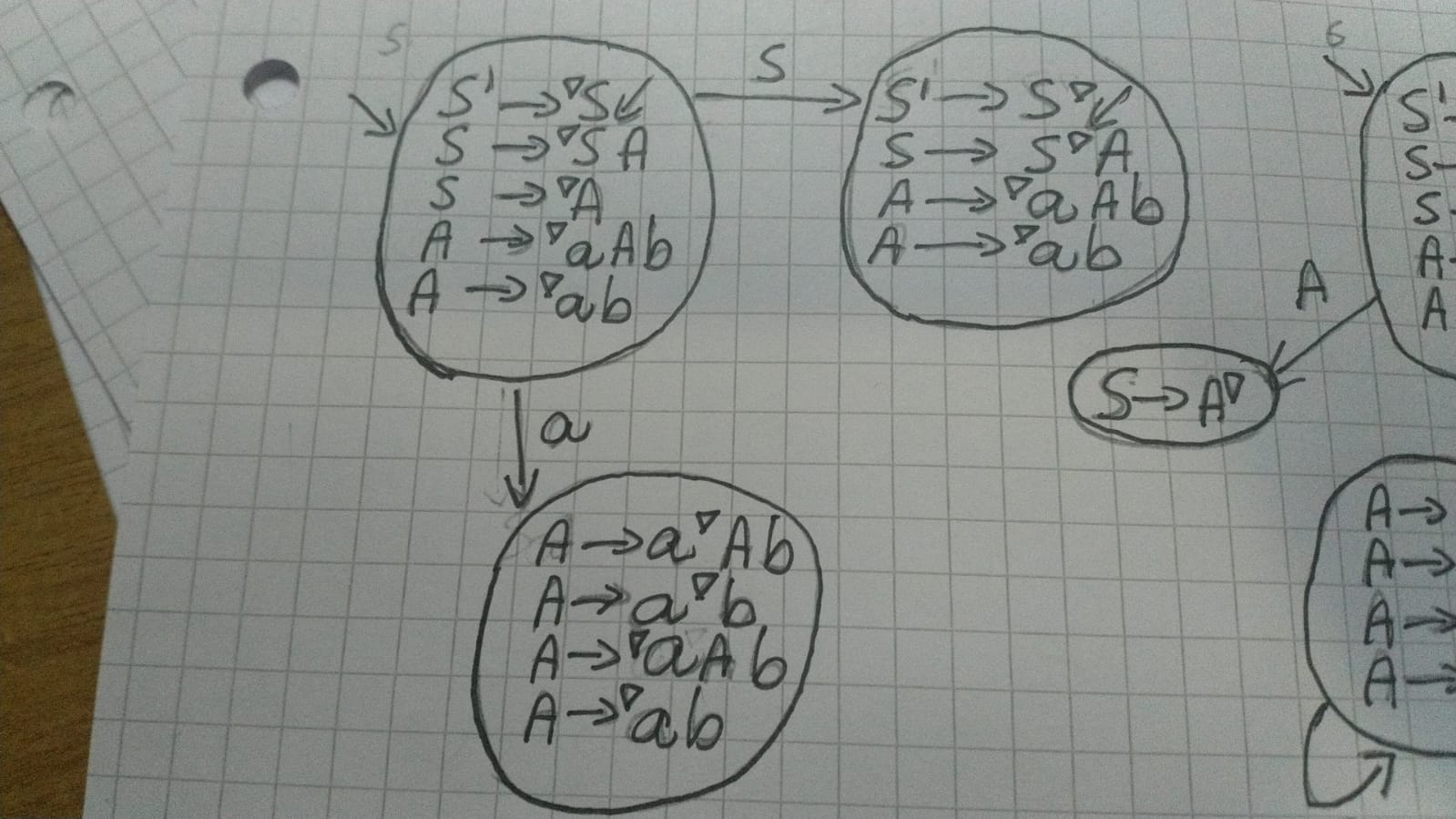
Esempio: a partire dalla seguente grammatica, si vuole costruire l’automa:

S'→S↙

S →SA|A

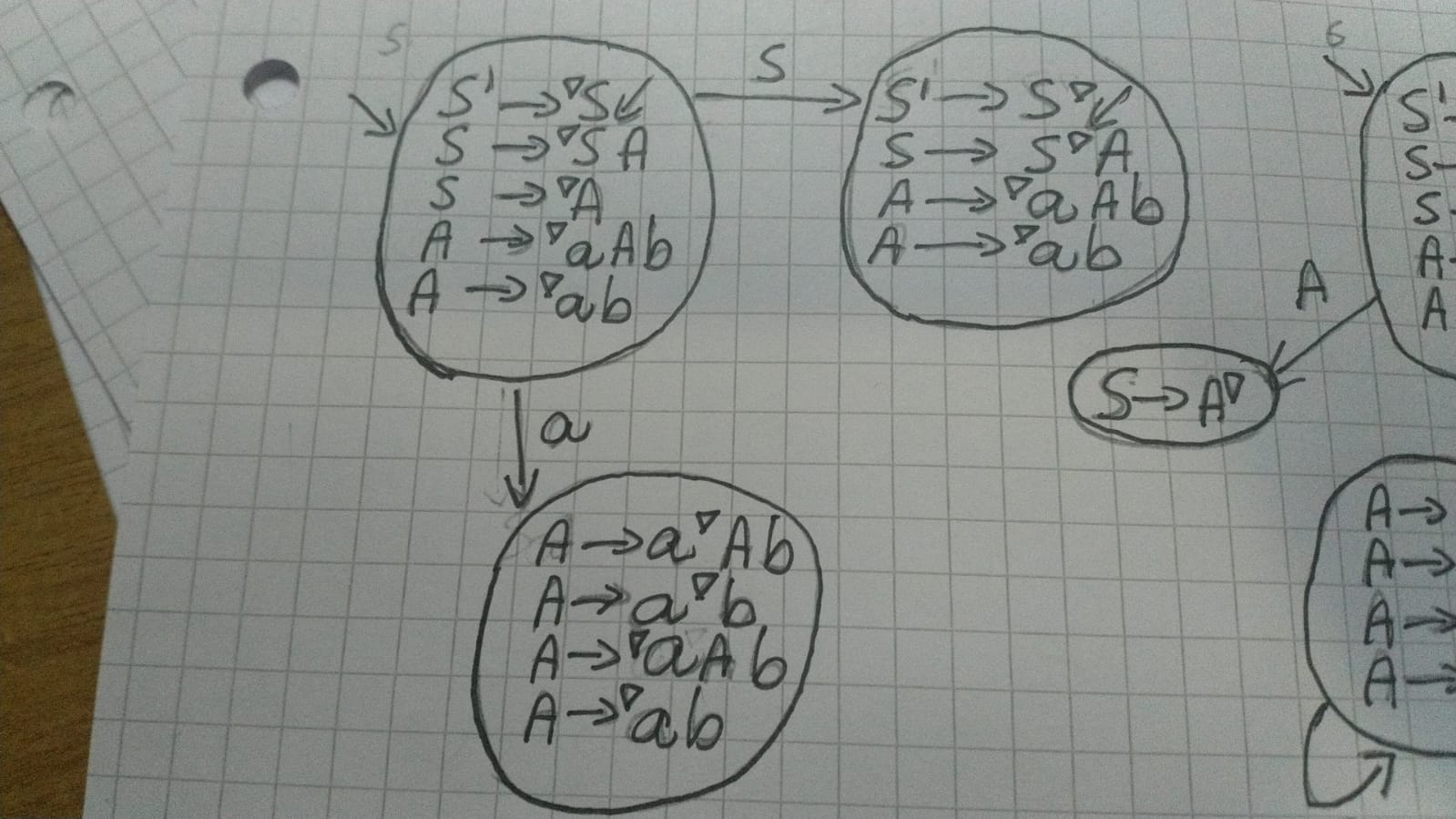
A →aSb|ab

Il primo candidato è S’ → ˇS↙️ e quindi si fa la chiusura: S→ˇSA e S→ˇA diventano altri candidati. Come si può vedere dalla regola S→ˇA, bisogna aggiungere anche la chiusura di quest’ultimo: quindi si aggiungono le regole di A come possibili candidati. Lo stato iniziale può quindi essere riassunto nella seguente immagine:

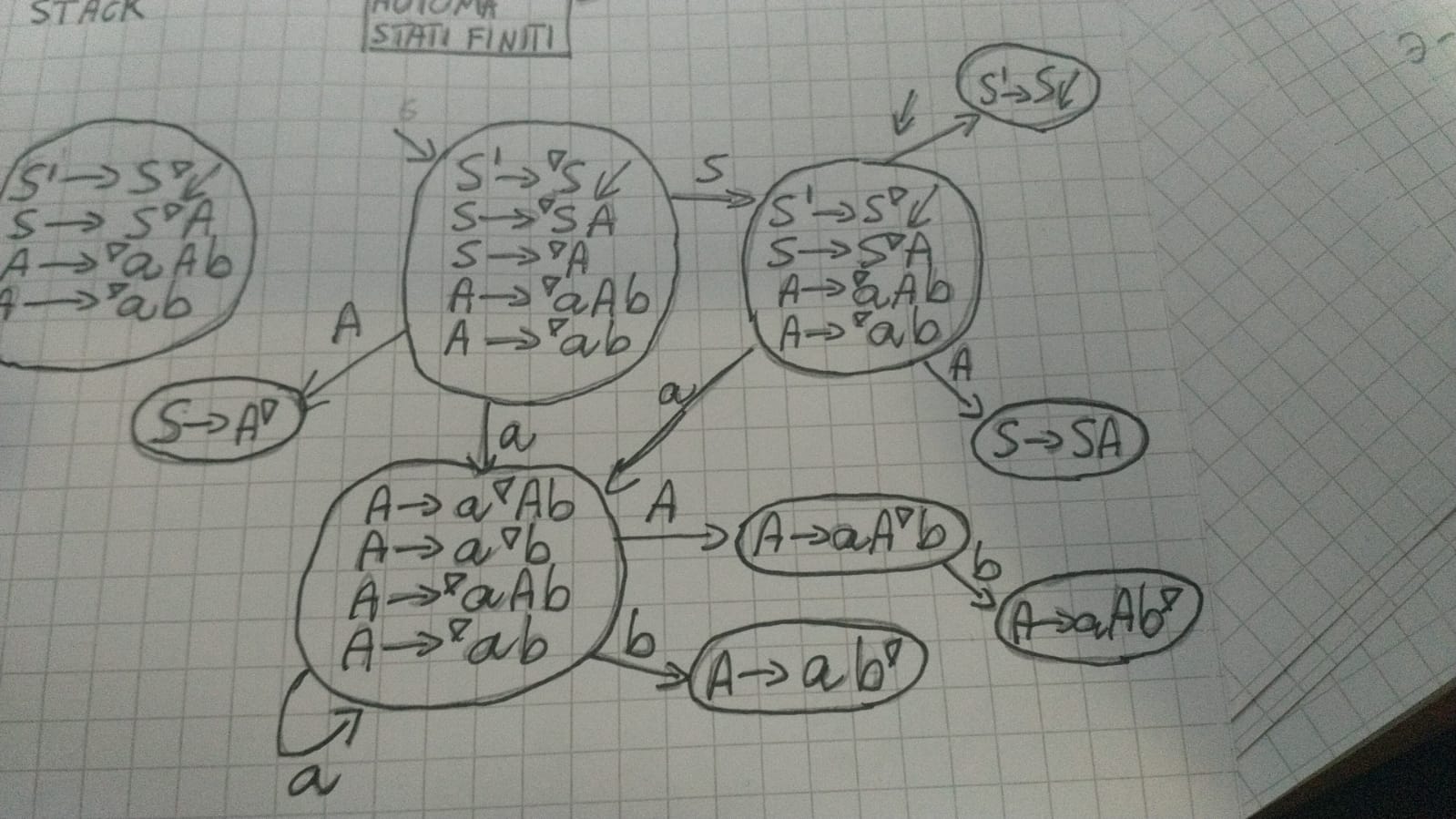


A questo punto bisogna rilevare i possibili goto: per S vi sono le prime due regole aggiunte mentre per a le ultime due, si hanno quindi due nuovi stati, il primo contenente S’ → Sˇ↙, S→ SˇA e la chiusura di quest’ultimo (dato che vi è un non terminale alla destra di ˇ) mentre il secondo contiene A→ aˇSb, la relativa chiusura e A→ aˇb.

Si ha quindi una situazione del genere:



Continuando ad aggiungere finchè è possibile, si ottiene il seguente automa:



Ora che l’automa è fatto, si analizzano gli stati dell’automa: gli stati in cui un candidato ha ˇ alla fine sono etichettati come stati di reduce, tutti quelli con un candidato con ˇ da qualsiasi parte ma non alla fine sono stati di shift. Tuttavia vi possono essere conflitti sia di tipo shift/reduce, sia reduce/reduce (spiegati in precedenza). In un caso in cui l’automa non presenta conflitti, in base allo stato in cui si trova, il parser controlla l’etichetta ed esegue l’azione appropriata.

Esempio: dato l’automa costruito in precedenza, fare il parsing della frase “aaabbb↙️”

| Pila | Input | Azione |
| --- | --- | --- |
| 0 | aaabbb↙️ | shift |
| 0a2 | aabbb↙️ | shift |
| 0a2a2 | abbb↙️ | shift |
| 0a2a2a2 | bbb↙️ | shift |
| 0a2a2a2b7 | bb↙️ | reduce A→ ab |
| 0a2aA6 | b↙️ | shift |
| 0a2aA6b8 | ↙️ | reduce A→ aAb |
| 0aA6 | ↙️ | shift |
| 0aA6b8 | ↙️ | reduce A→ aAb |
| 0A3 | ↙️ | reduce S→ A |
| 0S1↙️4 |  | reduce S’ → S |
| 0S’ |  |  |

### 

### Definizione formale di automa deterministico

Un automa a pila deterministico è una quintupla <Q,∑ ∪ V, δ,0,Q>, esso è formato dal solo stato iniziale q0 e come alfabeti utilizza ∑ e Г, quest’ultimo è formato dall’alfabeto ∑, dall’insieme di stati Q e da quello di non terminali V. La configurazione iniziale dell’automa ha il simbolo 0 nella pila, quello finale ha invece la pila vuota. la funzione di transizione δ' si basa su δ e, dato che c’è sempre uno stato dell’automa in cima alla pila dopo ogni operazione, vi possono essere due casi:

* se in cima vi è uno stato J etichettato come shift, la transizione δ'(pila, a, Ij) è uguale a (pila, JaK) se la transizione δ(J,a)=K;
* lo stato in cima alla pila è etichettato come reduce.

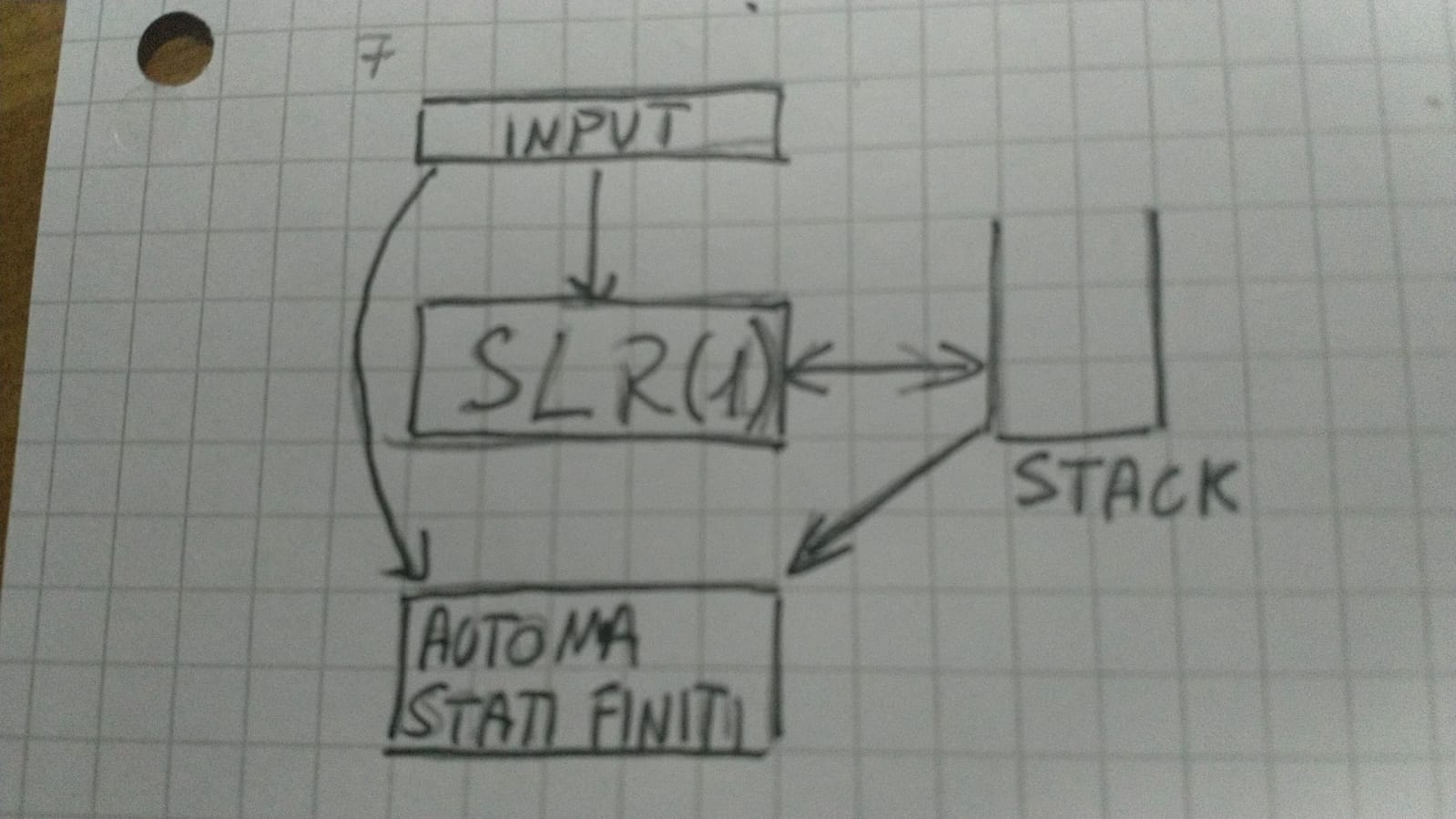
Non tutti i linguaggi LR(0) sono deterministici, quelli con prefissi ne sono un esempio:

S’--> S↙️, S→ aS|ε

In questo esempio si ha come possibile candidato S→ˇ e, dato che vi sono anche candidati che generano shift, si ha un conflitto shift/reduce. Questi linguaggi sono analizzabili con l’automa LR(1) oppure con una sua versione più semplice: SLR(1) (Simple LR(1)).

## Parser SLR(1)

Come trattato in precedenza, LR(0) permette il parsing dell’input a seconda dello stato dell’automa, controllando la pila è infatti possibile in stati etichettati come shift/reduce e quindi agire di conseguenza. L’automa tuttavia ripete molte volte lo stesso stato per dati simboli in input, quindi si è deciso di utilizzare LR(1), esso controlla anche un simbolo in input per decidere quando effettuare una shift o una reduce. Una versione più semplice è SLR(1), esso prende in eredità ciò che caratterizza LR(0) e aggiunge informazioni utili quando bisogna effettuare delle riduzioni, si rappresenta nel seguente modo:



Per prima cosa si costruisce l’automa a stati finiti come in LR(0), dopo si arricchiscono le riduzioni con dei simboli terminali, essi rappresentano il contesto di riduzione della testina dell’input (detto anche follow o insieme di prospezione, ovvero tutti i possibili terminali che seguono un dato non terminale). Le informazioni aggiuntive vengono utilizzate quando ci sono conflitti in uno stato dell’automa, infatti se il carattere controllato corrisponde a uno di quelli del follow, si fa la reduce, altrimenti si fa uno shift. Nel caso in cui vi siano conflitti reduce/reduce, i due insiemi non devono avere terminali in comune.Una grammatica è quindi analizzabile in modo deterministico da SLR(1) quando non ci sono conflitti, ovvero che è possibile sapere quando fare una shift e quando una reduce. Per calcolare il follow di un terminale, consideriamo il seguente esempio:

A→ αXβ

Il follow di X è tutto ciò che ci si aspetta dopo la riduzione, supponendo che β→ x1|...|xn, la parte destra della regola indica il first, ovvero l’insieme di terminali e non terminali che si trovano prima di un dato non terminale, se il simbolo xi è terminale, allora la forma sentenziale inizia con xi, altrimenti ci sarà una regola per esso e quindi si esegue il first dei non terminale più interni (essi sono contenuti nel first più esterno). Nel caso il non terminale xi produca ε, si passa al successivo, in tal caso vi possono essere due situazioni:

* se passando al successivo trovo un terminale, esso diventa il first e quindi finisco;
* se tutti i non terminali generano ε, allora il first del non terminale è ε;

### Algoritmi di first, FIRST e follow

#### first

Il ragionamento del first è stato spiegato nel precedente paragrafo: in poche parole:

* se trovo un simbolo terminale, esso è il first del non terminale dato;
* se trovo un simbolo non terminale, il first è formato da tutti i FIRST interni (ε viene esclusa), se il non terminale genera la stringa vuota, passo al successivo, altrimenti l’algoritmo si ferma;
* Se tutti i simboli sono non terminali e possono tutti generare ε, allora ε fa parte della chiusura;

L’algoritmo è il seguente:

first(X1,...,Xn)

f=Ø, i=1,more=true;

while(more)

if(i>n)

f=f U {ε}

more=false

else(Xi ∈ ∑)

f=f U {Xi}

more=false

else

f=f U FIRST(Xi) - {ε}

if(ε ∈ FIRST(Xi))

i++

else more=false

return f

#### FIRST

FIRST

∀A ∈ V, FIRST(A)=Ø

∀A ∈ V,∀A→ α ∈ P

FIRST(A)=FIRST(A) U first(α)

#### FOLLOW

L’algoritmo del follow funziona nel seguente modo: quando un terminale A appare più volte in una regola, esso viene considerato più volte, se invece la regola è nella forma B→ αAβ, tutto quello che può seguire B, può anche seguire A.

FOLLOW

∀A ∈ V,FOLLOW(A)=Ø

∀ occorrenza di A ∈ B→ αAβ

FOLLOW(A)=FOLLOW(A) U first(β) - {ε}

### Parser LR(1)

Il parser LR(1) è un tipo di parser in cui, oltre a controllare cosa c’è in pila, l’automa controlla anche un solo simbolo in input, qui l’insieme di prospezione entra a far parte della descrizione dell’automa, ogni candidato ha infatti un’insieme di simboli che si aspetta dopo la riduzione. Un candidato LR(1) è valido per un prefisso ascendente se α’α se, dato il candidato A→ αβ {a1,...,an}, è possibile la derivazione right-most dello start symbol in αAw con un numero incerto di passaggi, una seconda derivazione right-most serve a sostituire i non terminali:

S--(rm)\*→αAw--(rm)→ αβw

Può capitare che due stati siano uguali perchè hanno gli stessi candidati con gli stessi insieme di prospezione, tuttavia non ha importanza perchè l’insieme di prospezioe permette di capire quale candidato utilizzare, per questo motivo LR(1) è più potente di LR(0) anche se funziona più o meno allo stesso modo. QUesto approccio è molto utilizzato dal parser LALR(1), un’ ottimizzazione che permette l’accorpamento di più stati senza perdere capacità di riconoscimento.