AUTOMA DK

Ad ogni CFG $\mathcal{G}$, corrisponde un DFA detto AUTOMA DK, tale che accetti l'input z se e solo se:

(1) z è PREFISSO di una certa stringa valida $v = zx$;

(2) z termina con un HANDLE di v.

DK riconosce gli handles delle stringhe valide di $\mathcal{L}(\mathcal{G})$. Ogni ACCEPT STATE di DK indica quali sono le regole di riduzione che si applicano per quell'handle.

* Quindi, DK ci permette di capire se una CFG è o meno una DCFG!

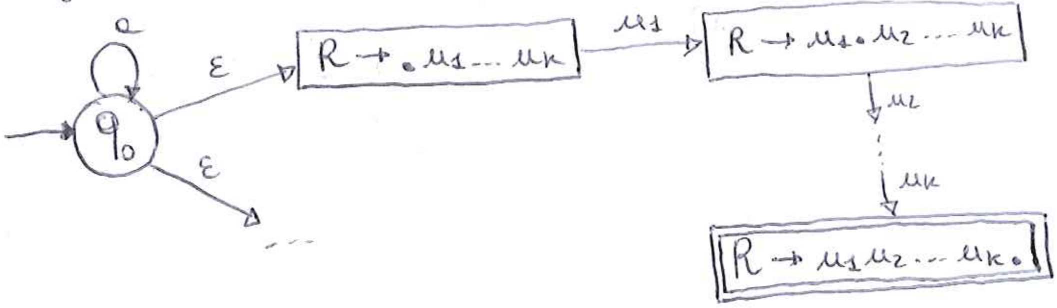
• Costruzione di DK:

DK è un DFA, costruibile da un NFA K, il quale si può costruire a partire da un altro NFA J.

(1) NFA J: J è un NFA che accetta ogni stringa in input che termina con la PARTE DESTRA di una regola di $\mathcal{G}$.

Gli stati sono detti "DOTTED RULES" e sono rappresentati come regole. Le regole con il "DOT" alla fine sono dette REGOLE COMPLETE e corrispondono a stati d'accettazione.
Le transizioni consumano il simbolo ALLA DESTRA del DOT.

In generale, J avrà la forma:

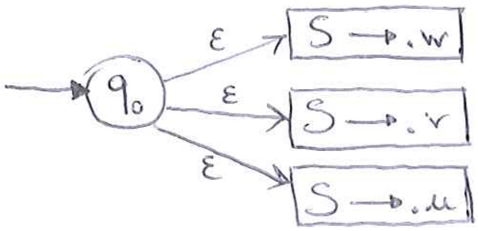


* Il SELF-LOOP su q_0: serve per indovinare dove inizia la parte destra della regola di $\mathcal{G}$ (NONDETERMINISMO).

Inoltre, c'è nondeterminismo nelle ε-arrows, una per ogni regola di $\mathcal{G}$!

(2) NFA K:

Se S è la start variable di G e $S \to u \mid v \mid w$, allora:



- Inoltre, $\forall a \in (\Sigma \cup V)$, $\forall\ B \to uav$:

  per ogni dotted rule $\boxed{B \to u.av}$, consideriamo anche lo stato $\boxed{B \to ua.v} \xleftarrow{a} \boxed{B \to u.av}$ e la freccia che li collega, leggendo "a".

- $\forall\ B \to uCv$, $C \to r$:

  avremo anche: $\boxed{B \to u.Cv} \xrightarrow{\varepsilon} \boxed{C \to .r}$, con un'ε-arrow, in maniera nondeterministica.

(3) DFA DK:

Convertire l'NFA K nel DFA DK con la costruzione analizzata a suo tempo: gli stati di DK sono <u>SOTTOINSIEMI DI STATI</u> di K; gli stati d'accettazione di DK sono quelli che includono <u>almeno</u> 1 REGOLA COMPLETA di K.

DK - TEST

Data una CFG G, costruire l'equivalente AUTOMA DK.

G è una DCFG $\iff$	(1) Ogni stato d'accettazione di DK ha ESATTAMENTE 1 UNICA regola completa; (2) ogni stato d'accettazione di DK <u>non</u> include nessuna dotted rule in cui un terminale segue immediatamente il dot ".".

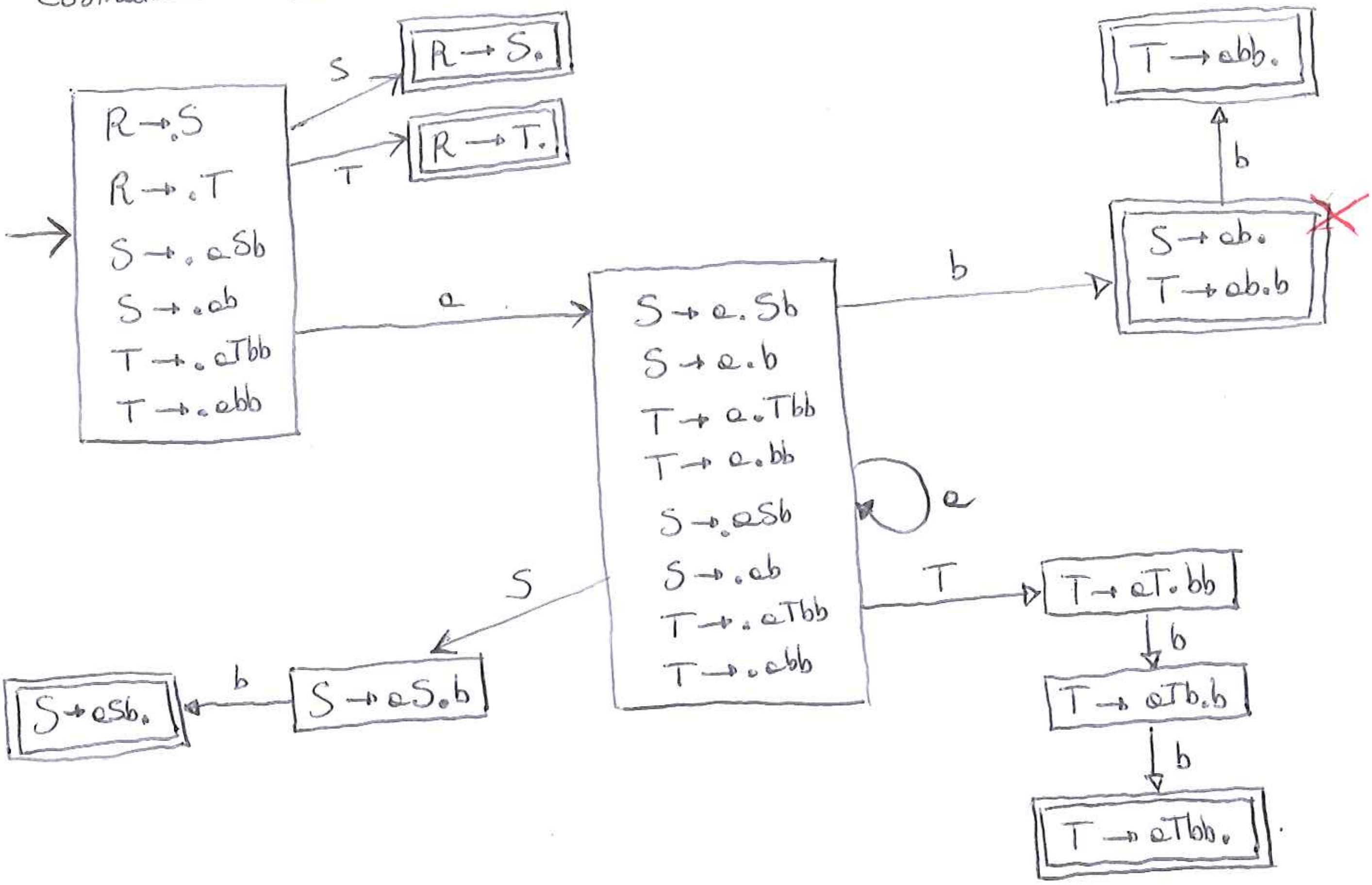
• Esempio:

Dire se G_1: $R \to S \mid T$ è o meno una DCFG.

$S \to aSb \mid ab$

$T \to aTbb \mid abb$

Costruiamo l'automa DK:



* Il DK-TEST **non** è superato!

Lo stato $\boxed{\begin{array}{l} S \to ab. \\ T \to ab.b \end{array}}$ è d'accettazione, ma presenta una regola con un terminale dopo il dot!

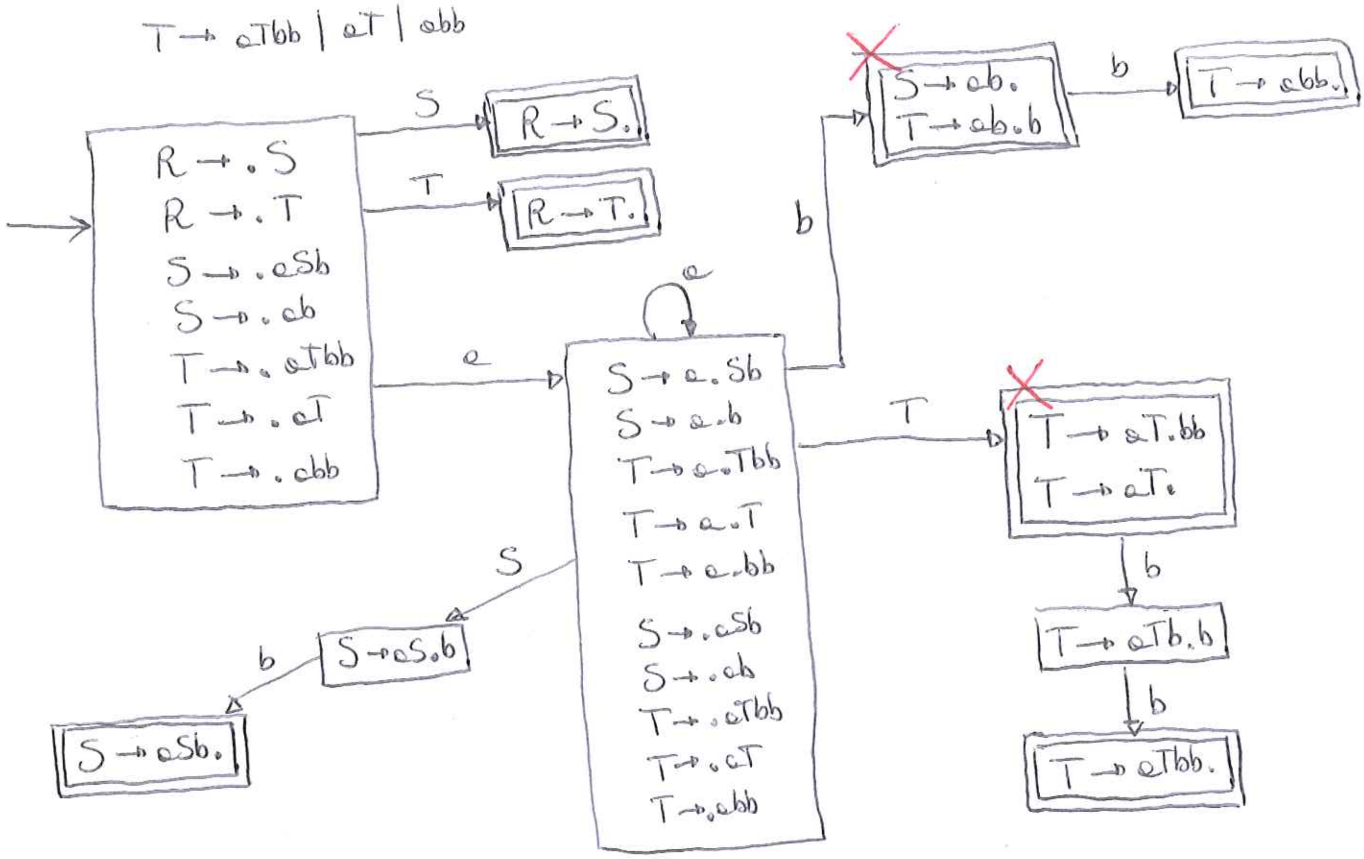
$\Rightarrow$ G_1 **NON** è una DCFG.

Esercizio:

G_0: $R \to S \mid T$

$S \to aSb \mid ab$

$T \to aTbb \mid aT \mid abb$



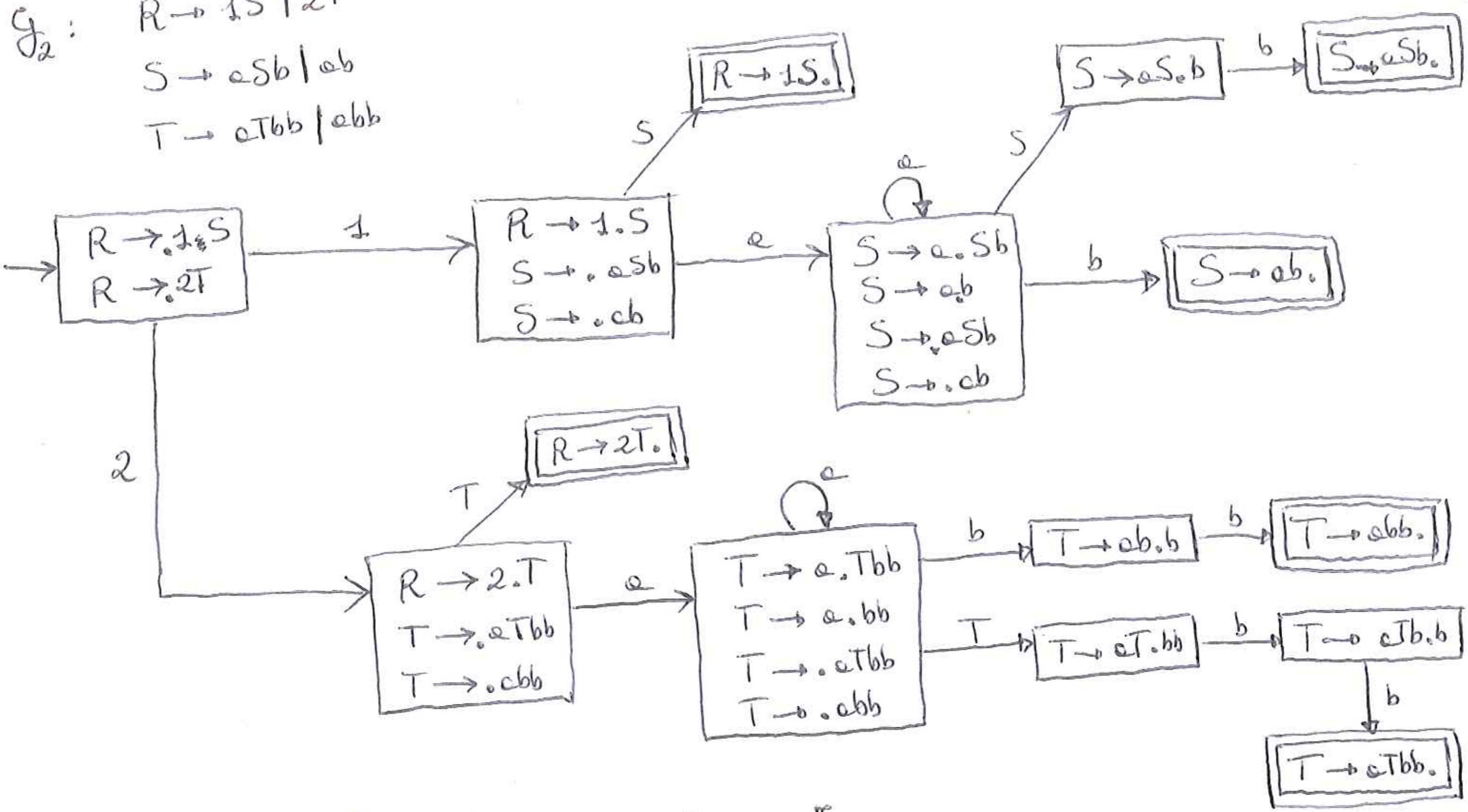
$\Rightarrow$ G_0 NON è una DCFG !

Esercizio:

G_2: $R \to 1S \mid 2T$

$S \to aSb \mid ab$

$T \to aTbb \mid abb$



* DK-TEST superato! G_2 è una DCFG!

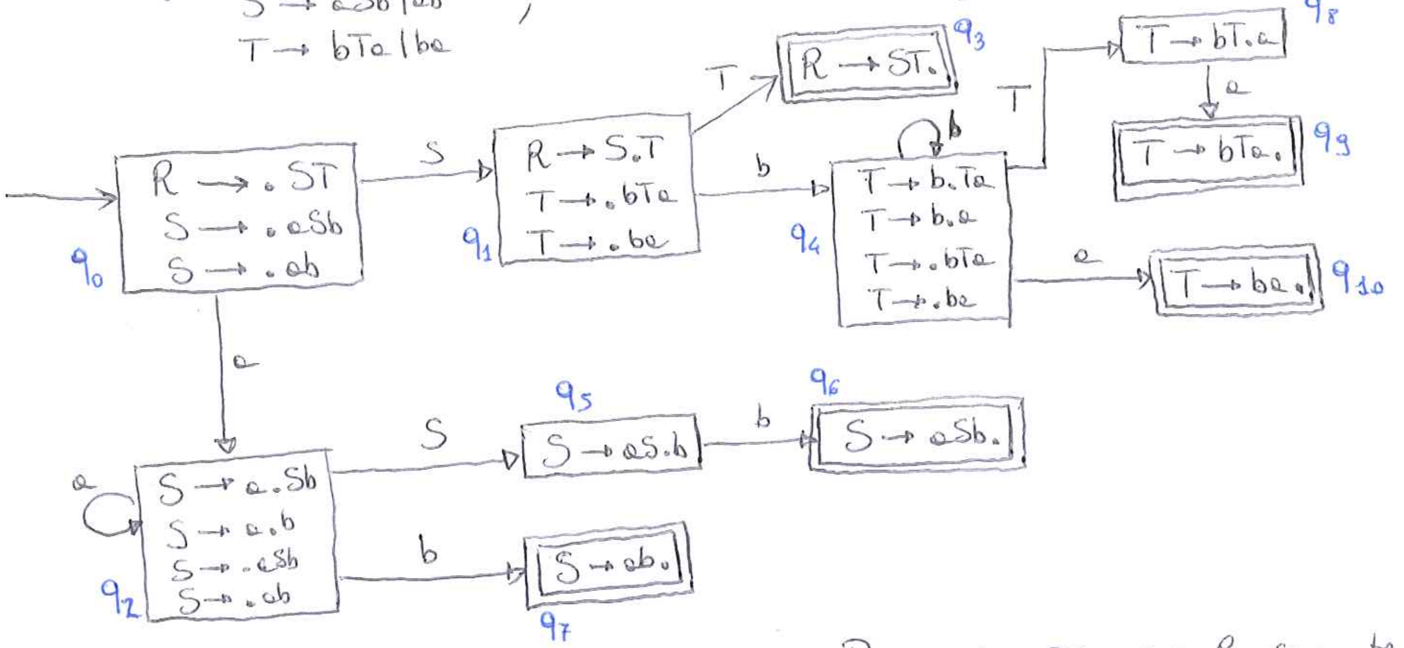
• TEOREMA: $\boxed{\text{Ogni DCFG ha un DPDA equivalente.}}$

Pseudocode:

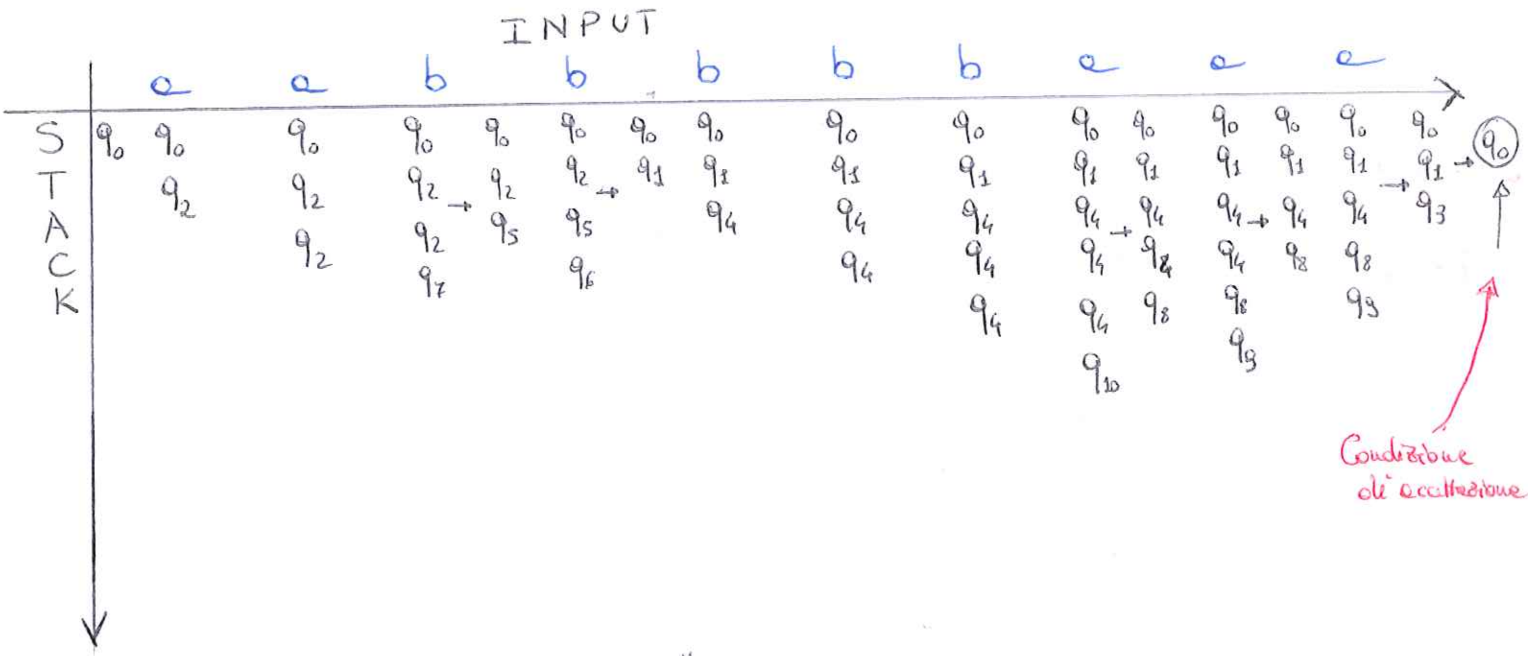
```
q := start state di DK          # q stato interno di P (DPDA)
i := 1                          # indice di input z (z_i)
do repeat:
    push(q)
    if q == [T → u.] then:      # torna indietro
        pop and discard |u| symbols from the stack
        q := pop()
        push(q)
        q := (q -T->)           # segui la freccia marcata con T da q
    else:
        q := (q -z_i->)         # segui la freccia con il prossimo simbolo dell'input (vai sul prossimo stato).
        i := i+1
    endif
until q == [S → h.]
accept if all z symbols have been read
```

* **NOTA**: in pratica, il DPDA P è basato sulla simulazione dell'automa DK per la DCFG G.

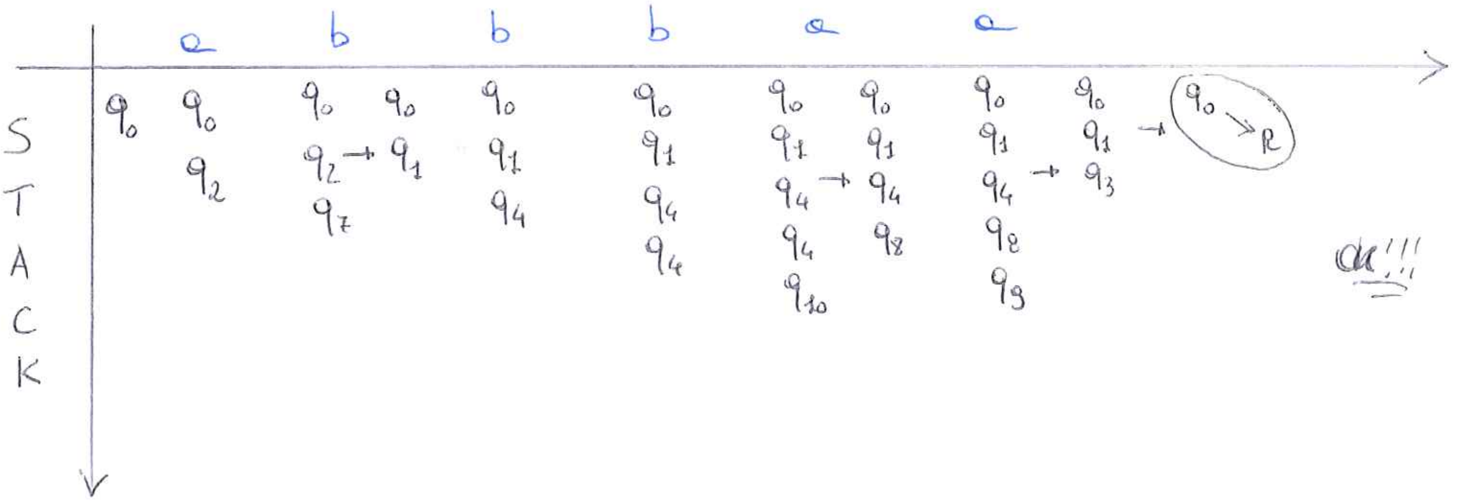
Esempio: G: $R \to ST$
$S \to aSb \mid ab$
$T \to bTa \mid ba$; Costruiamo l'automa DK:



$\Rightarrow$ G è una DCFG! Vediamo come il DPDA P simula DK con la seguente computazione: $P(z)$, z = "aabbbbbaaa"



Proviamo con $P(z)$, z = "abbbaa"



Dunque, abbiamo visto che:

$$\boxed{\forall \text{ DCFG } G, \ \exists \text{ DPDA } P \text{ tale che } L(P) = L(G)}$$

Tuttavia, NON E' VERO IL VICEVERSA!

~~$\forall$ DPDA P, $\exists$ DCFG G such that $L(P) = L(G)$~~

Vale però il seguente TEOREMA:

> Ogni DPDA che riconosce un ENDMARKED LANGUAGE ha una DCFG equivalente!

Avevamo già dimostrato che: $\forall$ PDA P, $\exists$ CFG G : $L(P) = L(G)$ ed il nostro PDA P ha le seguenti restrizioni:

(1) ogni passo è una PUSH(), una POP() o una READ(), ma non insieme;

(2) quando P accetta, SVUOTA LO STACK.

Per il 1° punto, non ci sono problemi, è applicabile anche al nostro caso.
Nel 2° punto, veniva usato il NONDETERMINISMO per indovinare quando l'input era finito e svuotare lo stack.

- Evita il NONDETERMINISMO: grazie all'essere un ENDMARKED LANGUAGE, quindi non deve indovinare; inoltre, per le regole $A_{pq} \rightarrow A_{pr} A_{rq}$, se si hanno più punti intermedi con lo stack vuoto (r, r'), si sceglie sempre quello più vicino a q.

RIEPILOGO

(1) A è un DCFL $\Longleftrightarrow \exists$ DPDA P : $L(P) = A$

(2) A è un DCFL $\Longleftrightarrow$ $A\dashv$ è un DCFL

(3) Se G è una DCFG $\Longrightarrow \exists$ DPDA P : $L(P) = L(G) \Longrightarrow L(G)$ è un DCFL

(4) Se P è un DPDA e $L(P) = A\dashv \ \Longrightarrow \exists$ DCFG G : $L(G) = L(P) = A\dashv$

- **Linguaggio "PREFIX-FREE":**

A è un linguaggio PREFIX-FREE se $\forall w \in A$, $\forall z \neq \varepsilon$, $wz \notin A$.
Cioè se non ci sono in A 2 stringhe tali per cui una è la sottostringa iniziale dell'altra (PREFIX).

- **LEMMA:** $\boxed{\text{Ogni ENDMARKED LANGUAGE } A\dashv \text{ è prefix-free.}}$

Ovvio, poiché $\forall w \in A\dashv$ non ci sarà mai "$\dashv$" prima della fine della stringa.

- **LEMMA:** $\boxed{\text{Se } \mathcal{G} \text{ è una DCFG} \Rightarrow \mathcal{L}(\mathcal{G}) \text{ è PREFIX-FREE}}$

Dunque, $\boxed{\text{la classe di linguaggi generati dalle DCFG è la classe dei linguaggi DCFL PREFIX-FREE.}}$

- **Esempio:**

$\mathcal{L} = \{0^n 1^m \mid 0 < m < n\}$ è un DCFL, ma $\nexists$ DCFG $\mathcal{G}$ tale che $\mathcal{L} = \mathcal{L}(\mathcal{G})$, poiché $\mathcal{L}$ non è prefix-free: sia "0001" $\in \mathcal{L}$ che "00011" $\in \mathcal{L}$ ed ovviamente "0001" è un PREFISSO per "00011".

- **"LOOKAHEAD":**

Sia $v = xhy$ una stringa valida per una data CFG. Supponiamo che l'handle di v sia $(h, T \to h)$.

[L'handle $(h, T \to h)$ è "FORZATO dal LOOKAHEAD K" se $(h, T \to h)$ è l'unico handle per ogni stringa valida del tipo $xh\hat{y}$, con $\hat{y} \in \Sigma^*$ ed y e $\hat{y}$ che COINCIDONO sui primi K simboli.]

* se $|y| < K$ o $|\hat{y}| < K \Rightarrow$ la stringa più corta è prefisso della seconda.

* HANDLE FORZATO $\equiv$ LOOKAHEAD (0).

- **GRAMMATICA $\mathcal{LR}(k)$:**

Una $\mathcal{LR}(k)$ - GRAMMAR è una CFG tale che l'handle di ogni stringa valida è forzato da LOOKAHEAD k.

Dunque: $\mathcal{LR}(0) \equiv$ DCFG.

- **TEOREMA:** $\boxed{\text{Se } \mathcal{G} \text{ è una } \mathcal{LR}(k),\ k>0 \Rightarrow \exists \text{ DPDA } P : \mathcal{L}(P) = \mathcal{L}(\mathcal{G}) \Rightarrow \mathcal{L}(\mathcal{G}) \text{ è DCFL.}}$

Cioè, le grammatiche $\mathcal{LR}(k)$ generano linguaggi DCFL.

DK1 - TEST

È il DK-Test per grammatiche $\mathcal{LR}(1)$, verifica se SONO o MENO $\mathcal{LR}(1)$!

- DIFFERENZE: ogni stato contiene le dotted rules, ma con i SIMBOLI DI LOOKAHEAD

$\boxed{T \to u.v \quad a}$ $\longrightarrow$ K1 ha letto u, che è parte di un handle $(uv, T \to uv)$, a condizione che "v" seguirà dopo "u" ed il simbolo di lookahead "a" seguirà dopo "v".

- START STATE:

$\boxed{S \to .u \quad a}$ $\quad \forall$ regola $S \to u$, $\forall a \in \Sigma \Rightarrow$ Tutti i simboli terminali sono contenuti come simboli di lookahead.

- SHIFT RULES:

(1) $\boxed{T \to u.xv \quad a} \xrightarrow[x \in \Sigma \cup V]{x} \boxed{T \to ux.v \quad a}$ Non cambiano i simboli di lookahead!

(2) $\boxed{T \to u.Cv \quad a} \xrightarrow{\varepsilon} \boxed{C \to .r \quad b}$

$\forall (C \to r) \in R$

Con le condizioni che:

- b è il 1° simbolo di ogni stringa che può essere derivata da v

OPPURE

- $b = a$, se v deriva la stringa vuota ε.

- ACCETTAZIONE:

$R_1 \to u.\quad a_1$

$R_2 \to w.xv\quad a_2$

R_1 ed R_2 sono detti "CONSISTENTI" se

(1) R_1 ed R_2 sono entrambe completate e $a_1 = a_2$;

OPPURE

(2) R_2 non è completata, ma a_1 segue immediatamente il DOT in R_2 (cioè: $x \equiv a_1$).

- DK1 - TEST ha successo se:

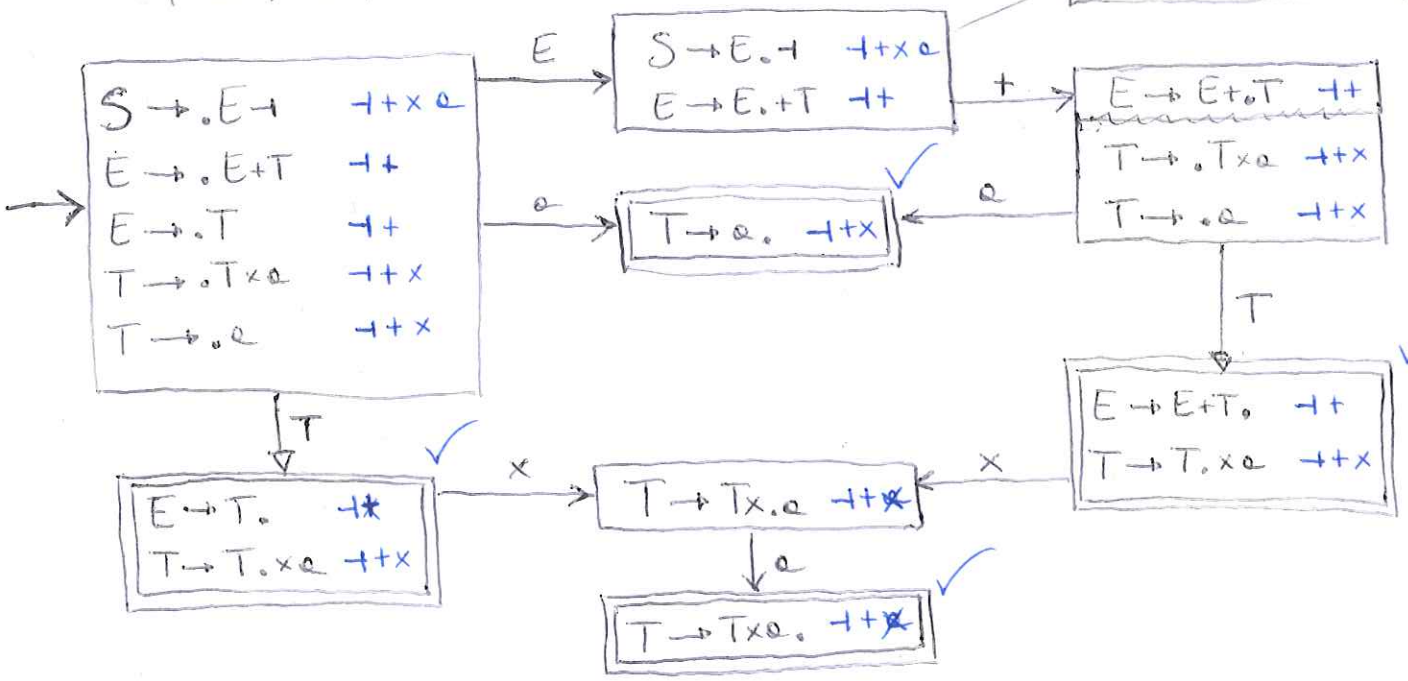
Ogni accept state NON contiene 2 regole CONSISTENTI!

- Esempio:

G: $S \to E\dashv$; G è una LR(1)-grammar?

$E \to E+T \mid T$

$T \to T\times a \mid a$



* Gli stati d'accettazione con 2 regole hanno come simbolo dopo il dot nelle regole non completate "×", ma non compare nei simboli di lookahead delle altre regole ⇒ NON sono CONSISTENTI

⇒ DK1 - Test superato ⇒ G è una LR(1) - GRAMMAR!

IL PARSER DI EARLEY

Questo parser è un algoritmo che ha COMPLESSITÀ TEMPORALE variabile, a seconda della grammatica passata in input; se l'input è di lunghezza n:

- $O(n^3)$ se $\mathcal{G}$ è AMBIGUA;
- $O(n^2)$ se $\mathcal{G}$ è NON AMBIGUA;
- $O(n)$ se $\mathcal{G}$ è una DCFG.
- $O(n)$ se $\mathcal{G}$ è $LR(k)$.

$\Rightarrow \forall$ DCFL L, $\exists_{DCFG}$ $\mathcal{G}$ tale che $\mathcal{G}$ è parsata in tempo lineare $O(n)$ dal Parser di Earley.

COSTRUZIONE:

Data una grammatica $\mathcal{G} = (V, \Sigma, R, S)$ ed una stringa di terminali $w \in \Sigma^*$, con $w = w_1 w_2 \dots w_n$:

il parser costruisce una sequenza di SOTTOINSIEMI DI STATI $S(0), S(1), \dots, S(k)$ in accordo con le seguenti 4 regole R_0, R_1, R_2 ed R_3:

- **R_0 - INIZIALIZZAZIONE:**

$S(0)$ include $\boxed{R_0 \rightarrow .u \mid 0}$ per ogni regola $R_0 \rightarrow u$ di $\mathcal{G}$, dove R_0 è la variabile di partenza di $\mathcal{G}$ e 0 è una "LABEL" che indica in quale momento per la 1ª volta la regola è stata introdotta.

- **R_1 - PREDIZIONE:**

Se $\boxed{A \rightarrow u.Bv \mid i} \in S(j)$ e $B \rightarrow z$ è una regola in $\mathcal{G}$, allora aggiungere ad $S(j)$ anche la regola $\boxed{B \rightarrow .z \mid j}$.

- **R_2 - COMPLETAMENTO**:

Se $\boxed{A \rightarrow u. \mid i} \in S(j)$, allora per ogni $\boxed{B \rightarrow v.Az \mid k} \in S(i)$, aggiungi ad $S(j)$ lo stato $\boxed{B \rightarrow vA.z \mid k}$.

- **R_3 - SCANNING**:

Per ogni stato $\boxed{A \rightarrow u.w_j z \mid k} \in S(j-1)$, aggiungi ad $S(j)$ lo stato $\boxed{A \rightarrow u w_j . v \mid k}$.

ACCETTAZIONE: Il Parser di Earley accetta SE E SOLO SE lo stato finale $S(n)$ include uno stato della forma: $\boxed{R_0 \rightarrow u. \mid 0}$

- **Esempio**:

$\mathcal{G}$: $E \rightarrow E+T \mid T$
$T \rightarrow T \times F \mid F$; verificare che "$a+a\times a$" $\in \mathcal{L}(\mathcal{G})$.
$F \rightarrow (E) \mid a$

Stringa di input w="$a+a\times a$"

- $S(0)$: ε

1 $\boxed{E \rightarrow .E+T \mid 0}$ R_0 2 $\boxed{E \rightarrow .T \mid 0}$ R_0 3 $\boxed{T \rightarrow .T\times F \mid 0}$ R_1 2

4 $\boxed{T \rightarrow .F \mid 0}$ R_1 2 5 $\boxed{F \rightarrow .(E) \mid 0}$ R_1 4 6 $\boxed{F \rightarrow .a \mid 0}$ R_1 4

- $S(1)$: a

7 $\boxed{F \rightarrow a. \mid 0}$ R_3 6 8 $\boxed{T \rightarrow F. \mid 0}$ R_2 7 → 4 9 $\boxed{E \rightarrow T. \mid 0}$ R_2 8 → 2

10 $\boxed{T \rightarrow T.\times F \mid 0}$ R_2 8 → 3 11 $\boxed{E \rightarrow E.+T \mid 0}$ R_2 9 1

- $S(2)$: $+$

12 $\boxed{E \rightarrow E+.T \mid 0}$ R_3 11 13 $\boxed{T \rightarrow .T\times F \mid 2}$ R_1 12 14 $\boxed{T \rightarrow .F \mid 2}$ R_1 12

15 $\boxed{F \rightarrow .(E) \mid 2}$ R_1 14 16 $\boxed{F \rightarrow .a \mid 2}$ R_1 14

- $S(3)$: a

(17) $\boxed{F \to a. \mid 2}$ R_3 (16)

(18) $\boxed{T \to F. \mid 2}$ R_2 (17)(16)

(19) $\boxed{T \to T.\times F \mid 2}$ R_2 (18)(13)

(20) $\boxed{E \to E+T. \mid 0}$ R_2 (18)(12)

(21) $\boxed{E \to E.+T \mid 0}$ R_2 (20)(1)

- $S(4)$: $\times$

(22) $\boxed{T \to T\times.F \mid 2}$ R_3 (19)

(23) $\boxed{F \to .(E) \mid 4}$ R_1 (22)

(24) $\boxed{F \to .a \mid 4}$ R_1 (22)

- $S(5)$: a

(25) $\boxed{F \to a. \mid 4}$ R_3 (24)

(26) $\boxed{T \to T\times F. \mid 2}$ R_2 (25)(22)

(27) $\boxed{E \to E+T. \mid 0}$ R_2 (26)(12)

ACCETTO!

(28) $\boxed{T \to T.\times F \mid 2}$ R_2 (26)(13)

(29) $\boxed{E \to E.+T \mid 0}$ R_2 (27)(1)

$\Rightarrow w =$ "$a+a\times a$" $\in \mathcal{L}(G)$!

- **Dal Parser di Earley al Parse Tree:**

(1) Costruire un albero seguendo le seguenti regole:

1.a) ha come RADICE lo stato accettante di $S(n)$;

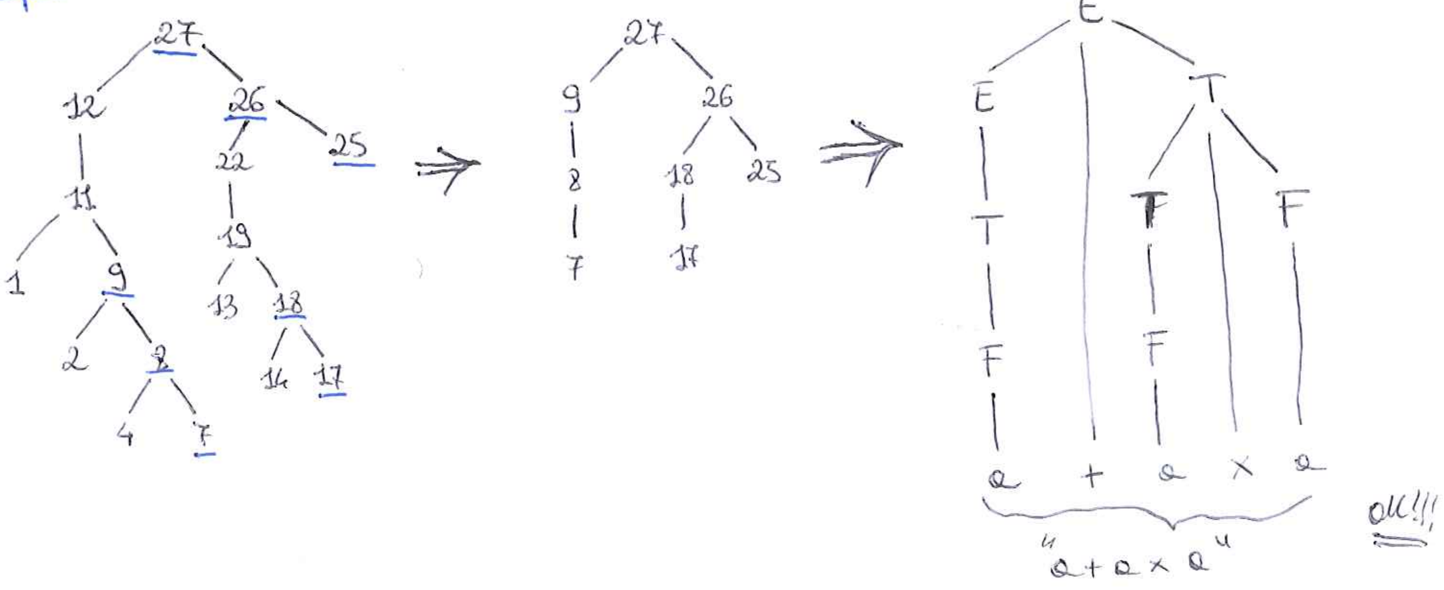
1.b) ogni nodo ha come FIGLI gli stati usati per aggiungerlo, col figlio sinistro prodotto prima;

1.c) le FOGLIE sono gli stati con le regole che non hanno variabili alla sinistra del dot.

(2) Rimuovere i nodi con regole non completate;

(3) Sostituire gli stati con le regole al posto dei numeri.

- Esempio:



ok!!!