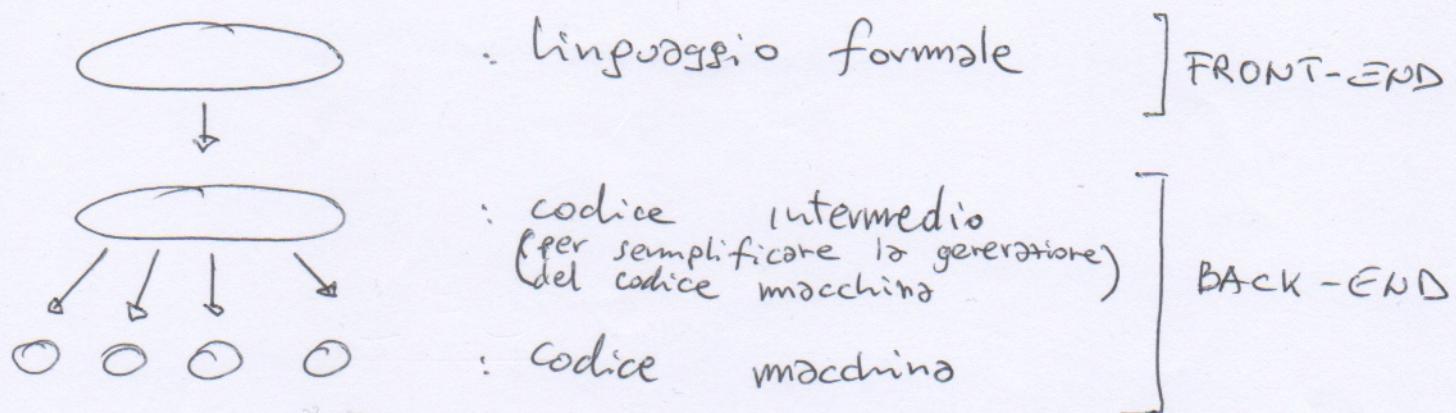


COMPILATORE



- ANALISI LESSICALE : $p = k \cdot 2 \Rightarrow id_1 \text{ ass } id_2 \text{ mul } n_1$
- ANALISI SINTATTICA : $id_1 \text{ ass } id_2 \text{ mul } n_1 \Rightarrow id_1 \text{ ass espressione}$
- ANALISI SEMANTICA : controlla se l'espressione appartiene al linguaggio.

REGULAR EXPRESSIONS

19M25

ϵ : parola nata.

α : alfabeto.

r_1, r_2, r_n : reg. exp.

$$\{d, e\} = \{(d)\}_0 \cup \{(e)\}_0 = (d|e)_0$$

$$\{dd, ed, de, ee\} = ((d|e)(d|e))_0$$

base: if $a \in \alpha$ then

a is a reg. exp.

$$\text{if } r = a \text{ then } \{a\} = \{a\}_0 \cup \{a\}_1 \cup \dots \cup \{a\}_n = (*a|*a)_0$$

$L(r) = \{a\}$ // linguaggio generato da r

if $a = \epsilon$ then

$$L(\epsilon) = \{\epsilon\}$$

$$(*(\epsilon|10)(\epsilon|10))_0$$

Step:

- alternation $[|]$: $r_1 | r_2$ è una reg. exp se

$$L(r_1 | r_2) = \{L(r_1) \cup L(r_2)\}$$

- concatenation $[.]$: $r_1 \cdot r_2$ è una reg. exp se
(o si omette)

$$L(r_1 \cdot r_2) = \{w = w_1 w_2 \text{ AND } w_1 \in L(r_1) \text{ AND } w_2 \in L(r_2)\}$$

- Kleene star $[*]$: r^* è una reg. exp se

$$L(r^*) = \{\epsilon\} \cup \{w_1 \dots w_k \text{ tc } w_i \in L(r) \text{ AND } k > 1\}$$

- parentesi $[()]$: (r) è una reg. exp se

$$L((r)) = L(r)$$

ESEMPIO

REGOLE EXPLORAZIONE

$$f(a|b) = \{f(a) \cup f(b)\} = \{a, b\}$$

stew clone : 3
selezione : 0

$$f((ab)(ab)) = \{aa, ab, ba, bb\}$$

gex wr : 2, 3, 7, 8

$$f(a|(ab)^*) = \{f(a) \cup f((ab)^*)\} = \{\epsilon, a, ab, abab, ababab, \dots\}$$

$$f(a^*|b^*) = \{\epsilon\} \cup \{a^* | k > 0\} \cup \{b^* | j > 0\}$$

trovare una reg exp che genera tutti gli identificatori possibili:

$$f((a| \dots | z)(a| \dots | z)^*)$$

$$\{B\} = \{B\}$$

$$\{(a)\} \cup \{(z)\} = (az)^*$$

$$\{(a)\} \cup \{(z)\} = (az)^*$$

$$\{x_{i-1} \dots x_0 (z)\} \cup \{x_{i-1} \dots x_0 (z)w\} = (xz)^*$$

$$(z)^* = (z)^*$$

LINGUAGGIO REGOLARE: linguaggio generato da espressione regolare

Priorità degli operatori come in algebra.
Ordini di precedenza:

- Klimi star
- concatenazione
- alternativa

ESEMPIO

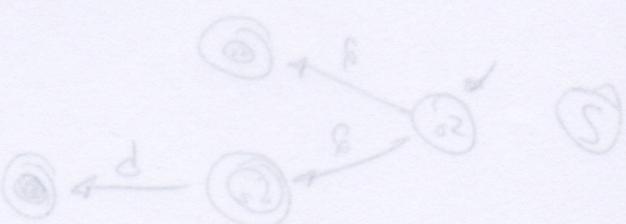
$$a|b^*c \rightarrow a|((b^*)c)$$

CONVENTIONI:

$$r^+ : \text{una o più occorrenze di } r$$
$$\Rightarrow rr^*$$

$$r? : \text{zero o una occorrenza di } r$$
$$\Rightarrow \epsilon|r$$

$$[abc] : \text{uno dei qualsiasi caratteri}$$
$$\Rightarrow a|b|c$$



AUTOMI NON DETERMINISTICI A STATI FINITI

NFA

definiti come una quintupla:

$$(S, \alpha, \text{moren}, s_0, F)$$

dove:

S : insieme degli stati

α : alfabeto, $\underline{\epsilon \in \Omega}$

s_0 : stato iniziale, $s_0 \in S$

F : sottoinsieme di stati finali di S , $F \subseteq S$
NB: essendo sottoinsieme, F può essere vuoto.

moren: funzione di transizione, $S \times (\alpha \cup \{\epsilon\}) \rightarrow 2^S$

Viene rappresentato come un grafo dove:

- nodi: sono gli stati

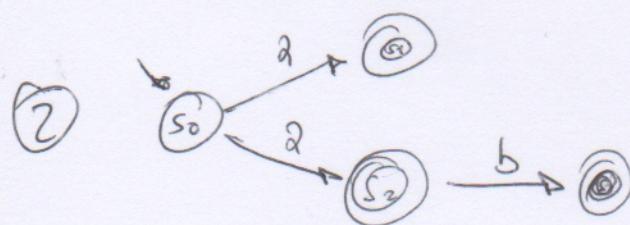
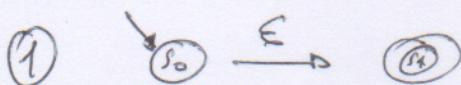
- archi: la funzione di transizione.

convenzione:

① : freccia entrante per indicare s_0

② : doppio cerchio per indicare lo stato finale.

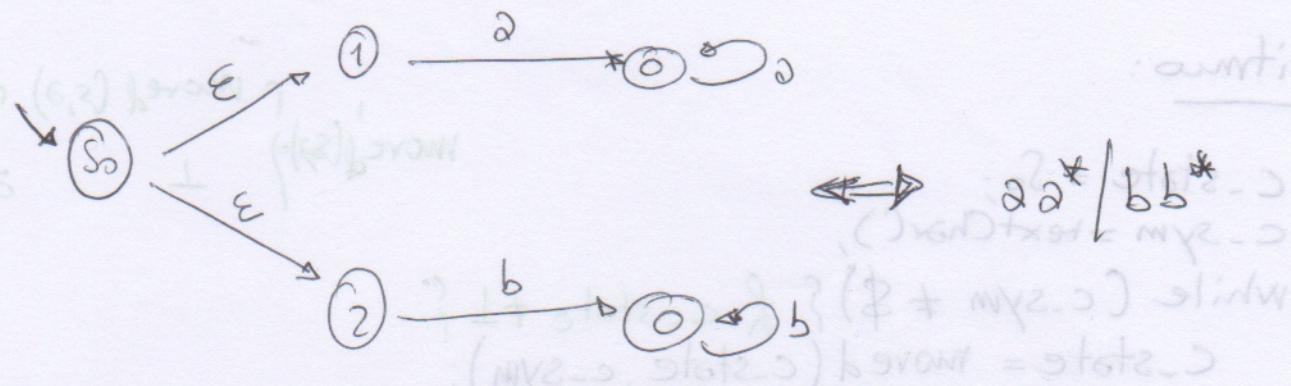
Esempio



l'automa ② non è deterministico perché con l'input "ab" è possibile prendere più percorsi, $s_1 \circ s_2$

DEFINIZIONE: Un NFA N accetta una stringa w se esiste almeno un cammino in N da s_0 fino a s_f , dove $s_i \in F$

esercizio: dato un automa, ricavare la reg. exp. che lo genera.



AUTOMA DETERMINISTICO A STATI FINITI DFA

definito come NFA ma cambia la funzione di transizione degli stati

$$(S, \alpha, \text{move}_d, s_0, F)$$

$\text{Move}_d: S \times \alpha \rightarrow S$

NB: non ci sono transizioni con ϵ in DFA, perché $\epsilon \notin \alpha$
Per ogni $s \in S$, per ogni $a \in \alpha$ c'è esattamente un percorso etichettato con a che ~~ha~~ da s

ALGORITMO DI SIMULAZIONE PER DFA

Input: - stringa w terminata da un carattere speciale \$
- DFA D con funzione di transizione totale
- DFA D con funzione di transizione parziale

Output: s/no alla domanda $w \in L(D)$

algoritmo:

```
c-state = S0;
c-sym = textChar();
while (c-sym != $) { if c-state ≠ ⊥ {
    c-state = moved(c-state, c-sym);
    c-sym = nextChar();
}
```

```
if (c-state ∈ F) return si;
else return no;
```

$\text{moved}(s, a)$ è definito
 $\text{moved}(s) =$ 1 altrove

$(\text{f}, \text{al}, \text{parz}, D, ?)$

$2 + |D| \times 2$: fasi

COSTRUZIONE di THOMPSON

PREPARAZIONE (Reg. exp \rightarrow NFA)

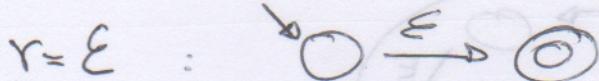
- Obiettivo - creare automi per le strutture delle reg-exp elementali.
- fornire regole di composizione di automi per gli operatori: $*$, $|$, $.$, * .

Osservazione 1: ogni passo della costruzione aggiunge al più di due stati
 \Rightarrow NFA risultante è al più 2^n stati, dove n è il numero di elementi.

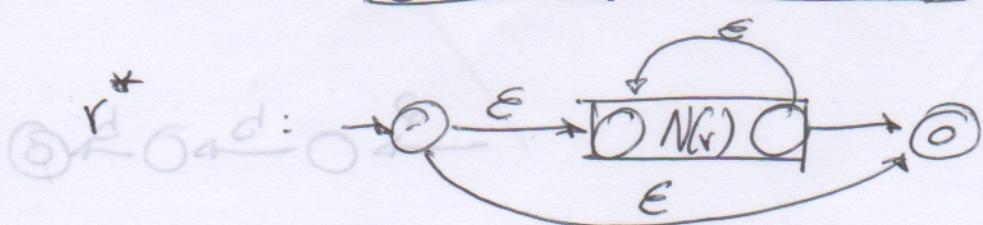
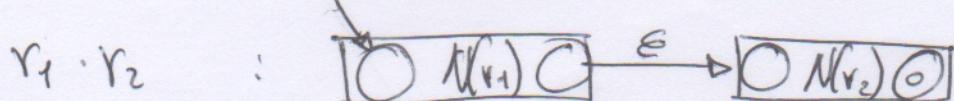
Osservazione 2: ogni NFA intermedio possiede:

- ha esattamente uno stato finale
- non ha archi entranti nello stato iniziale
- non ha archi uscenti dallo stato finale.

base:



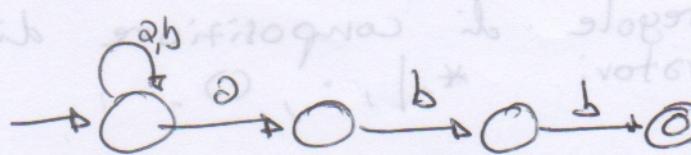
step:



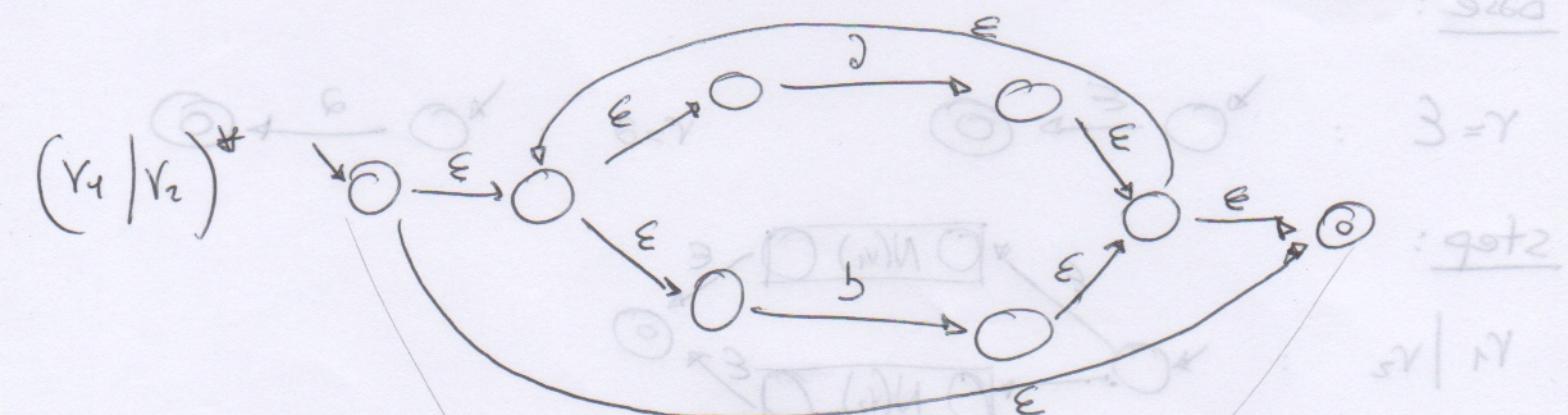
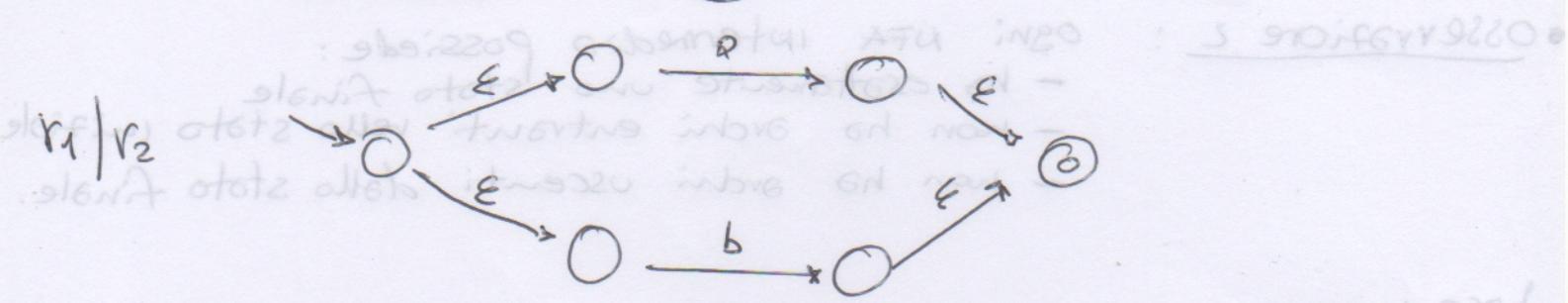
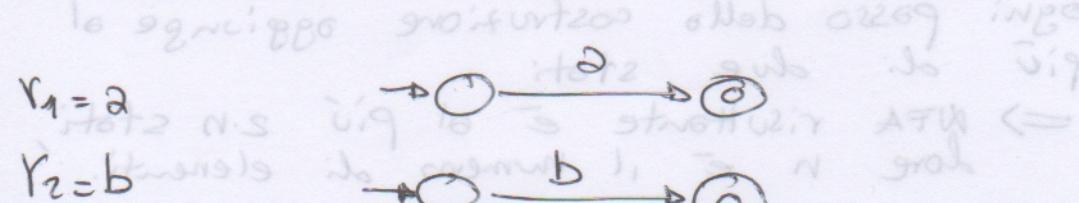
ESEMPIO

Dato la reg. exp: $(abb)^*abb$ costruire l'NFA.

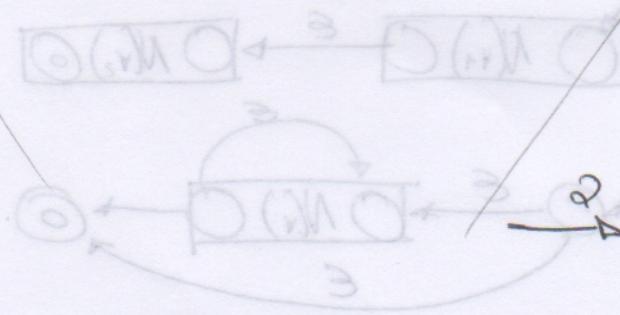
AUTOMA SENZA REGOLE DI THOMPSON



AUTOMA CON REGOLE DI THOMPSON

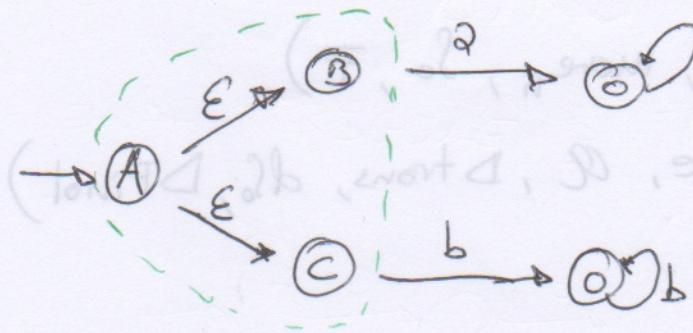


$(r_1 | r_2)^*abb$



(r)

SUBSET CONSTRUCTION



Si possono raggruppare come stato iniziale: E-chiusa

$$E\text{-chiusa}(\{A\}) = \{A, B, C\}$$

dato uno stato, quanti stati trovi facendo ε-passi?

Algoritmo E-chiusa(T)

$$E\text{-chiusa}(T) = T$$

push lo stato di T nello stack

while (stack non è vuoto) {

pop s_1 dallo stack

foreach (s_2 raggiungibili da $s_1 \xrightarrow{\epsilon} s_2$) {

if $s_2 \in E\text{-chiusa}(T)$ {

add s_2 to $E\text{-chiusa}(T)$

push s_2 nello stack

}

} {ogni stato ab raggiungibile da s_1 è un

} {ogni stato ab raggiungibile da s_1 è un

TRASFORMAZIONE = NFA \rightarrow DFA

Dato NFA $N = (S, \alpha, \text{move}_n, S_0, F)$

Determinare $D = (\Delta_{\text{State}}, Q, \Delta_{\text{trans}}, dS_0, \Delta_{\text{Final}})$
tali che $L(N) = L(D)$

- $dS_0 = \text{E-chiessa}(\{S_0\})$

Definizione: gl'insiemi state sono raggruppate secondo i

$\Delta_{\text{State}} = \{dS_0\}$ state $\{S, \alpha, A\} = (\{\lambda\})$

set dS_0 unmarked in Δ_{State}

While (there is some unmarked state in Δ_{State}) {

mark T

foreach ($a \in \alpha$) {

$$A = \bigcup_{t \in T} \text{move}_n(t, a)$$

$$B = \text{E-chiessa}(A)$$

If ($B \notin \Delta_{\text{State}}$) add B to Δ_{State} as unmarked

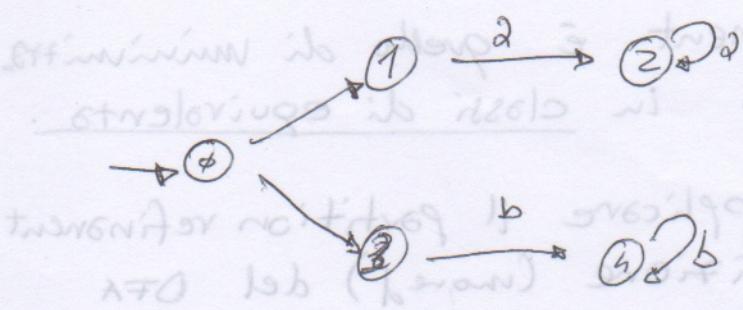
$$\Delta_{\text{trans}}(T, a) = B$$

}

}

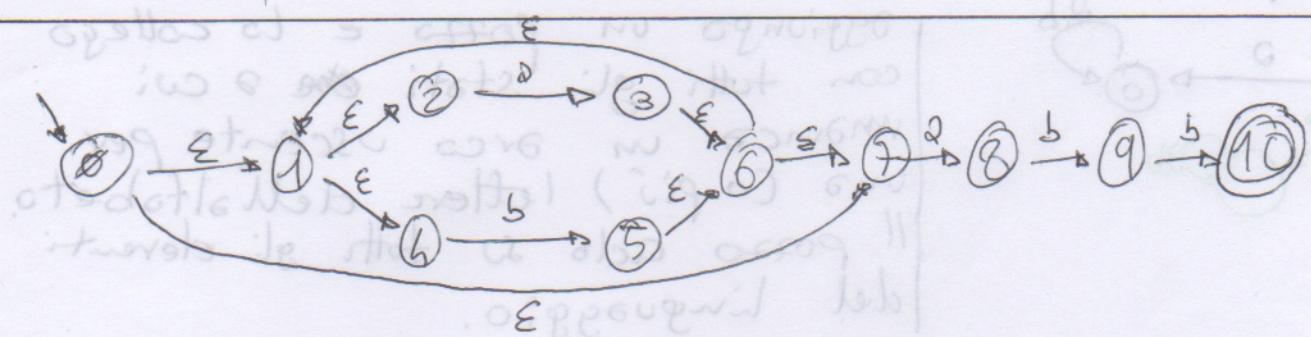
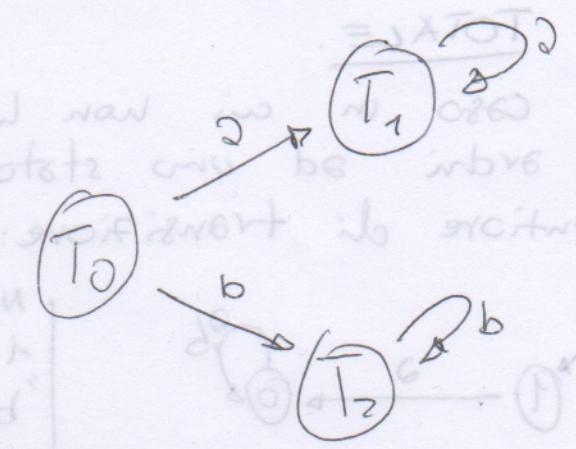
- Δ_{Final} : elementi degli Δ_{States} che contengono almeno uno che sia stato finale in N.

APPLICATION = SUBSET CONSTRUCTION

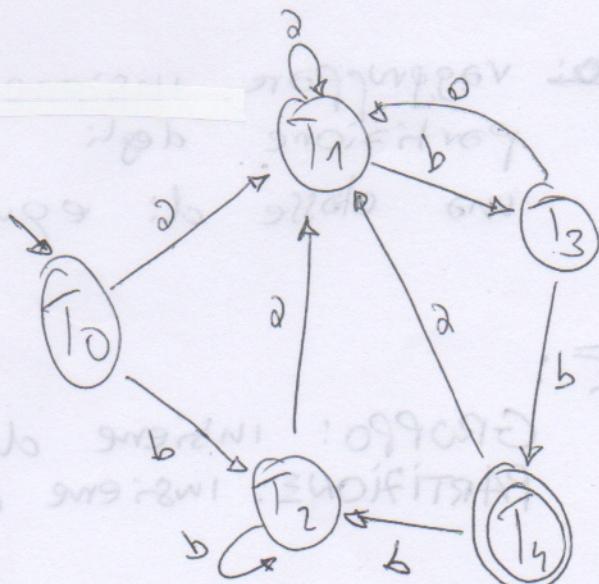


NFA

| | a | b |
|---------------------|-------------|-------------|
| $T_0 = \{0, 1, 2\}$ | \bar{T}_1 | \bar{T}_2 |
| $T_1 = \{1, 2\}$ | \bar{T}_1 | / |
| $T_2 = \{4\}$ | / | \bar{T}_2 |



| | a | b |
|----------------------------------|-------------|-------------|
| $T_0 = \{0, 1, 2, 4, 7\}$ | \bar{T}_1 | \bar{T}_2 |
| $T_1 = \{3, 8, 6, 7, 1, 5, 2\}$ | \bar{T}_1 | \bar{T}_3 |
| $T_2 = \{5, 6, 7, 1, 4, 2\}$ | \bar{T}_1 | \bar{T}_2 |
| $T_3 = \{5, 9, 6, 7, 1, 2, 5\}$ | \bar{T}_1 | \bar{T}_4 |
| $T_4 = \{10, 5, 6, 7, 1, 4, 2\}$ | \bar{T}_1 | \bar{T}_2 |
| | | FIN. |



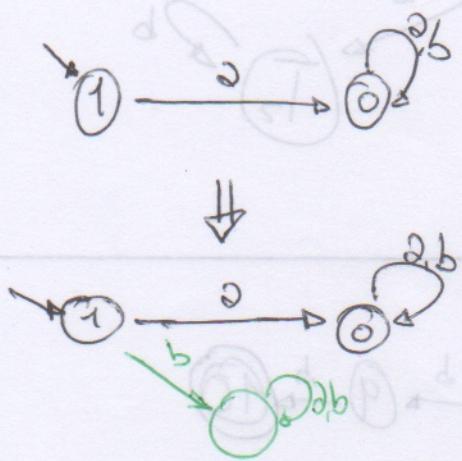
DFA

PARTITION REFINEMENT ($DFA \rightarrow \text{min DFA}$)

Lo scopo del partition refinement è quello di minimizzare il DFA raggruppando gli stati in classi di equivalenza.

Condizione NECESSARIA per applicare il partition refinement è che la funzione di transizione (move) del DFA sia TOTALE.

Nel caso in cui non lo fosse si applicano al DFA degli archi ad uno stato posto in modo da completare la funzione di transizione:



Non è totale perché nello stato
"1" manca un arco che porta
"b"

Aggiungo un pozzo e lo collego
con tutti gli stati ~~a~~ a cui
manca un arco uscente per
una (o più) lettere dell'alfabeto.
Il pozzo accia su tutti gli elementi
del linguaggio.

Scopo: raggruppare insieme gli stati per generare una partizione degli stati, i quali rappresentano una classe di equivalenza.

DEF:

GRUPPO: insieme di stati

PARTIZIONE: insieme di gruppi.

algoritmo:

if transaction function not total, add sink and proper edges

$$P = (F, S \setminus F)$$

while (P is REFINABLE) {

 apply a one-step refinement on P ;

 Update P

NB: questo algoritmo non è deterministico perché possiamo scegliere uno dei possibili passi per raffinare.

} figura sub ob ob → riportando di strutturazione - ituzionali, attesi → slavati state di struttura ON

definizioni:

• IS REFINABLE: dato un DFA con funzione di transizione totale mored, una partizione P dei suoi stati in m gruppi di stati è raffinabile se: mettendo tutti

$$\exists s_1, s_2 \in G_0 \text{ AND}$$

$$(\exists G_0, G_1, G_2 \in P : G_1 \neq G_2) \text{ AND } (\forall a \in \Sigma : \text{mored}(s_1, a) \in G_1 \text{ AND } \text{mored}(s_2, a) \in G_2)$$

è raffinabile se con una lettera, mi muovo in una transizione che vado in un gruppo G_1 e in un gruppo G_2 .

one-step refinement:

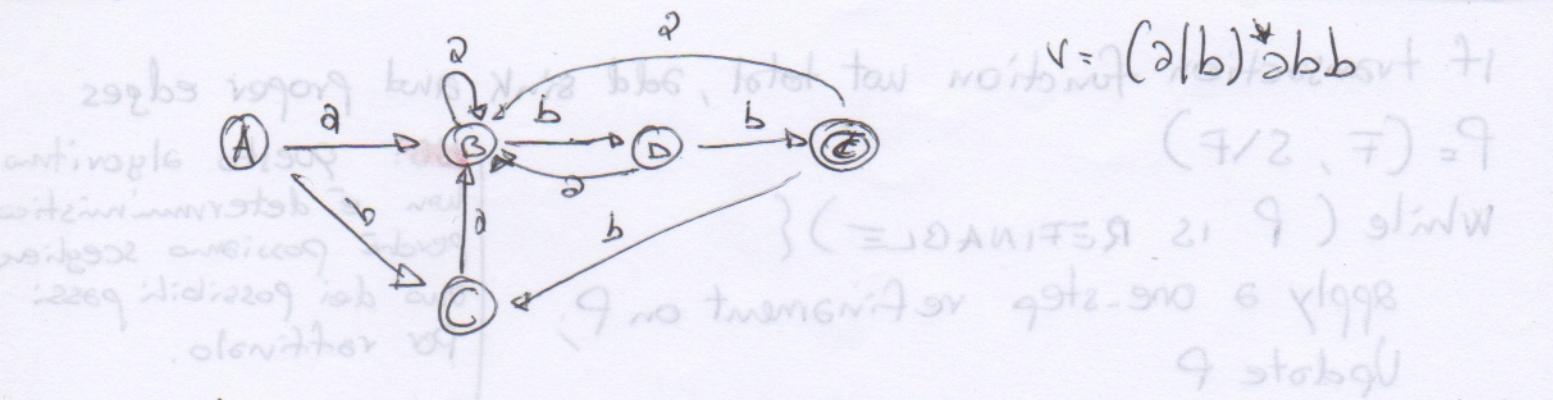
- scegli $G_0, G_1, G_2 \in P$ e $a \in \Sigma$

- dividiamo G_0 in due sottoinsiemi G_{01}, G_{02} t.c. $\forall G' \in P, \forall i \in \{1, 2\}, \forall s, t \in G_i, \text{mored}(s, a) \in G' \iff \text{mored}(t, a) \in G'$

- stato iniziale: lo stato che rappresenta il gruppo che contiene lo stato iniziale originario, è lo stato iniziale.

- stato finale: tutti gli stati che rappresentano gruppi che contengono solo stati finali, sono stati finali.

esempio:



$$v = (a/b)^*abb$$

$$(7/2, 7) = 9$$

- Inizialmente la partizione è data da due gruppi: uno contenente lo stato finale, e l'altro, rimanenti.

$$G_1 = \{E\} \quad G_2 = \{A, B, C, D\}$$

- In E entra una "b", quindi controllo nel gruppo G_2 dove mi portano le transizioni con la "b": $A \rightarrow B$, $B \rightarrow D$, $C \rightarrow C$, $D \xrightarrow{b} E$.
D non può stare in G_2 perché con una b-transizione mi porta in G_1 .

$$G_1 = \{E\} \quad G_{21} = \{A, B, C\} \quad G_{22} = \{D\}$$

- In D entra una "b", controllo negli altri gruppi: se ci sono delle b-transizioni che mi portano a D: $A \xrightarrow{b} C$, $C \rightarrow C$, $B \xrightarrow{b} D$.
B non può stare in G_{21} perché mi porta in G_{22} .

$$G_1 = \{E\} \quad G_{211} = \{D\} \quad G_{212} = \{B\} \quad G_{213} = \{A, C\}$$

- provando a partizionare G_{213} con la b-transizione, non si ottiene una nuova partizione, in quanto $A \xrightarrow{b} C$.
Con la a-transizione si ottiene: $A \xrightarrow{a} B$, $E \xrightarrow{a} B$. Quindi possono stare in una partizione unica.

Si raggiunge una struttura minima quando si ottengono ~~due~~ parti dei gruppi con le stesse caratteristiche, terminando i simboli dell'alfabeto.

COMPLESSITÀ

Per eseguire il "run" di una parola w su un DFA ha sostanzialmente una complessità che dipende dalla lunghezza della parola w : $O(|w|)$.

Questa complessità è dovuta agli algoritmi di costruzione che si eseguono solo una volta come preprocessing.

LEMMA: per ogni $n \in \mathbb{N}$ c'è un NFA con $(n+1)$ stati il cui equivalente minimo DFA ha almeno 2^n stati.

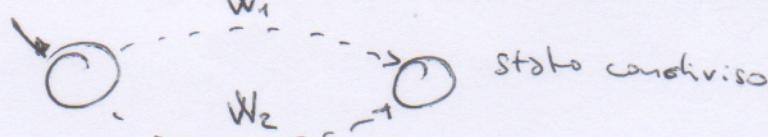
DIMOSTRAZIONE: $L = L((a/b)^* a (a/b)^{n-1}) \Rightarrow$ NFA che accetta L

$\underline{n+1} = +1$ $+1$ $\underbrace{\xrightarrow{a/b} \dots \xrightarrow{a/b}}_{n-1 \text{ stati}}$ $\xrightarrow{a/b} \textcircled{0}$

OSSERVAZIONE:

- L contiene parole w t.c. $|w| \geq n$
- ci sono almeno 2^n parole distinte sull'alfabeto $\{a, b\}$ che sono lunghe almeno n .

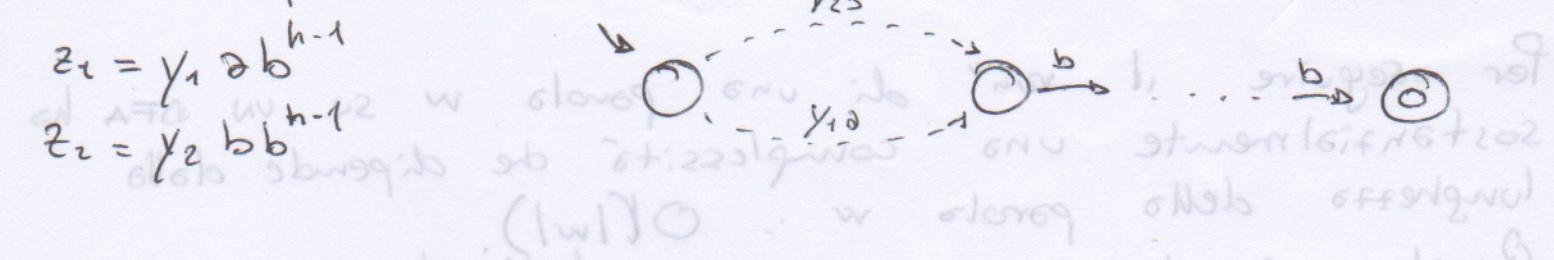
ASSUNZIONE: se assumiamo che ci sia un DFA con meno di 2^n stati, $\exists w_1, w_2$ t.c. $|w_1| = |w_2| = n$ AND $w_1 \neq w_2$ AND condividono uno stato del DFA.



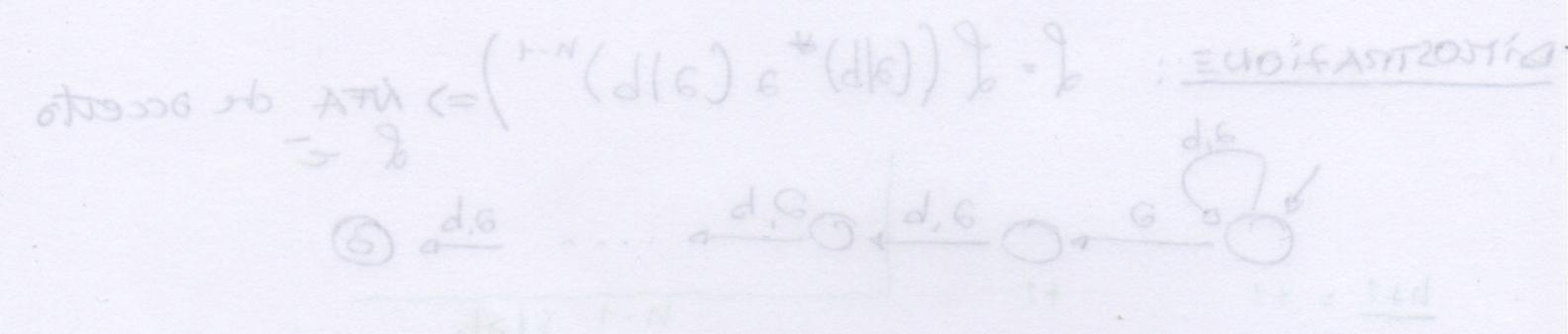
dal momento che $w_1 \neq w_2$ le differenze tra di loro è di almeno un simbolo, considerando quello più a destra.

$$\Rightarrow \begin{cases} \text{se } w_1 = y_1 a x, & w_2 = y_2 b x \\ \text{o } w_1 = y_1 b x, & w_2 = y_2 a x \end{cases}$$

In altre parole:



- z_1 e z_2 raggiungono lo stesso stato finale in DFA
- con $y_1 \in L$, sappiamo che s è finale
- contraddizione: $y_2 \notin L$ perché s' non può essere finale in quanto alla posizione n -esima non matcha lo stato in comune, cioè " a ".



negli st. w stringhe lunghe - : EQUAZIONE
stringhe lunghe "s" avendo avuto "d" -
non avendo avuto "d" cioè $\{d, G\}^*$ illeso

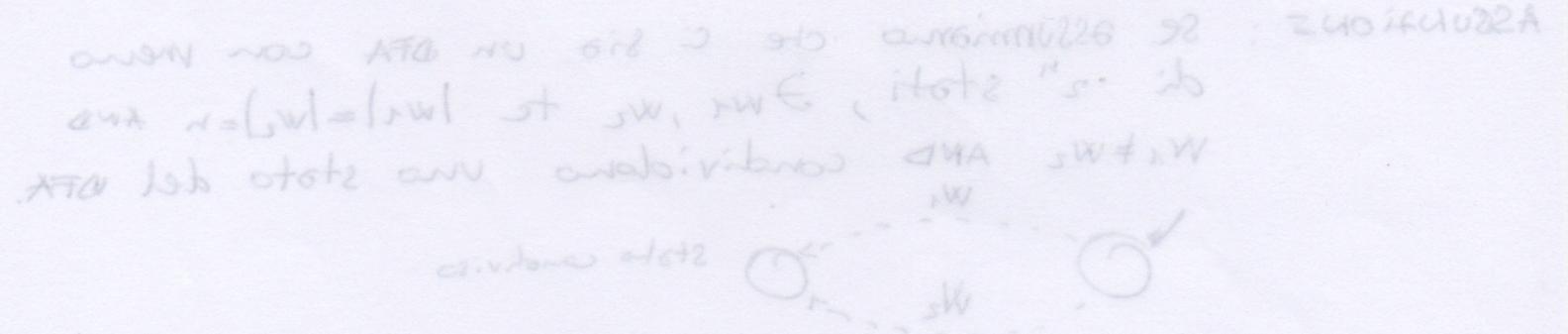
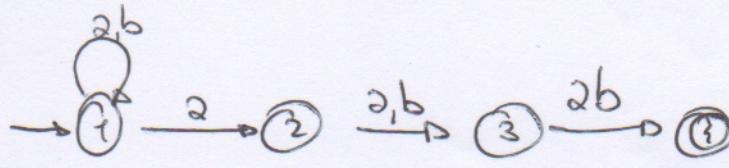


abb. un stringhe di $d \neq w$ gli obblighi lab
obblighi, dobbiamo avendo il \rightarrow or
entro e rig alleq

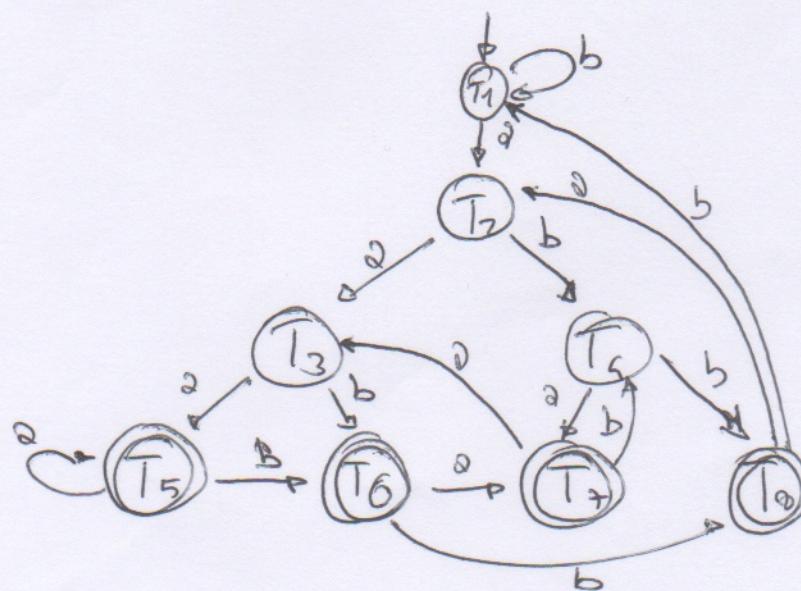
$$xG, X = w \quad , \quad xG, X = v \quad \leftarrow$$
$$xG, X = w \quad , \quad xG, X = v \quad 0$$

ESEMPPIONFA

$$L = \left((a/b)^* a (a/b)^{3-1} \right)$$

trasformazione attraverso SUB-SET CONSTRUCTION:

| | a | b |
|--------------------|-------|-------------|
| $T_1 : 1$ | T_2 | T_1 |
| $T_2 : 1, 2$ | T_2 | \bar{T}_3 |
| $T_3 : 1, 2, 3$ | T_5 | T_6 |
| $T_4 : 1, , 3$ | T_7 | \bar{T}_8 |
| $T_5 : 1, 2, 3, 4$ | T_5 | T_6 |
| $T_6 : 1, , 3, 4$ | T_7 | T_8 |
| $T_7 : 1, 2, , 4$ | T_3 | \bar{T}_5 |
| $T_8 : 1, , , 4$ | T_2 | \bar{T}_4 |



COMPLESSITÀ

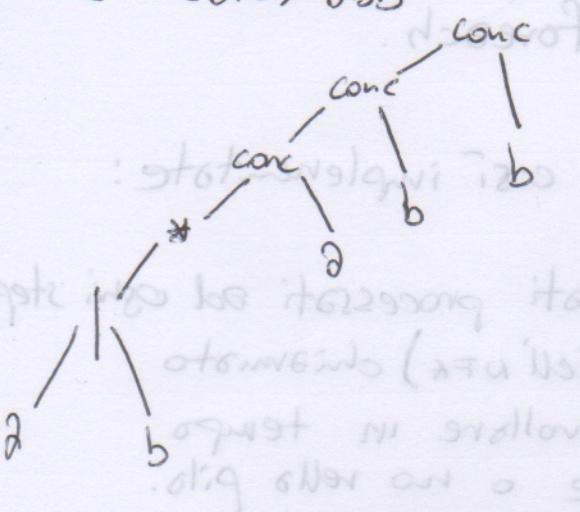
Vengono analizzate in dettaglio le complessità di tutte le funzioni che portano a capire se la stringa appartiene ad un linguaggio.

COMPLESSITÀ DELLA COSTRUZIONE A' THOMPSON

L'algoritmo di Thompson prevede i seguenti step.

- generare un syntax tree detto reg-exp.

es: $(ab)^*abb$



esprime una gerarchia di operandi.

costo: $O(|r|)$

dove " r " è la somma di tutti gli operatori e operandi.

- aggiungere al più di due stati rispetto agli originali.
- ogni stato non finale ha un arco uscente con un simbolo dell'alfabeto, oppure due archi uscenti con " ϵ ".

alla fine della costruzione:

- numero di stati $\leq |r| + 2$
- numero di transitioni $\leq 2(|r| + 2)$

globalmente il costo computazionale richiesto da Thompson è $O(|r|)$

$(s \mid r \mid b) \circ \leftarrow$

COMPLESSITÀ SUBJECT construction

FATTO = 39/100

Analizzando l'algoritmo della subject construction, le parti che maggiormente contribuiscono ad aumentare la complessità sono:

[...]
foreach ($a \in A$)

$$\begin{aligned} A &= \bigcup_{t \in T} \text{more}_n(t, a) \\ B &= \epsilon\text{-closure}(A) \end{aligned}$$

[...]

queste due funzioni sono ripetute per ogni stato del DFA e, per ognuna di queste, scansione tutte le lettere dell'alfabeto con il foreach.

Queste due funzioni possono essere così implementate:

- T è salvato come una `LinkedList`
- uno stack per indicare tutti gli stati processati ad ogni step
- bit vector (inizializzato con gli stati dell'NFA) chiamato `alreadyOn[]` che viene usato per controllare in tempo costante se uno stato è presente o no nello stack.

$t = t.takeNext()$

while ($t \neq \text{null}$)

clean stack

set all state `alreadyOn` on false

foreach ($s \in \text{more}_n(t, a)$)

(if `alreadyOn[s]` then `closure(s)`)

`closure(s)`

push s nello stack

`alreadyOn[s] = true`

foreach ($t \in \text{more}_n(s, \epsilon)$)

if ($\neg \text{alreadyOn}[t]$) then `closure(t)`

costano $O(n+m)$

dove n : non stati non deterministici

m : non transizioni non deterministiche

Globalmente:

$O(nd \cdot |A| \cdot (n+m))$

nd : num stati deterministici

tenendo conto che si arriva dalla costruzione di Thompson:

A dipende da $r \Rightarrow$

$n \leq 2r$ e $m \leq r$

$\Rightarrow O(nd \cdot |r|^2)$

SIMULAZIONE SU NFA

Gli algoritmi di simulazione si possono applicare sia sugli NFA che sui DFA, ovviamente avendo un costo computazionale differente.

Input: $w \$$, NFA

out: yes, no

algoritmo:

$$c_states = \epsilon\text{-closure}(\{s_0\})$$

$$c_symbol = nextChar()$$

$$\text{while } (c_symbol + \$)$$

$$A = \bigcup_{s \in c_states} \text{move}(s, c_symbol)$$

$$c_states = \epsilon\text{-closure}(A)$$

$$c_symbol = nextChar()$$

$$\text{if } (c_state \cap F \neq \emptyset) \text{ then yes}$$

else no

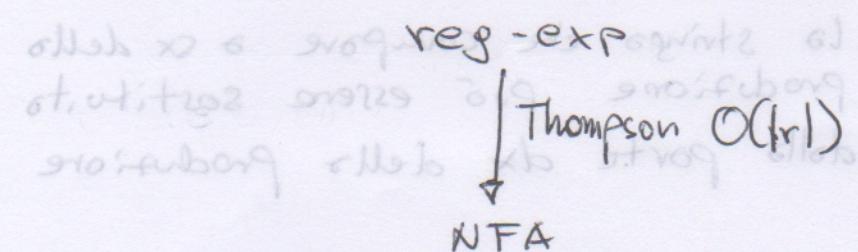
costo

$$O(h+m)$$

complessivamente l'algoritmo costa computazionalmente:

$$O(|w| \cdot |r|)$$

RIASSUMENDO



simulazione di w
 $O(|w| \cdot |r|)$

Subset Construction
 $O(nd \cdot |r|^2)$

simulazione di w
 $O(|w|)$

GRAMMATICHE GENERATIVE

ATU 02 340FAJUTR

G = (V, T, S, P)

- V: vocabolario, insieme di simboli
 T: sottoinsieme di V di simboli terminali
 S: simbolo (non finale) iniziale della grammatica
 P: insieme di produzioni

L'insieme di produzioni ha la forma chi
 $\vec{x} \rightarrow z$ dove \vec{x} stringa di elementi in V^+ che contiene almeno un elemento in $V \cap T$.
 \vec{z} è una stringa in V^* .

CONVENTIONI

- lettere maiuscole per simboli non terminali in $V \setminus T$
- lettere minuscole per simboli terminali in T
- lettere greche per stringhe su V^*

ESEMPIO

$$G = (\{A, S, a, b\}, \{a, b\}, S, P)$$

$$\begin{aligned} P: \quad & S \rightarrow aAb \\ & aA \rightarrow aaAb \\ & A \rightarrow \epsilon \end{aligned}$$

la stringa che compone \vec{x} della produzione può essere sostituita dalla parte \vec{x} della produzione

Dalle produzioni si ricava che

$$L(G) = \{a^n b \mid n \geq 0\}$$

w i b s o r o f f o r m a
 $(l_v l_b)O$

w i b s o r o f f o r m a

$$(l_v l_b)O$$

DEF: Una stringa γ deriva direttamente dalla stringa ν in $G = (V, T, S, P)$, denotato da $\nu \xrightarrow{G} \gamma$ (gamma deriva direttamente da ν nella grammatica G)
Se:

$$\nu = \epsilon \lambda^* , \quad \lambda \rightarrow \beta \in P \Rightarrow \gamma = \epsilon \beta \lambda^* \quad \text{def. } \nu = \epsilon \lambda^*$$
$$\{\gamma \mid \gamma = (\beta) \}$$

DEF: Una stringa γ deriva da ν , denotato da $\nu \xrightarrow{*} \gamma$ se esiste una sequenza di stringhe $w_0 \dots w_n$ t.c. $\nu = w_0$, $\gamma = w_n$ AND w_{i+1} deriva direttamente da w_i in G

DEF: Il linguaggio generato da G , denotato da $L(G)$ è dato da:

$$L(G) = \{ w \text{ t.c. } w \in T^*, \quad S \xrightarrow{*} w \}$$

DEF: Una grammatica si dice context-free se ogni produzione ha la forma $A \rightarrow \gamma$ con $A \in V \cap T$.
Se G non è context-free \Rightarrow context dependent.

ESMPI alla strukturerhövda och egenvärde end : T=4
 $S \rightarrow aSb$ och struktur $(q, 2, T, V) \Rightarrow n \in q$
 $S \rightarrow ab$ eller q och $f(G) = \{a^h b^h \mid h \in \omega\}$ med
: 92

$S \rightarrow AB$ $\Leftrightarrow q \in q \in b \in q \in q \in q$: T=4
 $A \rightarrow aA \mid a$ $f(G) = \{a^+ b^+\}$
 $B \rightarrow Bb \mid b$ struktur, q och egenvärde end : T=4
egenvärde i struktur enligt $\Rightarrow q^* \subseteq q$

$S \rightarrow aSBC \mid abc$ struktur $\Rightarrow n \in w$ och strukturerhövda
 $C \rightarrow Bc$ $f(G) = \{a^h b^h c^h \mid h \in \omega\}$
 $bB \rightarrow bb$ struktur, \Rightarrow egenvärde II : T=4
: struktur $\Rightarrow (\neg)$

$$\{w \in q^*, \text{ s.t. } w\} = (\neg)$$

inga \Rightarrow struktur \Rightarrow struktur \Rightarrow struktur : T=4
 $T/V \ni A$ $\Rightarrow A$ är struktur \Rightarrow struktur
struktur \Rightarrow struktur \Rightarrow struktur \Rightarrow struktur \Rightarrow struktur \Rightarrow struktur

ANALISI LESSICALE

Analizzate le seguenti di parole del linguaggio di programmazione con grammatica libera dal contesto.

Esempio di grammatica context free:

$$\begin{aligned} S &\rightarrow \text{if } E \text{ then } S \text{ else } S | \epsilon \\ E &\rightarrow T \text{ rrelOp } T \mid T \\ T &\rightarrow \text{id} \mid \text{num} \end{aligned}$$

Per identificare ogni keyword del linguaggio si utilizzano reg exp molto semplici, che poi verranno combinate in un enorme automata.

rif = if

rthen = t.h.c.n

relse = e.l.o.e

rdigit = 0 | ... | 9

rdigits = rdigit (rdigit)*

rid = (a | ... | z | A | ... | Z) (a | ... | z | A | ... | Z | 0) ... | 9)*

whitespaces = ("blank" | "tab" | "newline") ("blank" | "tab" | "newline")*

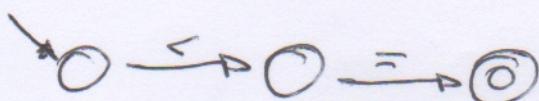
vnum = rdigits · voptionFraction · voptionExponent

voptionFraction = (.rdigits) | E

voptionExponent = (E (+|-|ε) rdigits) | E

rrelOp = < | ≤ | > | ≥ | = | <>

AUTOMA PER rrelOp



tutto tranne
=, >
return (rrelOp, lessEqual)

return (rrelOp, less)

AND RETRACT

return (rrelOp, lessEqual)

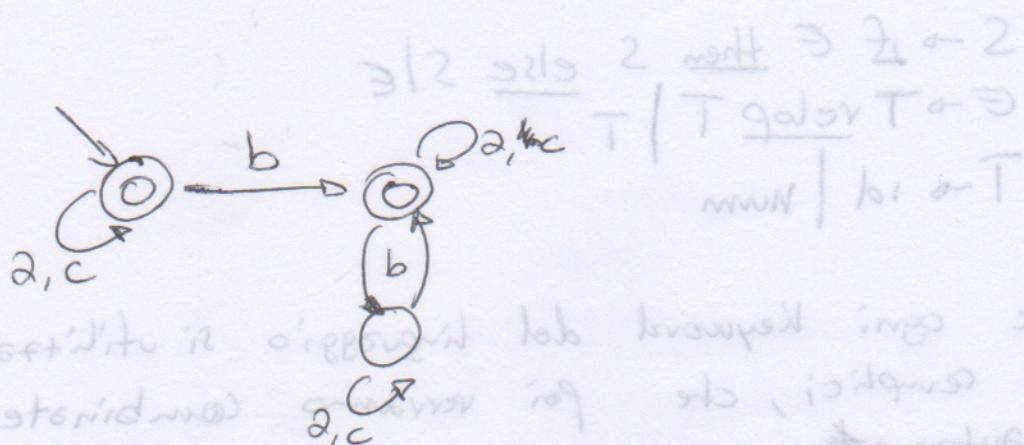
informazione
semantica, verrà
utilizzata per
copiare a cosa si riferisce
il token.

: torna indietro di un carattere visto
stringa input, perché dopo < può
esserci un'espressione così

... < (s+2)

Esercizi

- Sia D il min-DFA che per il riconoscimento del linguaggio sua $\{a, b, c\}$ che consiste in tutte le parole in cui 'b' occorre K volte con $K = \text{dispari} \cup \{0\}$.
 Dire quante transizioni etichettate da 'c' e quanti stati ha D .



$$3! \left(\text{triangleright} \right) = \text{multigalor}$$

$$3! \left(\text{triangleright} (3! - 1) \right) = \text{tricor} \times \text{multigalor}$$

$$\leftrightarrow | = | = | < | < | = | > | > = \text{galor}$$

$$*(p| \dots |o|s| \dots |A|s| \dots |g)(s| \dots |A|s| \dots |g) = \text{bir}$$

$$*(\text{"adverb"} | \text{"det"} | \text{"Nmod"})(\text{"adverb"} | \text{"det"} | \text{"Nmod"}) = \text{zazogzazogzazog}$$

galor si = ANTONIA

(2222, galor) multor

multor
multor, multor
reg stoffidu
multor boc a zog
zog

$\circ_4 = \circ_4 \circ_4 \circ_4$

multor /
 $\circ = \circ$

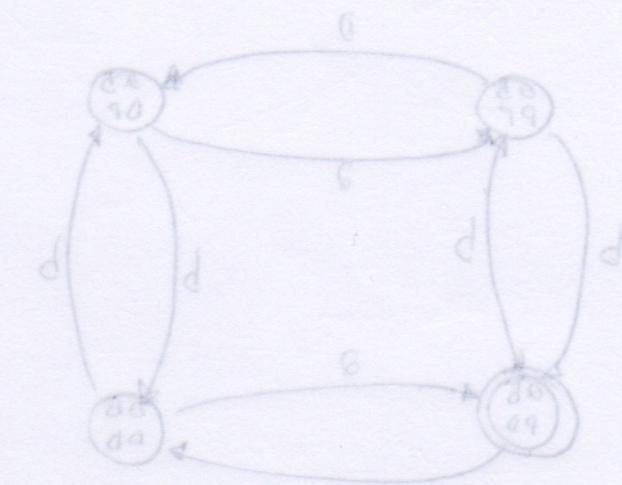
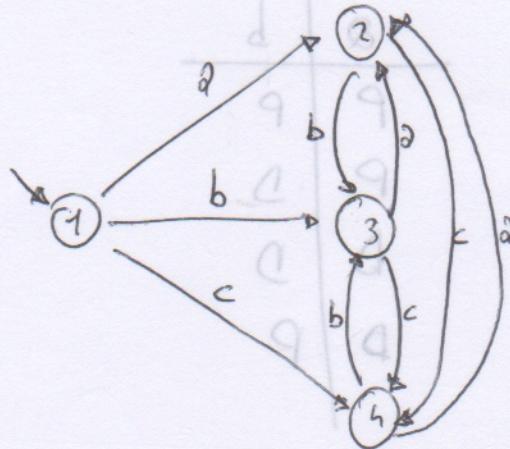
(2222, galor) multor

che restava su di questo binomio era: TJANTES qua
- cog > ogab - zog, tigni cogniti
- zog > zogab - zog, tigni cogniti

(s+) > ...

ESERCIZI

1) formare il min-DFA per il riconoscimento delle parole non vuote su $\{a, b, c\}$ che non contengano occorrente di "aa", "bb", "cc".



2) Siano $r_1 = (r/s)^*$ e $r_2 = (r^*/s^*)$ per r, s reg. exp. generiche. Dire, motivando la risposta se $L(r_1) = L(r_2)$ o no.

$$r = a, s = b \Rightarrow r_1 = (a/b)^* \quad \text{e} \quad r_2 = (a^*/b^*)$$

$$w = ab \Rightarrow \cancel{w \in L(r_1)}, w \notin L(r_2)$$

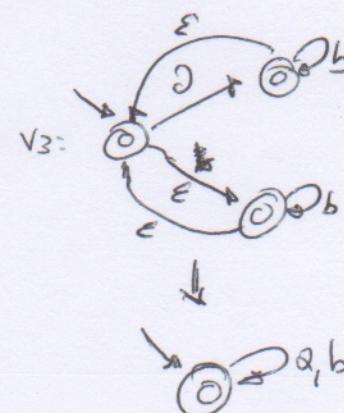
□

3) Siano $r_1 = (a/b)^*$, $r_2 = (a^*/b^*)$, $r_3 = ((\epsilon/a)b^*)$.

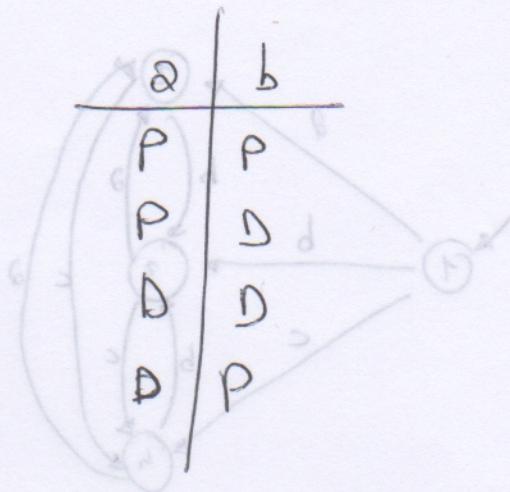
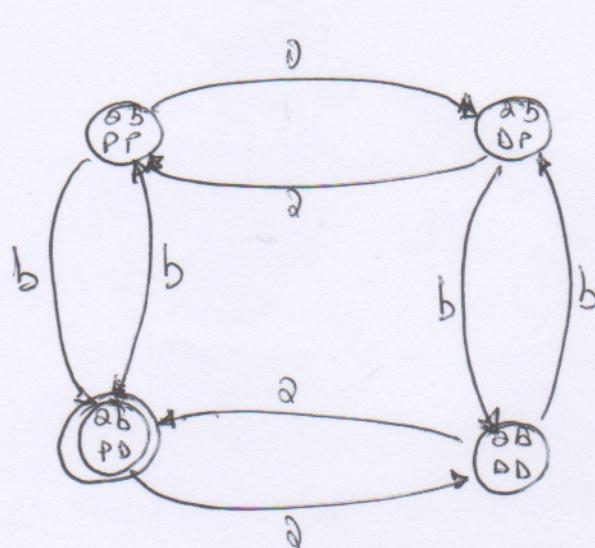
$$\text{Dimostrare } L(r_1) = L(r_2) = L(r_3)$$

$$r_1: \xrightarrow{\epsilon} \textcircled{0} \xrightarrow{a,b} \textcircled{0}$$

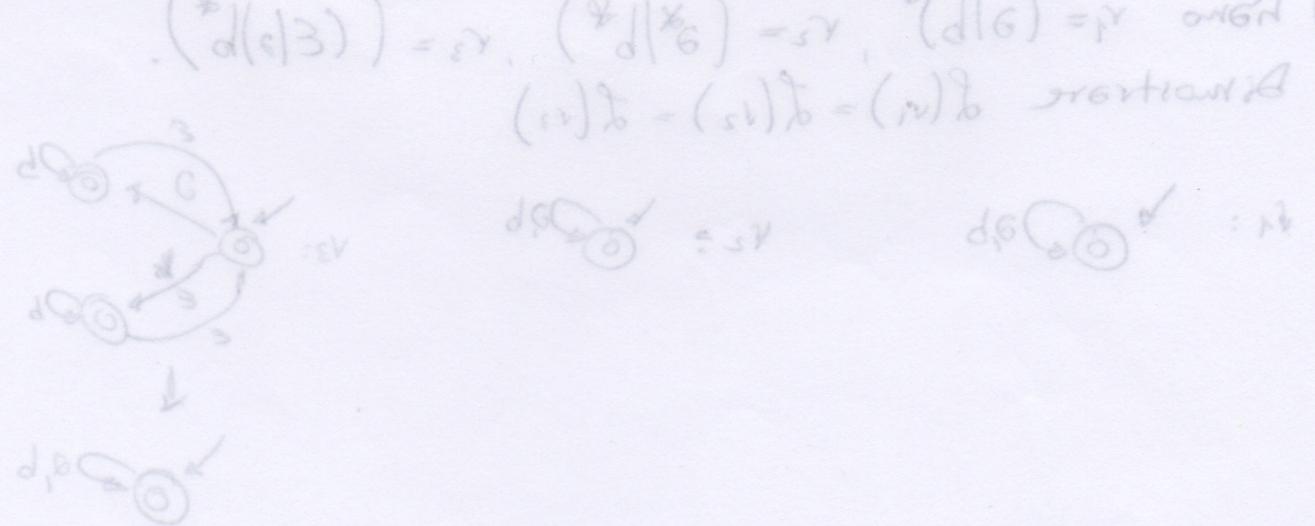
$$r_2: \xrightarrow{\epsilon} \textcircled{0} \xrightarrow{a,b} \textcircled{0}$$



4) Fornire il min-DFA che riconosca le parole $t \in L = \{w \in \{a,b\}^* \mid \text{il simbolo "a" compare un numero pari di volte e il simbolo "b" un numero dispari di volte}\}$

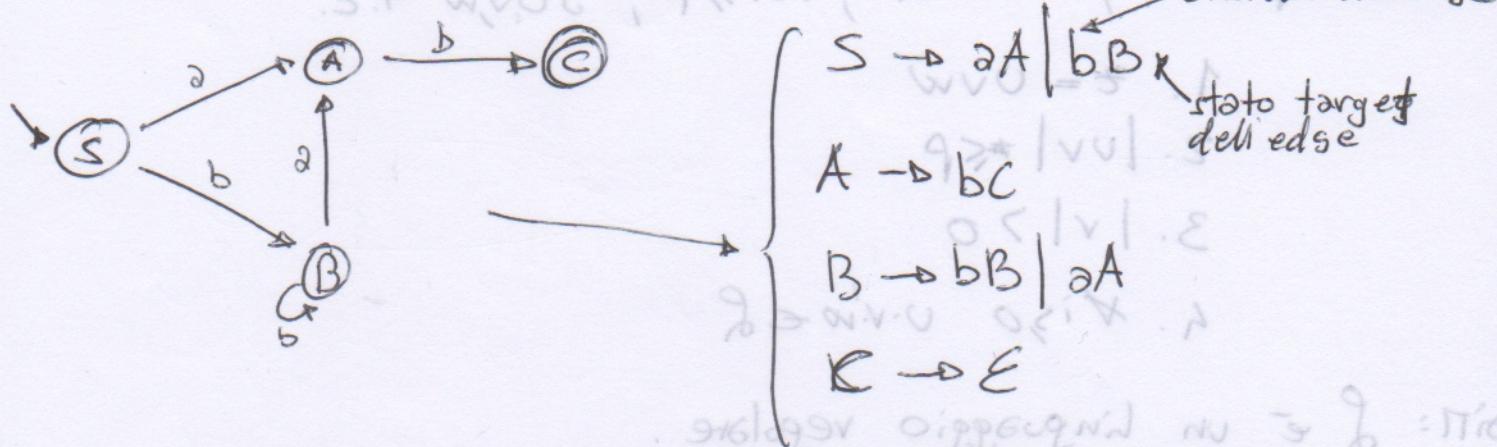


$$(\ast d \mid \ast b) =_{sr} \ast (d \mid b) =_{sr} \ast (d \mid b) =_{pr} d = z, b = y \\ (\ast r \mid \ast b) =_{sr} \ast (r \mid b) =_{sr} \ast (r \mid b) =_{pr} r = w$$



GRAMMATICHE REGOLARI

Dato un automa è possibile ricavare la grammatica del linguaggio nel seguente modo:



Tutte le grammatiche che rispettano i seguenti vincoli:

- a sinistra della produzione hanno un simbolo non terminale
- ogni produzione è composta almeno da un simbolo terminale e uno non terminale ($\circ E$)

Viene chiamato REGOLARE

Supponiamo di avere questo linguaggio $\{a^nb^m \mid n, m \geq 0\}$.

Di questo linguaggio non è possibile creare un automa a stati finiti in quanto richiederebbe un numero infinito di stati; quindi non è possibile nemmeno descrivere la grammatica. Il riconoscimento di questi tipi di linguaggi vengono eseguiti tramite l'utilizzo di una pila.

LEMMA PER I LINGUAGGI REGOLARI

Sia L un linguaggio regolare.

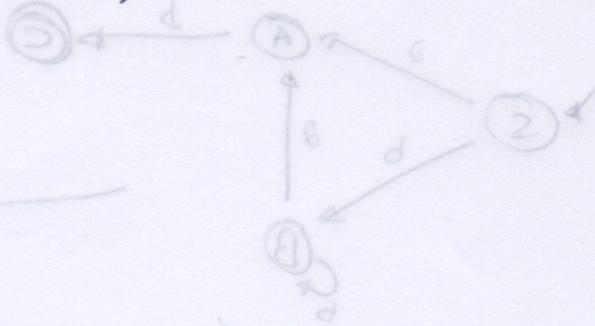
Allora $\exists p \in \mathbb{N}$, $\forall z \in L$, $|z| \geq p$, $z = uvw$ t.c.

$$1. z = uvw$$

$$2. |uv| \leq p$$

$$3. |v| > 0$$

$$4. \forall i \geq 0 \quad uv^i w \in L$$

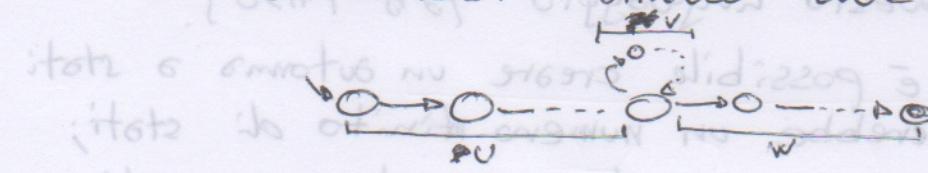


Dim: L è un linguaggio regolare.

$\Rightarrow \exists M$ automa a stati finiti M t.c. $L(M) = L$

$\Rightarrow p$ è uguale al numero di stati: automa M è osserviamo che il cammino più lungo sull'automa M che parte dallo stato iniziale e termina in uno stato finale e passi al più una volta per qualunque stato, è lungo al massimo ($p-1$)

\Rightarrow se $z \in L$ t.c. $|z| \geq p$ allora nel cammino di riconoscimento di z c'è almeno uno stato che viene attraversato almeno due volte.



6.9.15 - 25

Questo lemma viene usato per dimostrare che un linguaggio NON è regolare. Viene dimostrato per contraddizione sulle quattro proprietà dei linguaggi regolari (P_1, \dots, P_n)

tesi:

$$\exists p \ \forall z \ \exists vrw . \underbrace{(P_1 \wedge P_2 \wedge P_3 \wedge P_n)}_P \rightarrow \text{arretrabile}$$

Per procedere per contraddizione dobbiamo negare la tesi:

$$\begin{aligned}
 & \neg (\exists p \ \forall z \ \exists vrw . P) = \neg \forall p \ \exists z \ \neg (\exists vrw . P) = \\
 &= \forall p \ \forall z \ \forall vrw \ \neg (\exists vrw . P) = \forall p \ \forall z \ \forall vrw \ \neg (P_1 \wedge P_2 \wedge P_3 \wedge P_n) = \\
 &= \forall p \ \forall z \ \forall vrw \ \neg ((P_1 \wedge P_2 \wedge P_3) \wedge P_n) = \quad \text{applicando leggi} \\
 &= \forall p \ \forall z \ \forall vrw \ \neg (P_1 \wedge P_2 \wedge P_3) \vee \neg P_n = \quad \text{di de Morgan} \\
 &= \forall p \ \forall z \ \forall vrw . ((P_1 \wedge P_2 \wedge P_3) \rightarrow \neg P_n) \quad \neg A \xrightarrow{\text{implicazione}} B = \neg A \vee B
 \end{aligned}$$

$\forall p \in \mathbb{N}^+, \exists z \in L, |z| \geq p, \text{arretrabile t.c.}$

$(z = vrw) \wedge |v| < p \wedge |r| > 0 \rightarrow \exists i > 0, v_i w \notin L$

ESEMPIO $L = \{a^h b^h \mid h > 0\}$ non è regolare.

DIM: Supponiamo che L sia regolare.

Se vogliamo $p \in \mathbb{N}^+$ arbitrario.

Consideriamo $z = a^p b^p$.

$\Rightarrow |z| \geq p$ e qualunque u, v, w siano se $z = uvw$ e $|uv| \leq p$ e $|v| > 0$

allora v contiene solo "a" ed almeno uno "b"

allora $uv^2w = a^p b^p$ per $j > 0$

allora $uv^2w \notin L$ che contraddice il lemma \square

$B \cap A^c = \emptyset \Leftrightarrow (B \cap A) = \emptyset$

$\vdash \neg \neg f \vdash f$
 $\neg \neg (A \rightarrow B) \vdash A \rightarrow B$

Esercizi Pumping Lemma

$L = \{ww^R \mid w \in \{a,b\}^*\}$, con R funzione di reverse

Dim: - Sappiamo L regolare.

- Sappiamo p come pumping length

- Dobbiamo scegliere una $z = uvw$ che ^{vai più una} ~~contraddizione~~ contraddica la ~~z~~ $\Rightarrow z \neq uvw$.
- $z = a^{p+k}$: NON VA BENE! perché qualunque uvw scelgo, uvw è ~~f~~ sempre
- $z = abba$: NON VA BENE! perché viola un requisito della ~~z~~ $\Rightarrow z \neq abba$ del lemma $|z| > p$, ovvero che non dipende da p .
- $z = a^pb^pb^pa^p$: questa z può funzionare

$$z = a \underbrace{\dots a}_p \underbrace{b \dots b}_p \underbrace{b \dots b}_p \underbrace{a \dots a}_p = uvw$$

Se ipotizzo $|v| \leq p$ allora v contiene solo lettere "a" e v ne contiene almeno una per via di $|v| > 0$.

$$\text{Se } i=2 \Rightarrow uv^iw = a^k \underbrace{b^p b^p}_{\text{con } K > p \text{ perché}} a^p$$

$\Rightarrow uv^iw \notin L$ perché $K > p$
(regola \downarrow è stata regata)

con $K > p$ perché ho aggiunto delle lettere "a" nella prima parte

□

$$L = \{a^i b^j \mid i > j, i, j > 0\}$$

NOTES 2019/2020 - 15/02/2020

- BIM: - Suppongo che L sia regolare.
 - Considero p arbitraria con le caratteristiche del lemma.
 - Sceglio $z = uvw = a^{p+1} b^p$ t.c. $|z| > p$ (\rightarrow è vera)

$$z = a \underbrace{a^p}_{p+1} b \underbrace{b^p}_{p} = uvw$$

se $|uv| \leq p$ allora v contiene solo "a", ma se contiene almeno una "b" $|v| > 0$.

allora per $i=0$ ottengo una parola che non appartiene al linguaggio L

$$uvw = a^k b^p, \text{ e } k \leq p \Rightarrow \notin L$$

□

$$L = \{a^h b^k c^{h+k} : h, k > 0\}$$

\Rightarrow suppongo che sia regolare

\Rightarrow considero la pumping length p

$$\Rightarrow \text{considero } z = a^p b^p c^{2p} = uvw \quad |z| > p$$

□

Con le condizioni che

$$|uv| \leq p \quad |v| > 0$$

uv deve contenere solo 'a' o 'b' o 'c'

$$\Rightarrow \text{quindi } uv^i w = a^p b^p c^{2p}, i < p, \notin L$$

□

PUMPING LEMMA LINGUAGGI LIBERI

Sia L un linguaggio regolare e libero dal contesto.
 Allora $\exists p \in \mathbb{N}^+$ t.c. $\forall z \in L : |z| > p$.

$$\exists u, v, w, x, y \text{ t.c. } z = uvwxy$$

$$\wedge |vwx| \leq p$$

$$\wedge |vx| > 0$$

$$\wedge \forall i \in \mathbb{N} : i > 0. uvwxy^i \in L$$

DIM: Essendo L libera dal contesto, \exists una grammatica $G = (V, T, S, P)$ t.c. $L = L(G)$, G grammatica libera.

Possiamo supporre che G sia in forma normale di Chomsky (no ridondante, massimo uno ϵ -produzione per il simbolo iniziale).

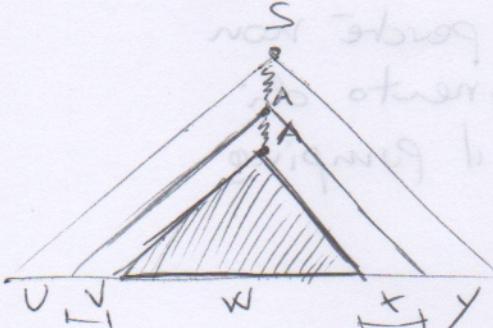
Sia $n = |VT|$, cioè il numero di non-terminali in G .

Definiamo p come la lunghezza della parola più lunga che può essere derivata da S utilizzando al più $n+1$ passi di derivazione.

Allora se consideriamo $z \in L$ t.c. $|z| > p$, l'albero di derivazione di z è t.c.: esiste un cammino in cui

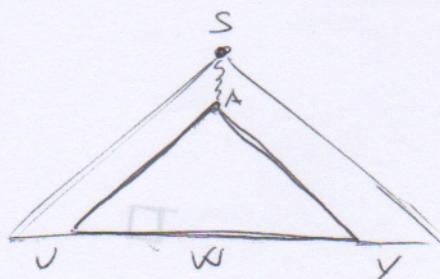
lo stesso non-termionale compare 2 volte.

Se $i = 0$ il sottoalbero più interno viene espanso fino a quello centrale.



derivazione di z da S

$p \geq |vwxy|$ perché è la sottostringa massima senza ripetizioni.



espansione se $i=0$

ESERCIZIO

$$L = \left\{ a^n b^n c^n \mid n \geq 0 \right\}$$

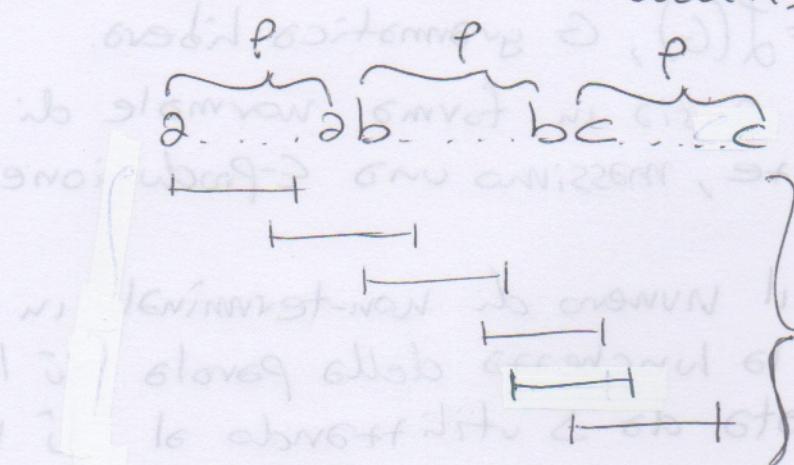
Supponiamo che sia libera dal contesto e regolare.

Scegliamo un arbitraria stringa p e prendiamo $z = a^p b^p c^p$.

$$\Rightarrow |z| > p \quad \& \quad z \in L$$

$$\Rightarrow \text{se } z = uvwxy \quad \& \quad |vwx| \leq p \quad \& \quad |vx| > 0$$

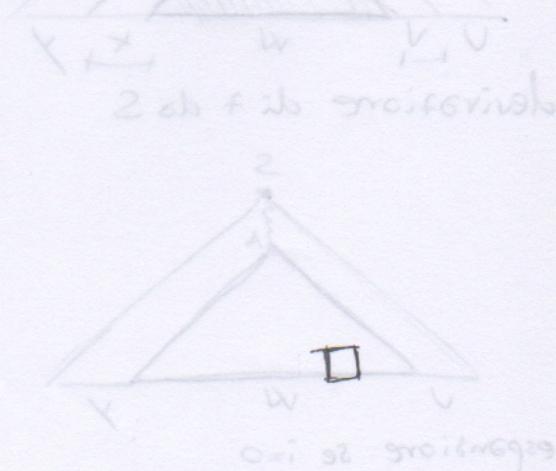
allora vwx non contiene alcuna occorrenza di a o di c , perché è la stringa centrale con un'arbitraria suddivisione.



se $|vwx| \leq p$

può essere uno di questi casi essendo $|v|$

\Rightarrow se consideriamo $uv^0w^0x^0y^0 \notin L$ perché non contiene il corretto bilanciamento di a e c , quindi contraddice il pumping lemma.



UNIONE DI DUE LINGUAGGI LIBERI

LEMMA: La classe dei linguaggi liberi è chiusa rispetto all'unione: se L_1, L_2 liberi, $L_1 \cup L_2$ è libero.

L è libero se esiste una grammatica libera

$$G \text{ tc. } L(G) = L$$

DIM: Supponiamo che L_1 e L_2 siano liberi allora $\exists G_1, G_2$ grammatiche libere t.c.

$$L(G_j) = L_j \text{ per } j=1,2$$

$$G_j = (V_j, T_j, S_j, P_j)$$

\Rightarrow previa videonominazione in caso di collisione dei simboli non terminali delle due grammatiche.

Definisco $G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\})$

con S nuovo simbolo iniziale t.c. $\notin V_1 \cup V_2$

INTERSEZIONE DI LINGUAGGI LIBERI

$$L_1 = \{a^nb^m|n \geq 0, m \geq 0\} \quad L_2 = \{a^nb^m|n \geq 0, m \geq 0\}$$

$$S \rightarrow AC$$

$$A \rightarrow aAb|ab$$

$$C \rightarrow c|\epsilon$$

$$S \rightarrow AB$$

$$A \rightarrow aA|B$$

$$B \rightarrow bB|c|bc$$

$L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\}$: I linguaggi liberi NON sono chiusi per intersezione.

DIM: Sia L_1 e L_2 come sopra e intersezione sia chiusa, dimostrando esistendo G_1, G_2 libere t.c. $L_1 = L(G_1)$ e $L_2 = L(G_2)$, che $L_1 \cap L_2$ sono due linguaggi liberi. Quindi: $L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\}$ si dimostra che non è libero \Rightarrow NON sono chiusi per intersezione

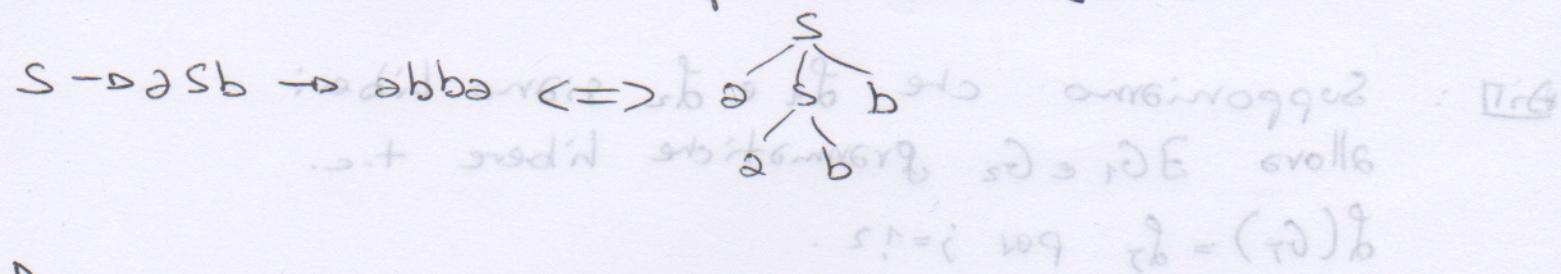
PARSING - ANALISI SINTATTICA

La fase di parsing si occupa di ricavare un albero di derivazione (derivation tree) partendo da una stringa del linguaggio.

ESEMPIO

Supponiamo la seguente grammatica: $S \rightarrow aSb | ab$

L'albero di derivazione di una possibile stringa è:



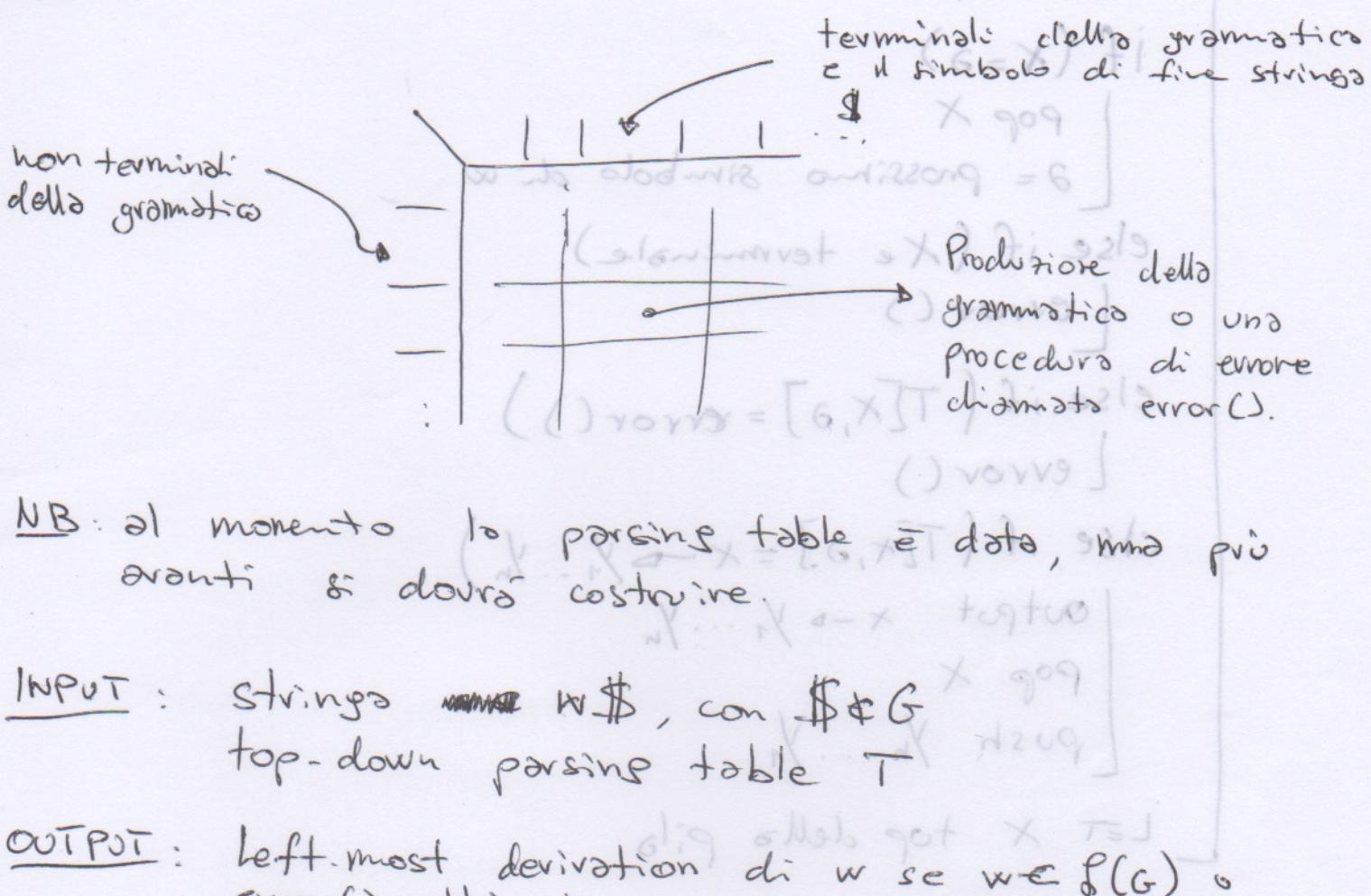
Dopo un passo di derivazione posso scegliere quale simbolo non terminale devirare: ho a disposizione due strategie:

- DERIVAZIONI LEFT-FIRST: trattate con top-down parsing
- DERIVAZIONI RIGHT-FIRST: trattate con bottom-up parsing

DETERMINISTIC - TOP DOWN PARSING

OTTIMIZZAZIONE

Questo tipo di parsing ha bisogno di una pre-elaborazione per la costruzione di una tabella chiamata parsing table.



NB: al momento lo parsing table è dato, si va più avanti e si dovrà costruire.

INPUT: stringa ~~w\$~~ $w\$$, con $\$\in G$
top-down parsing table T

OUTPUT: left-most derivation di w se $w \in f(G)$ o error() altrimenti.

INIZIALIZZAZIONE: $w\$$ in un input buffer,
 $\$S$ nella pila.

ALGORITMO:

ONCEPPE UNA GOT

STRUMENTO

LGT è primo simbolo di w

LET X = top della pila

while ($X \neq \$$)

if ($X = a$)

 pop X

 a = prossimo simbolo di w

else if ($X \in \text{terminal}$)

 error()

else if ($T[X, a] = \text{error}()$)

 error()

else if ($T[X, a] = X \rightarrow y_1 \dots y_n$)

 output $X \rightarrow y_1 \dots y_n$

 pop X

 push $y_n \dots y_1$

LET X = top della pila

ESEMPPIO

TORTO

TORTU

KCAT

| | | |
|-----------------------------------|---------|---|
| $E \rightarrow TE'$ | 32^T | - vogliamo usare il topdown parsing sulle strings |
| $E' \rightarrow + TE' \epsilon$ | 32^-3 | |
| $T \rightarrow FT'$ | | |
| $T' \rightarrow * FT' \epsilon$ | | $id * + id * id \$$ |
| $F \rightarrow (E) id$ | | |

Dalle produzioni viene data la parsing table:

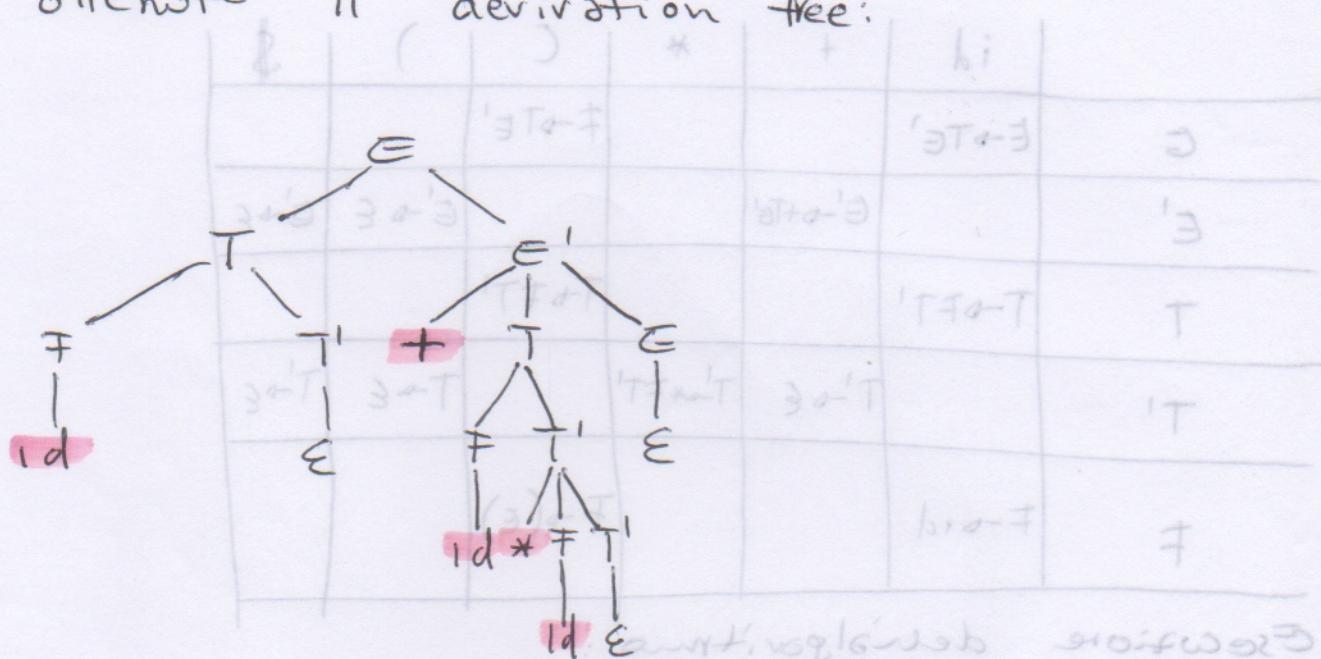
| | id | + | * | (|) | \$ |
|------|---------------------|---------------------------|----------------------|---------------------|---------------------------|---------------------------|
| E | $E \rightarrow TE'$ | | | $F \rightarrow TE'$ | | |
| E' | | $E' \rightarrow + TE'$ | | | $E' \rightarrow \epsilon$ | $E' \rightarrow E$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow FT'$ | | $T' \rightarrow E$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | | | $F \rightarrow (E)$ | | |

Esecuzione del algoritmo:

| STACK | INPUT | OUTPUT |
|-----------------|-------------------|---------------------------|
| $\$ E$ | $id + id * id \$$ | $E \rightarrow TE'$ |
| $\$ E T^*$ | | $T \rightarrow FT'$ |
| $\$ E' T^* F$ | | $F \rightarrow id$ |
| $\$ E' T^* id$ | $+ id * id \$$ | |
| $\$ E' T^*$ | | $T' \rightarrow \epsilon$ |
| $\$ E' T^+ id$ | | $E' \rightarrow TE'$ |
| $\$ E' T^*$ | $id * id \$$ | $T \rightarrow FT'$ |
| $\$ E' T^* id$ | | $F \rightarrow id$ |
| $\$ E' T^*$ | $* id \$$ | $T' \rightarrow * FT'$ |
| $\$ E' T^* F *$ | | |
| $\$ E' T^* F$ | $id \$$ | |
| | | $F \rightarrow id$ |

| STACK | INPUT | OUTPUT | PREDICTED |
|-------------|--------------|--------|-----------|
| \$ E T' id | + \$ + id id | id id | id id |
| \$ E T' | \$ | T' → E | E → E |
| \$ E b1b2b3 | b1b2b3 | E' → E | b1b2b3 |
| \$ | | b1b2b3 | b1b2b3 |
| \$ | | b1b2b3 | b1b2b3 |

Se lo stack è vuoto e la parola da parsare è finita, allora la parola appartiene al linguaggio e abbiamo ottenuto il derivation tree:



| TC9T00 | TC9U1 | NCAT2 |
|--------|-----------------|--------|
| '3T->3 | \$ b1 * b2 + b3 | '3\$ |
| 'TT->T | | 'T'3\$ |
| 'b1->T | | 'T'3\$ |
| | 'b1 * b2 + | 'T'3\$ |
| '3->T | \$ b1 * b2 + b3 | 'T'3\$ |
| '3T->3 | | 'T'3\$ |
| 'TT->T | 'b1 * b2 | 'T'3\$ |
| 'b1->T | | 'T'3\$ |
| | 'b1 * b2 + | 'T'3\$ |
| 'TT*>T | 'b1 * | 'T'3\$ |
| | | 'T'3\$ |
| 'b1->T | 'b1 | 'T'3\$ |
| | | 'T'3\$ |

COSTRUZIONE = PARSING TABLE = - PARSING TOP-DOWN

La parsing table è una tabella, dove troviamo sulle colonne i simboli terminali e sulle righe, non-terminali della grammatica. Questa tabella ha la funzione di accettare se una stringa è derivabile da una produzione della grammatica.

Data una produzione, dove lo colloco nella parsing table?

Date le tre possibili produzioni:

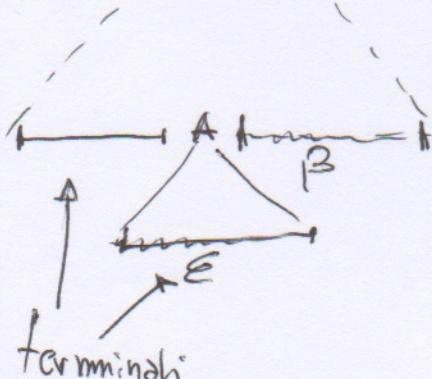
- $A \rightarrow \alpha$, $\alpha \neq \epsilon$: dove α è il risultato di un qualsiasi ^{numero} di passi di derivatione per arrivare ad un terminale.

Xai, $\alpha = \star$ dipi

va messo nella cella $[A, \alpha]$

- $A \rightarrow B\beta$: dove B è un non terminale, in questo caso dobbiamo procedere ricorsivamente fino a trovare un simbolo terminale.

- $A \rightarrow \epsilon$: Se A deriva ϵ va messo nella colonna della derivazione della esterna parola successiva



FUNZIONE FIRST

Dato $G = (V, T, S, P)$, $\forall X \in V$, l'insieme $\text{FIRST}(X)$
 è calcolato:

1. Se $X = a$ allora $\text{FIRST}(X) = \{a\}$
2. Se $X \rightarrow \epsilon \in P$ allora aggiungi ϵ in $\text{FIRST}(X)$

3. se $X \rightarrow Y_1 \dots Y_n \in P$ con $n > 1$
 allora:

$j = 1$

while $j \leq n$

aggiungi $a \in \text{FIRST}(X)$ ogni $b \in T$ t.c. $b \in \text{FIRST}(Y_j)$

if $\epsilon \in \text{FIRST}(Y_j)$

| $j++$

else

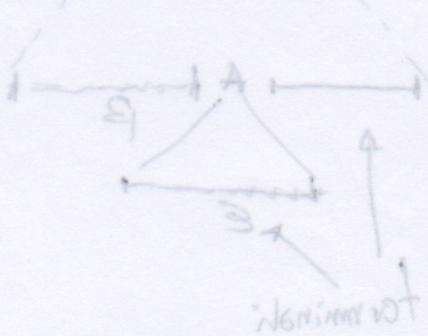
| break

if $j = n+1$
 | aggiungi $\epsilon \in \text{FIRST}(X)$

FIRST: ritorna l'insieme
 dei primi non terminali
 della produzione
 della grammatica

$\text{FIRST}(q), A \rightarrow q, q \in V^*$, $q = y_1 \dots y_n$

basta modificare l'algoritmo sostituendo ad X , q



ESEMPIO

Calcolare FIRST dalla seguente grammatica: $G = \{T, E, F, T'\}$

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE'| \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' | \epsilon \\ F &\rightarrow E(E) | id \end{aligned}$$

| | FIRST |
|------|----------------|
| E | (, id) |
| E' | + , ϵ |
| T | (, id) |
| T' | * , ϵ |
| F | (, id) |

FUNTION = FOLLOW

Dato $G = (V, T, S, P)$ ~~set di simboli terminali~~ ~~set di simboli non terminali~~ ~~set di produzioni~~ $V \subseteq NT$

$\text{Follow}(A)$ viene calcolato:

aggiungere $\$$ a $\text{Follow}(S)$

repeat

foreach $B \rightarrow \alpha A \beta$ add $\text{FIRST}(\beta) \cup \{\epsilon\}$ a $\text{Follow}(A)$
 if $B = \epsilon$ or $\epsilon \in \text{FIRST}(\beta)$ aggiungi $\text{Follow}(B)$ a $\text{Follow}(A)$

until saturation

ESEMPIO

$$\begin{aligned} S &\rightarrow aABb \\ A &\rightarrow Ac \mid d \\ B &\rightarrow CD \\ C &\rightarrow e \mid \epsilon \\ D &\rightarrow f \mid \epsilon \end{aligned}$$

| | FIRST | FOLLOW |
|-----|----------------|---------|
| S | a | \$ |
| A | d | c f b c |
| B | e f ϵ | b |
| C | e ϵ | f b |
| D | f ϵ | b |

GENERATION PREDICTING PARSING TABLE

INPUT: grammatica $G = (V, T, S, P)$

OUTPUT: tabella di parsing table

ALGORITMO:

foreach $A \rightarrow q \in P$

$\forall b \in \text{FIRST}(q)$, add $A \rightarrow q$ to $T[A, b]$

if $\epsilon \in \text{FIRST}(q)$, add $A \rightarrow q$ to every $T[A, +]$,

$x \in \text{FOLLOW}(A)$

SET error() in every empty cells

N.B.: se T non ha entry multi definite $\Rightarrow G$ è detto

LL(1)

controllo input
da s+ a dx

derivation
(s) \rightarrow left most

numero di simboli
che leggiamo alla volta
nella stringa in ingresso.

(A) word & (A) follow ignorare

non ignorare l'ultimo

| wor(l) | T2S1T | |
|--------|-------|---|
| # | q0 | 2 |
| s | b | A |
| p | q7s | 0 |
| d+ | q3s | 1 |
| d | q7 | 0 |

0.91T=23

ddAg → 2
hd1, A → A
0C → C
3 | s → C
3 | t → C

ESEMPPIO

| G: | WORD | FIRST | * | + |
|------------------------------------|------|-------|---|--------------------------------|
| $E \rightarrow TE^1 (+)$ | | | * | - A partire dalla grammatica |
| $E^1 \rightarrow +TE^1 \epsilon$ | | | * | G, generare la parsing table |
| $T \rightarrow FT^1$ | | |) | nel caso di un parser top-down |
| $T^1 \rightarrow *FT^1 \epsilon$ | | |) | predittivo. |
| $F \rightarrow (E) id$ | (| | * | |

| FIRST | FOLLOW | | |
|---------------|----------|--|----------|
| E (id | \$))) | | \$) |
| E^1 + E | | | \$) |
| T (id | + | | + \$) |
| T^1 * E | | | + \$) |
| F (id * | | | + * \$) |

| | + | * | (|) | id | \$ |
|-------|----------------------------|-----------------------|----------------------|----------------------------|----------------------|----------------------------|
| E | + | * | $E \rightarrow TE^1$ | | $E \rightarrow TE^1$ | |
| E^1 | $E^1 \rightarrow +TE^1$ | | | $E^1 \rightarrow \epsilon$ | | $E^1 \rightarrow \epsilon$ |
| T | | | $T \rightarrow FT^1$ | | $T \rightarrow FT^1$ | |
| T^1 | $T^1 \rightarrow \epsilon$ | $T \rightarrow *FT^1$ | | $T^1 \rightarrow \epsilon$ | | $T^1 \rightarrow \epsilon$ |
| F | | | $F \rightarrow (E)$ | | $F \rightarrow id$ | |

ESEMPIO

$G_1: E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

| | FIRST | FOLLOW | |
|-----|---------|----------------------|------------------|
| E | $+ * ($ | $\$ +) * +) * +)$ | $\$ \Rightarrow$ |
| T | $* ($ | $) \$ \Rightarrow$ | $\$ \Rightarrow$ |
| F | $(id$ | $id \Rightarrow$ | $\$ \Rightarrow$ |

| E | $+$ | $*$ | $($ | $)$ | id | $\$$ |
|-----------------------|---------|--|---------|-------------------|--------------------|---------|
| $E \rightarrow E + T$ | | $E \rightarrow E + T$ $E \rightarrow T$ | | $E \rightarrow T$ | $E \rightarrow id$ | |
| \vdots | \dots | \dots | \dots | \dots | \dots | \dots |
| \vdots | \dots | \dots | \dots | \dots | \dots | \dots |

Nel caso in cui la parsing table sia multidefinita
 \Rightarrow la grammatica G non è LLL(1) perché esibisce
la proprietà LEFT-RECURSION.

Una grammatica è left Recursion se per qualche terminale A vale:

$A \stackrel{+}{\Rightarrow} A_2$ dove " $\stackrel{+}{\Rightarrow}$ " significa in più possi

Quando si ha solo un passo per arrivare alla ricorsione
si dice Left-Recursion IMMEDIATA

$A \rightarrow A_2$



ALGORITMO PER ELIMINARE LA LEFT RECURSION DA G

1. eliminare la LEFT RECURSION IMMEDIATA
 2. eliminare la LEFT RECURSION REL CASO GENERALE
- : ottenendo now } $d/bA/s - A$

$$1| \quad A \rightarrow A_2 | \beta$$

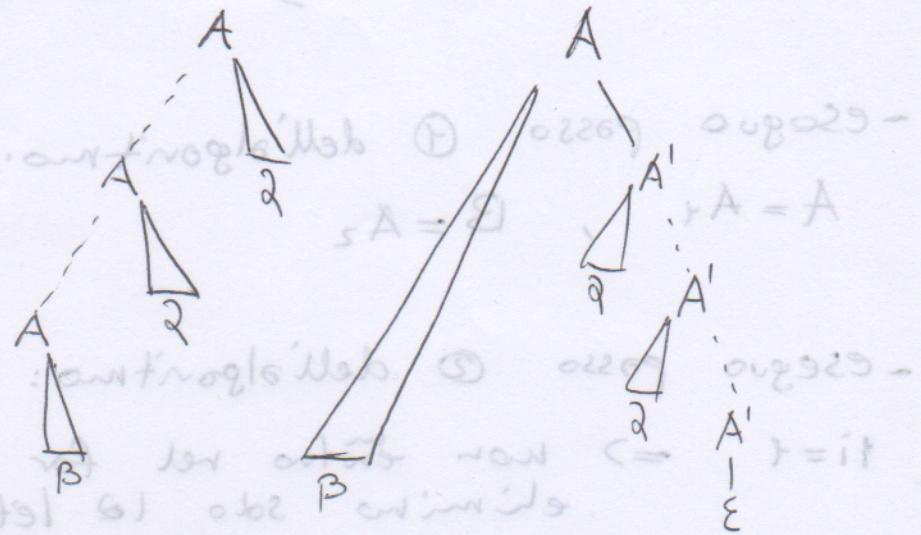
genera le stringhe

" β^2^n " con $n \geq 0$

possiamo riscrivere
come

$$A \rightarrow \beta A'$$

$$A' \rightarrow \beta A' | \epsilon$$



$$2| \quad A \rightarrow A_{21} | A_{22} | \dots | A_{2n} | \beta_1 | \beta_2 | \dots | \beta_n , \text{ di } \neq \epsilon \quad i=1 \dots n$$

IDEA: prendere un nuovo simbolo A'

sostituendo le produzioni con le seguenti produzioni:

$$\begin{array}{l|l} A \rightarrow \beta_1 A' & \dots \beta_n A' \\ A' \rightarrow \alpha_1 A' & \dots \alpha_m A' \end{array} \quad \epsilon$$

ALGORITMO:

1 - fissare un ordine dei non terminali $A_1 - A_n$

2 - for $i=1$ to n do

for $j=1$ to $i-1$ do

siano $A_j \rightarrow \delta_1 | \dots | \delta_k$ tutte le produzioni per A_j
replace $A_i \rightarrow A_j \gamma$ con $A_i \rightarrow \delta_1 \gamma | \dots | \delta_k \gamma$

eliminare la left recursion immediata su A_i

[d,a] class of reg rules sub

ESEMPPIO

G:

$$\left. \begin{array}{l} A \rightarrow B_a | b \\ B \rightarrow B_c | A_d | b \end{array} \right\}$$

Presenta una left recursion non immediata:
 $A \Rightarrow B_a \Rightarrow A_d a$

- eseguo passo ① dell'algoritmo.

$$A = A_1, B = A_2$$

- eseguo passo ② dell'algoritmo:

$i=1 \Rightarrow$ non entro nel for per i più interni ed elimino solo la left recursion immediata per $A_1 = A \Rightarrow A_1$ non ha left rec. immediata

$i=2, j=1 \Rightarrow$ guarda le produzioni $A_j = A_1 = A$.

Rimpiazza A_{ij} con tutte le possibili espansioni di $A_j = A_1 = A$ in tutte le possibili produzioni di $A_j = A_2 = B$

$$B \rightarrow A_d \rightsquigarrow B \rightarrow B_a d | b d$$

$$\Rightarrow G: A \rightarrow B_a | b$$

$$B \rightarrow B_c | B_a d | b d | b$$

Elimino la left recursion su $A_j = A_2 = B$

$$\Rightarrow G: A \rightarrow B_a | b$$

$$B \rightarrow b d B' | b B'$$

$$B' \rightarrow c B' | \epsilon$$

NON $\in LL(1)$ perché nella parsing table avremo due entry per la cella $[B, b]$.

Esercizio

AUTORES A EQUISAFS ISOTAT

Eliminare la left recursion dalla seguente grammatica:

$$G: E \rightarrow E + T \mid T \quad \left. \begin{array}{l} \text{presenta due left recursion} \\ \text{immediate sulla } E \end{array} \right\}$$

$$T \rightarrow T * F \mid F$$

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

$$G_2: E \rightarrow + T E' \mid E$$

$$E' \rightarrow T E'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$T \rightarrow F T'$$

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

Sia ora la seguente grammatica: A

$$S \rightarrow G$$

$$G \rightarrow P \mid PG$$

$$P \rightarrow id:R$$

- ① computare i FOLLOW
 ② dire se è LL(1)

$$R \rightarrow id R \mid \epsilon$$

1)

| | FIRST | FOLLOW | |
|---|----------------|--------|--------|
| S | id | \$ | \$ |
| G | id | id, | \$ |
| P | id | id, | \$, id |
| R | id, ϵ | id | \$, id |

2)

... id ...

G $\rightarrow P$
 G $\rightarrow PG$

entry multi-defined
 \Rightarrow NON \in LL(1).

FATTORIZZAZIONE A SINISTRA

Una grammatica è fattorizzabile a sinistra se ci sono due o più produzioni che:

$$A \rightarrow q_1 | \dots | q_n \text{ t.c. } \text{FIRST}(q_1) \cap \dots \cap \text{FIRST}(q_n) + \emptyset \neq \emptyset$$

Con il seguente algoritmo si può raccapriccire la fattorizzazione a sinistra:

repeat

foreach non terminale A

trova il più lungo prefisso comune a due opposte produzioni per A

if $\neq \emptyset$

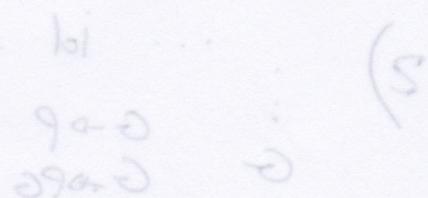
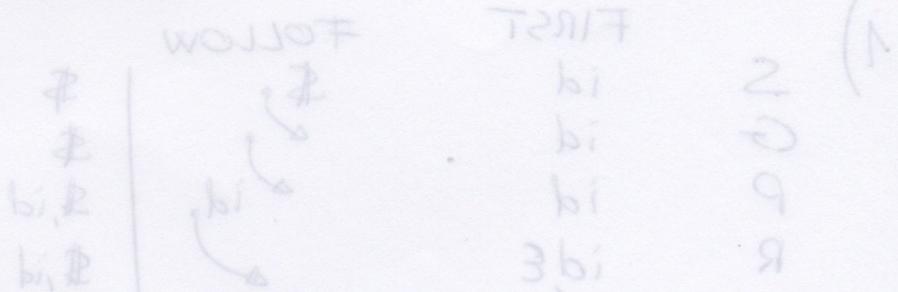
replace $A \rightarrow q_1 \beta_1 | \dots | q_n \beta_n | \gamma_1 | \dots | \gamma_k$ con

$A' \rightarrow q_1 \beta_1 | \dots | q_n \beta_n | \gamma_1 | \dots | \gamma_k$

A' $\rightarrow \beta_1 | \dots | \beta_n$ t.c. A' è nuovo simbolo

until nessuna coppia di produzioni per lo stesso terminale o prefisso comune.

benvenuti - it will work
(r) is non ←



BOTTOM UP - PARSING

Oblieettivo: costruire una derivazione right-most
costruendo l'albero di derivazione dalle foglie fino alla radice

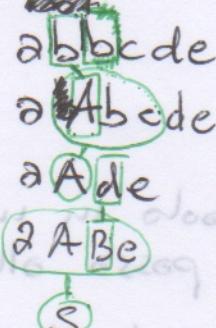
ESEMPIO

$$\begin{aligned} S &\rightarrow 2ABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

$$w = abcd$$

#, italiamet-nar, italiamet

ricostruzione bottom up:



Lo scopo del bottom up parsing è di trovare le sottostringhe β uguali alla parte dx della produzione $A \rightarrow \beta + c$.

Per esempio:

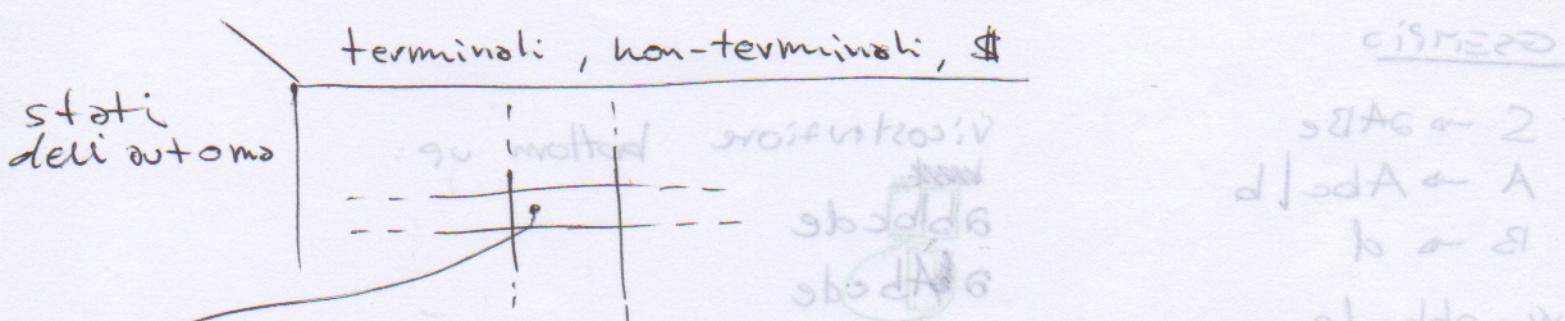
$S \xrightarrow{\text{right}} 2Aw \xrightarrow{\text{right}} 2ABw \xrightarrow{\text{right}} 2Abcd$

$$S \xrightarrow{\text{right}} 2Abcde \Rightarrow abcd$$

si sceglie la seconda "b" come HANDLE perché appare nella parte dx della produzione dell'at-

ALGORITMO BOTTOM UP PARSING

Parsing Table: è il rendering degli stati di un automa



- shift J: leggi il simbolo in input, e ponetilo nella pila e passa allo stato J.
 - reduce A \rightarrow B: reduce secondo la produzione $A \rightarrow B$
 - goto J: J è uno stato, può esserci in presenza di un non terminale.
 - Accept: ho svuotato la pila e ho \$ in input.

Input Buffer: c'è la stringa con terminatore \$

Stack: ci sono sequenze di simboli e stcoli:

| simbolo | stato | simbolo | Stato | ... |
|---------|-------|---------|-------|-----|
| | | | | → |

ALGORITMO

INPUT: grammatica G , parola w , parsing table M .
OUTPUT: derivazione right-most a rovescio se $w \in L(G)$
o error() altrimenti.

INIT: $w\$$ nel buffer e lo stato iniziale della parsing table nello stack.

ALGORITMO:

```
let b the first symbol of w$  
while true do  
    let S the top of the stack  
    if M[S,b] = shift n  
        push b onto the stack  
        push n onto the stack  
        let b the next symbol in input  
    elseif M[S,b] = reduce A → β  
        pop 2-|β| symbols of the stack  
        let m the state of the top of the stack  
        let n such that M[m,A] = goto n  
        push A  
        push n  
        output A → β  
    else if M[S,b] = accept  
        break  
    else error()
```

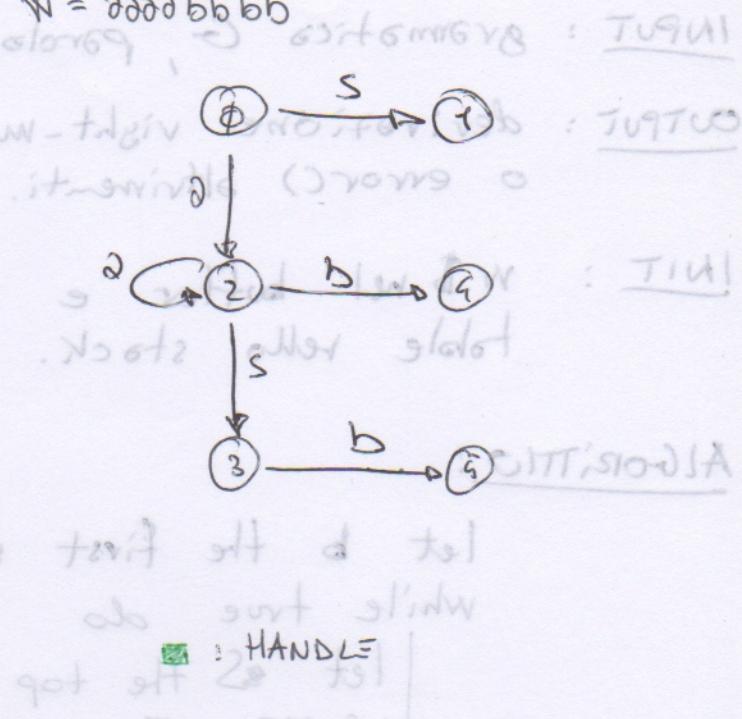
SEMPIO

OTTIMO SA

$$G: S \rightarrow aSb|ab$$

| | a | b | \$ | S |
|---|---------|-------------------|-------------------|--------|
| 0 | shift 2 | | | goto 1 |
| 1 | | | | accept |
| 2 | shift 2 | shift 4 | | goto 3 |
| 3 | | shift 5 | | |
| 4 | | reduce S → ab | reduce S → ab | |
| 5 | | reduce S → aSb | reduce S → aSb | |

$$W = aaaa bbbb bb$$



Pila \rightarrow

- \emptyset
- $\emptyset a2$
- $\emptyset a2 a2$
- $\emptyset a2 a2 a2 a2 a2$
- $\emptyset a2 a2 a2 a2 b5$
- $\emptyset a2 a2 a2 a2 S_3$
- $\emptyset a2 a2 a2 S_3 b5$
- $\emptyset a2 a2 S_3$
- $\emptyset a2 S_3$
- $\emptyset a S_3 b5$
- $\emptyset S_1$

buffer input

aaaa bbbb \$

output

$S \rightarrow ab$

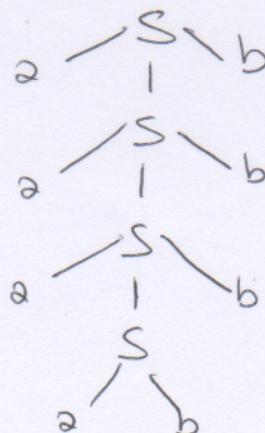
$S \rightarrow aSb$

$S \rightarrow aSb$

$S \rightarrow aSb$

Accept

Rileggendo al contrario l'output si produce l'albero di deriv:



ALGORITMO

INPUT: grammatica G , parola w , parsing table M .

OUTPUT: derivazione right-most a rovescio se $w \in L(G)$
o error() altrimenti.

INIT: $w\$$ nel buffer e lo stato iniziale della parsing table nello stack.

ALGORITMO

let b the first symbol of $w\$$
while true do

 let s the top of the stack

 if $M[s, b] = \text{shift } n$

 push b onto the stack

 push n onto the stack

 let b the next symbol in input

 else if $M[s, b] = \text{reduce } A \rightarrow \beta$

 pop $|\beta|$ symbols of the stack

 let m the state of the top of the stack

 let n such that $M[m, A] = \text{goto } n$

 push A

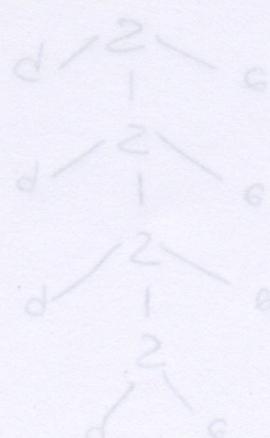
 push n

 output $A \rightarrow \beta$

 else if $M[s, b] = \text{accept}$

 break

 else error()



GENERATION PARSING TABLE - BOTTOM UP PARSING SLR(1)

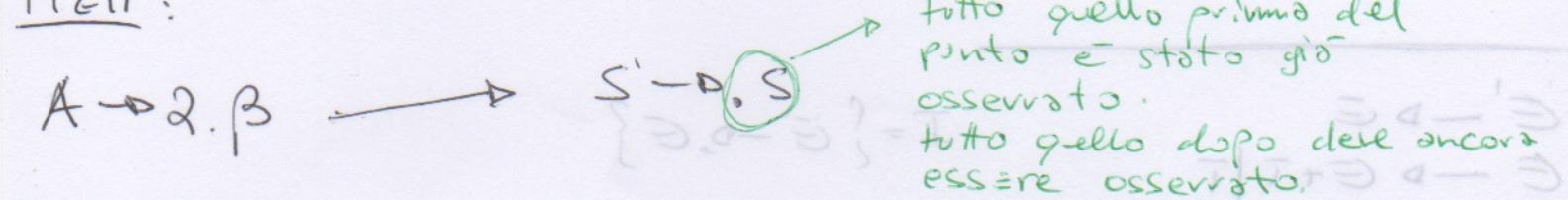
Una qualsiasi grammatica $G = (V, T, S, P)$ può avere più produzioni per il simbolo iniziale. Questo può essere un problema per la generazione della bottom up parsing table perché è la rappresentazione di un automma a stati finiti.

Per questo si considera la grammatica arricchita

$$G' = (V \cup \{S'\}, T, S', P \cup \{S' \rightarrow S\}) \quad S' \notin V$$

Dobbiamo definire una funzione di transizione e una funzione per costruire lo stato di destinazione.

ITEM:



Si definisce la funzione closure(I) dove I è l'ITEM di partenza. Questa funzione genera tutti gli ITEM da I:

function closure (I)

```

J = I
repeat
    foreach A → a.Bβ in JT. → T
        foreach B → γ in PI. → T
            if B → γ ∈ J
                add B → γ to J
until saturation
return J
  
```

$$P^I = P \cup \{S' \rightarrow S\}$$

N.B: se l'elemento dopo il punto è un terminale non ci sono closure da quella produzione.

ESEMPI Di CLOSURE

$A \rightarrow aBc \mid aC$ $I = \{A \rightarrow a.Bc, A \rightarrow a.C\}$
 $B \rightarrow d \mid Cde$
 $C \rightarrow e \mid \epsilon$

- 1 $[A \rightarrow a.Bc]$
- 2 $[A \rightarrow a.C]$
- 3 $[B \rightarrow .d]$ [generati da $B \rightarrow .d$]
 $[C \rightarrow .e]$ [generati da $C \rightarrow .e$]
 $[C \rightarrow .\epsilon]$ [generati da $C \rightarrow .\epsilon$]

$E' \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

$I = \{E' \rightarrow .E\}$

$\left. \begin{array}{l} E' \rightarrow .E \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array} \right\}$
closure (I)

(I) erende noitcnut
 $I = L$

modarator l'hnu
Invutor

now gioninuot nu 3 ctnq l, ogalo akwatu'l se: 84
 .anoifuborg ollng ab eredojs aro2 5

FUNZIONI DI TRANSITIONE DELL'AUTOMA

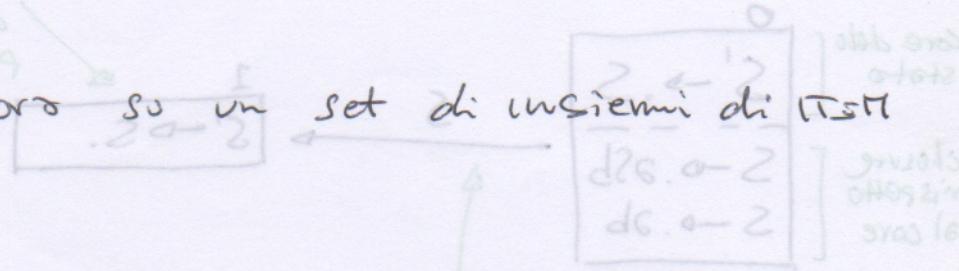
La funzione di transizione è il "goto" dello tableau.

$$\text{goto}(I, X) = \text{closure}(K)$$

K è l'insieme degli item nella forma 2.0^*2

$$A \rightarrow Q.X.\beta \quad \text{t.c. } A \rightarrow Q.X.\beta \in I$$

La funzione goto lavora su un set di insiemi di item
 $C = \{\text{closure}\{[s^* \rightarrow s]\}\}$



repeat

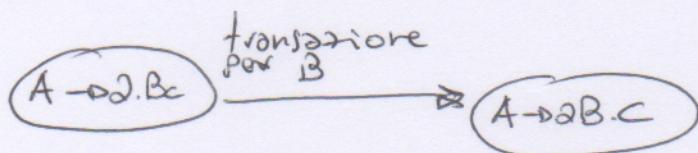
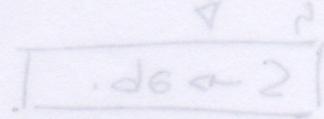
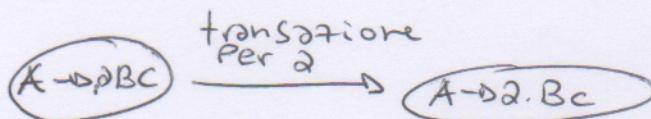
foreach set in C

foreach $x \in \Sigma$ t.c. $\text{goto}(I, x) \neq \emptyset \wedge \text{goto}(I, x) \notin C$

add $\text{goto}(I, x)$ to C

until saturation

In sostanza la funzione goto dice che con un certo simbolo in lettura posso spostare un "punto" sulla produzione:



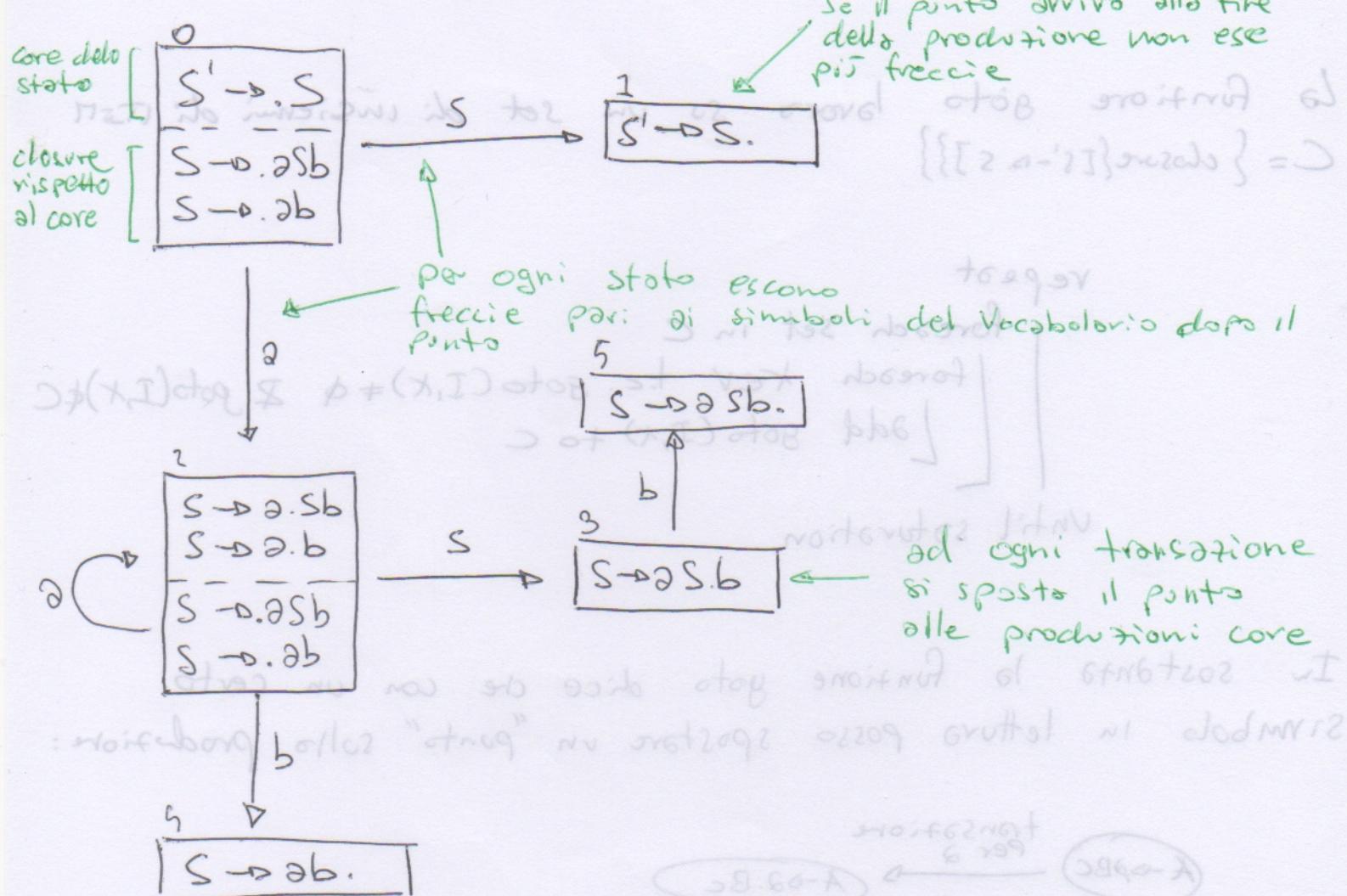
ESEMPPIO

AMMOSA' E AVERIFIQUAR TUTTI I PROBLEMI

Dato la grammatica L, arricchita G' , costruire l'automa per la generazione della parsing table bottom-up.

$S' \rightarrow S \cdot S$

$I \Rightarrow S \cdot S \cdot A$



Una volta costruito l'automa, il passaggio a tabella non è necessario, in quanto è già riscritto
dell'automa in forma tabulare.

Per riempire la tabella seguire queste regole:

1 - $S \rightarrow S$: questo tipo di stato è speciale perché ha il punto in fondo alla produzione e dallo stato aggiunto va nello stato iniziale
 \Rightarrow corrisponde ad $T[K, \$] = \text{ACCEPT}$ dove K è il numero dello stato

2 - $k : A \rightarrow \beta$: tutte le ~~le~~ gli altri stati con il punto in fondo alla produzione si mettono in $T[k, \text{FOLLOW}(A)] = \text{reduce } A \rightarrow \beta$

3 - $0 \xrightarrow{\alpha} 2$: tutte le transazioni fra stati sono degli shift in $T[0, \alpha] = \text{shift } 2$

TERMINALI: SHIFT
o ACCEPT

NON-TERM
GOTO

| | 2 | b | \$ | S |
|---|---|-----------------|-----------------|---|
| 0 | 2 | | | |
| 1 | | | Acc | |
| 2 | 2 | 5 | | |
| 3 | | 5 | | |
| 4 | | reduce S->ab | reduce S->ab | |
| 5 | | reduce S->Sb | reduce S->Sb | |

N.B:

COSTRUTTONE DELLA SLR PARSING TABLE

INPUT: grammatica G' arricchita.

OUTPUT: SLR parsing table per G' .

ALGORITMO:

1 - computare la collezione di insiemi di item e la goto function. Chiamiamo gli stati $\{I_0, I_1, \dots, I_n\}$ con $I_0 = \text{closure}(\{S' \rightarrow S\})$

2 - definiamo la riga per lo stato I_j come segue

- Se $A \rightarrow \alpha \beta \in I_j \wedge \text{goto}(I_j, \alpha) = I_k$

$$T[I_j, \alpha] = \text{shift } I_k$$

- Se $A \rightarrow \alpha \in I_j$ con $A \neq S'$

$$\forall x \in \text{FOLLOW}(A) \quad T[I_j, x] = \text{reduce. } A \rightarrow \alpha$$

- Se $S' \rightarrow S_i \in I_j$

$$T[I_j, \$] = \text{accept}$$

- se sono state generate entry multi definite
STOP, ERROR

else

Se $\text{goto}(I_j, A) = I_k$

$$T[I_j, A] = \text{goto } I_k$$

- riempire le caselle vuote con error().

| | | | |
|--|--|---|---|
| | | S | O |
| | | S | |
| | | S | |
| | | | E |

Esercizio

Дата la grammatica arricchita G' calcolare la bottom up parsing table SLR(1)

$$\begin{array}{ll} E' \rightarrow G & T \rightarrow T \# F \\ E \rightarrow E + T & F \rightarrow (E) \mid id \\ \text{---} & \end{array}$$

1. Partendo dal simbolo iniziale della grammatica, usarlo come core dello stato zero e fare lo chiusura.

| | |
|---|-------------------------|
| 0 | $E' \rightarrow .G$ |
| | $E \rightarrow .E + T$ |
| | $E \rightarrow .T$ |
| | $T \rightarrow .T \# F$ |
| | $T \rightarrow .F$ |
| | $F \rightarrow .(E)$ |
| | $F \rightarrow .id$ |

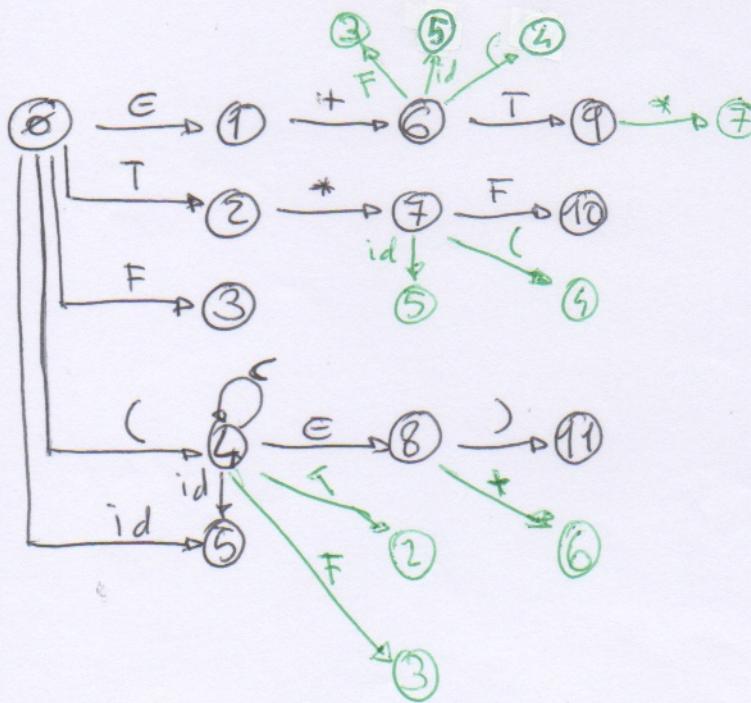
| | |
|---|------------------------|
| 1 | $E' \rightarrow E.$ |
| | $E \rightarrow E + T$ |
| 2 | $E \rightarrow T.$ |
| | $T \rightarrow T \# F$ |
| 3 | $T \rightarrow F.$ |

| | |
|---|-------------------------|
| 4 | $F \rightarrow (E)$ |
| | $E \rightarrow .E + T$ |
| | $E \rightarrow .T$ |
| | $T \rightarrow .T \# F$ |
| | $T \rightarrow .F$ |
| | $F \rightarrow .(E)$ |
| | $F \rightarrow .id$ |

| | |
|---|------------------------|
| 5 | $F \rightarrow .id$ |
| 6 | $E \rightarrow E + T$ |
| | $T \rightarrow T \# F$ |
| | $T \rightarrow .F$ |
| | $F \rightarrow .(E)$ |
| | $F \rightarrow .id$ |

| | |
|---|------------------------|
| 7 | $T \rightarrow T \# F$ |
| | $F \rightarrow .(E)$ |
| | $F \rightarrow .id$ |

| | |
|----|------------------------|
| 8 | $F \rightarrow (E.)$ |
| 9 | $E \rightarrow E + T$ |
| 10 | $T \rightarrow T \# F$ |
| 11 | $F \rightarrow (E.)$ |



$$FOLLOW(5) = \{ +, *,), \$ \}$$

$$FOLLOW(2) = \{ +,), \$ \}$$

$$FOLLOW(3) = \{ +, *,), \$ \}$$

$$FOLLOW(9) = \{ +,), \$ \}$$

$$FOLLOW(10) = \{ +, *,), \$ \}$$

$$FOLLOW(11) = \{ +, *,), \$ \}$$

dai driver

Calcolare i FOLLOW degli stati con il punto davanti fine produzione

During transition.

Come prora eseguire il run sulla parola $w = (\text{id} + \text{id}) * \text{id}$

Pila

\emptyset

$\emptyset(\cdot)$

$\emptyset(\cdot, \text{id}^*)$

$\emptyset(\cdot, F_3)$

$\emptyset(\cdot, T_2)$

$\emptyset(\cdot, E_8)$

$\emptyset(\cdot, E_8 + T)$

$\emptyset(\cdot, E_8 + T^*)$

$\emptyset(\cdot, E_8 + T^* F_3)$

$\emptyset(\cdot, E_8 + T^* F_3)$

$\emptyset(\cdot, E_8)$

$\emptyset(\cdot, E_8)T^*$

$\emptyset F_3$

$\emptyset T_2$

$\emptyset T_2 * T$

$\emptyset T_2 * T^* id^*$

$\emptyset T_2 * T^* F_{10}$

$\emptyset T_2$

$\emptyset E_1$

Input

$(\text{id} + \text{id}) * \text{id} \$$

$\text{id} + \text{id}) * \text{id} \$$

$+ \text{id}) * \text{id} \$$

Output.

$E | GTB \rightarrow T$

$F \rightarrow id$

$T \rightarrow F$

$C \rightarrow CT$

$26.0 - 2$

$d26.0 - 2$

$F \rightarrow id$

$T \rightarrow F$

$C \rightarrow ET + T$

$E | GTB \rightarrow T$

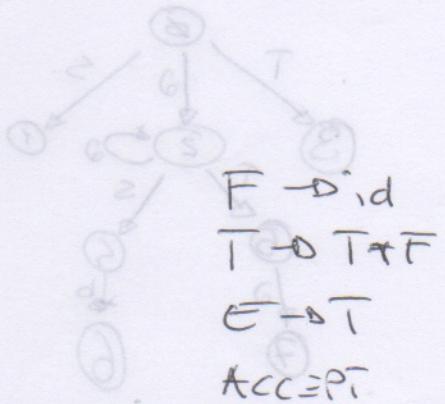
$T \rightarrow F$

$F \rightarrow id$

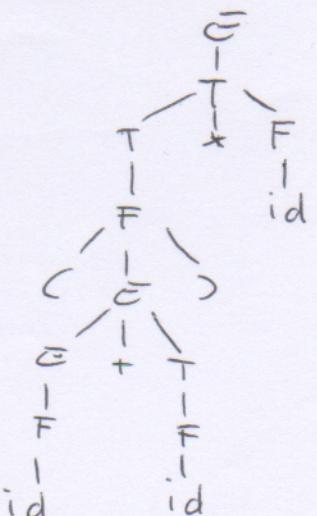
$T \rightarrow T + F$

$E \rightarrow T$

$ACC \Rightarrow T$



Ceggendole a ritroso



NB: Per controllare se l'automa è multi definito, senza scrivere la tabella, controllare che per ogni stato che si ha calcolato i FOLLOW, non ci siano archi uscenti con il simbolo presente nei FOLLOW.

ESERCIZIO Data la grammatica G , scrivere la passing table.

DATA LA GRAMMATICA G , SCRIVERE LA PARSING TABLE.

$$S \rightarrow \partial S \bigg| \partial Sb \bigg| \bar{1}$$

$$T \rightarrow \partial \bar{T} \partial / \partial$$

$$S' \rightarrow S$$

| |
|-------------------------------------|
| $S' \rightarrow S$ |
| $S \rightarrow \alpha S$ |
| $S \rightarrow \alpha Sb$ |
| $S \rightarrow \alpha T$ |
| $S \rightarrow \alpha T\alpha$ |
| $T \rightarrow \alpha \cdot \alpha$ |

1
 $S^1 \rightarrow S.$

$S \rightarrow S.S$
 $S \rightarrow S.Sb$
 $\bar{T} \rightarrow \bar{S}.T\bar{\alpha}$
 $T \rightarrow S.$
 - - - - -
 $S \rightarrow S.S$
 $S \rightarrow S.Sb$
 $S \rightarrow .\bar{T}$
 $T \rightarrow .\bar{S}\bar{T}\bar{\alpha}$
 $T \rightarrow .S$

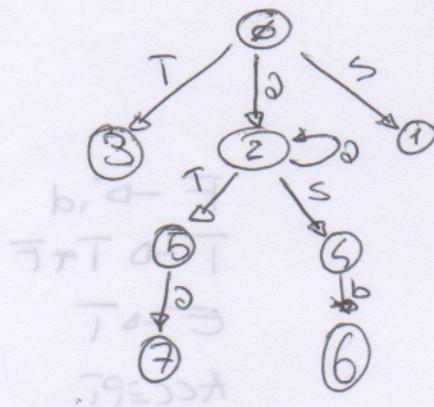
$S \rightarrow T$.

S → a S.
S → a S.b

$$T \rightarrow \partial T, \partial$$
$$S \rightarrow T.$$

6 S → eSb.

$T \rightarrow \partial T_2$



$$\text{Follow}(S) = \{ b, \$ \}$$

$$\text{Follow}(T) = \{a, b, \$\}$$

2nd 2nd

lab added \$0/

195502

T

tomorrow of old
90 minutes

| | | | | | |
|---|--|---|---------------------------------------|--------|--------|
| 0 | shift 2 | | | goto 1 | goto 3 |
| 1 | | | ACCEPT | | |
| 2 | reduce $T \rightarrow S$ shift 2 | reduce $T \rightarrow S$ | reduce $T \rightarrow S$ | goto 4 | goto 5 |
| 3 | | | | | |
| 4 | | reduce $S \rightarrow \partial S$ shift 6 | reduce $S \rightarrow \partial S$ | | |
| 5 | shift 7 | reduce $S \rightarrow T$ | reduce $S \rightarrow T$ | | |
| 6 | | reduce $S \rightarrow \partial Sb$ | reduce $S \rightarrow \partial Sb$ | | |
| 7 | reduce $T \rightarrow \partial Ta$ | reduce $T \rightarrow \partial Ta$ | reduce $T \rightarrow \partial Ta$ | | |

28d/A8d/2

22/b8d/A

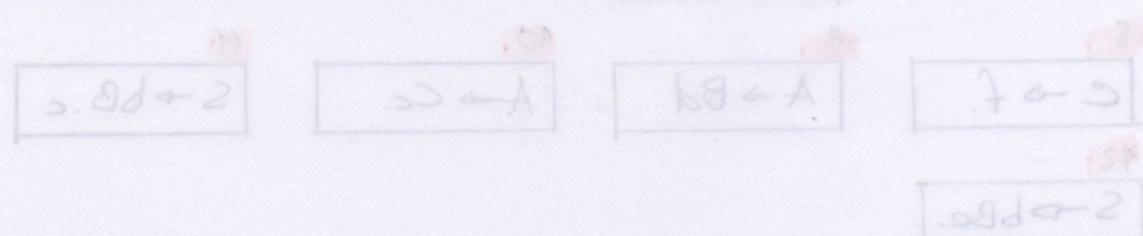
3/9/a/d

3/7/a/J

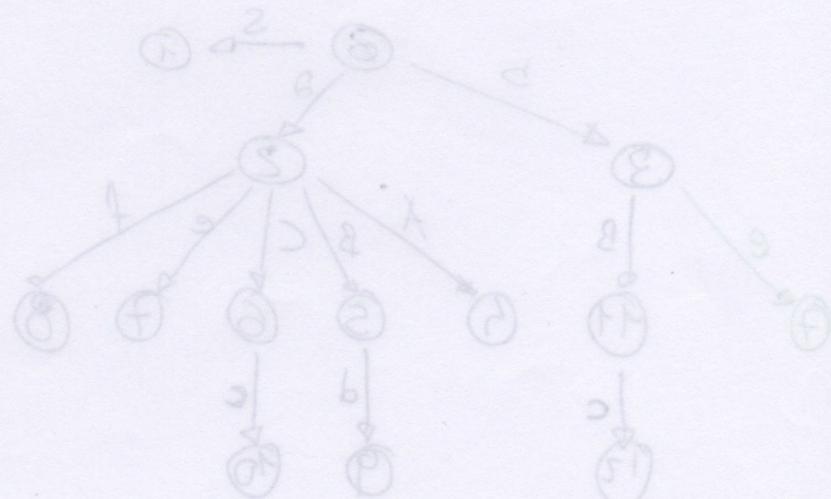
2-a-12

| |
|--------|
| 2.0-2 |
| A6.0-2 |
| 2d.0-2 |

■ Questa grammatica è ambigua in quanto presenta entry multi definite.



$\{b, \# \} = (1) \text{ word}$
 $\{b, - \} = (2) \text{ word}$
 $\{- \} = (3) \text{ word}$
 $\{\# \} = (4) \text{ word}$

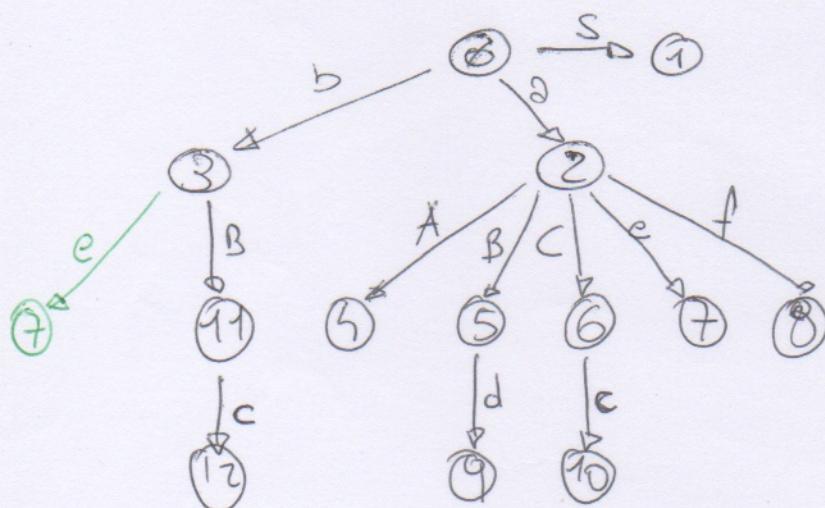


Esercizio

Data la grammatica G , scrivere la tabella dei parsing bottom up:

$$\begin{aligned} S &\rightarrow aA \mid bBc \\ A &\rightarrow Bd \mid Cc \\ B &\rightarrow e \mid E \\ C &\rightarrow f \mid E \\ S' &\rightarrow S \end{aligned}$$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---------------------|---|---------------------|----------------------|----------------------|---------------------|---|--------------------|---|---------------------|----|----------------------|----|
| $S' \rightarrow S.$ | | $S \rightarrow a.A$ | $S \rightarrow b.Bc$ | | $A \rightarrow B.d$ | | $B \rightarrow e$ | | $A \rightarrow C.c$ | | $S \rightarrow bB.c$ | |
| | | $A \rightarrow .Bd$ | | | | | | | | | | |
| | | $A \rightarrow .Cc$ | | | | | | | | | | |
| | | $B \rightarrow .e$ | | | | | | | | | | |
| | | $B \rightarrow .$ | | | | | | | | | | |
| | | $C \rightarrow .f$ | | | | | | | | | | |
| | | $C \rightarrow .$ | | | | | | | | | | |
| | | | | $S \rightarrow a.A.$ | | | $B \rightarrow e.$ | | | | | |
| | | | | | | | | | | | | |
| | | | | | $A \rightarrow B.d$ | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |



$$\begin{aligned} \text{FOLLOW}(A) &= \{ \$, d \} \\ \text{FOLLOW}(B) &= \{ c, d \} \\ \text{FOLLOW}(C) &= \{ c \} \\ \text{FOLLOW}(S) &= \{ \$ \} \end{aligned}$$

VANTAGGI CON GRAMMATICHE AMBIGUE

- Ce grammatiche ambigue generano tabelle di parsing multi-definite, ma non sempre sono peggiori, perché:
- genera una tabella di parsing più piccola
 - solitamente la grammatica è più leggibile.

Per questi motivi esistono tecniche che fanno scegliere al parser un'azione nell'entry multi-definito per raggiungere i nostri scopi.

ESEMPIO

$G \rightarrow E+E \mid E*E \mid (E) \mid id$ ← grammatica ambigua

- genera la parsing table aggiungendo $E \rightarrow E$

| 0 |
|-----------------------------|
| $E \rightarrow E$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 1 |
|-----------------------------|
| $E \rightarrow E$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 2 |
|-----------------------------|
| $E \rightarrow (E)$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 3 |
|-----------------------------|
| $E \rightarrow id$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 4 |
|-----------------------------|
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 5 |
|-----------------------------|
| $E \rightarrow E$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

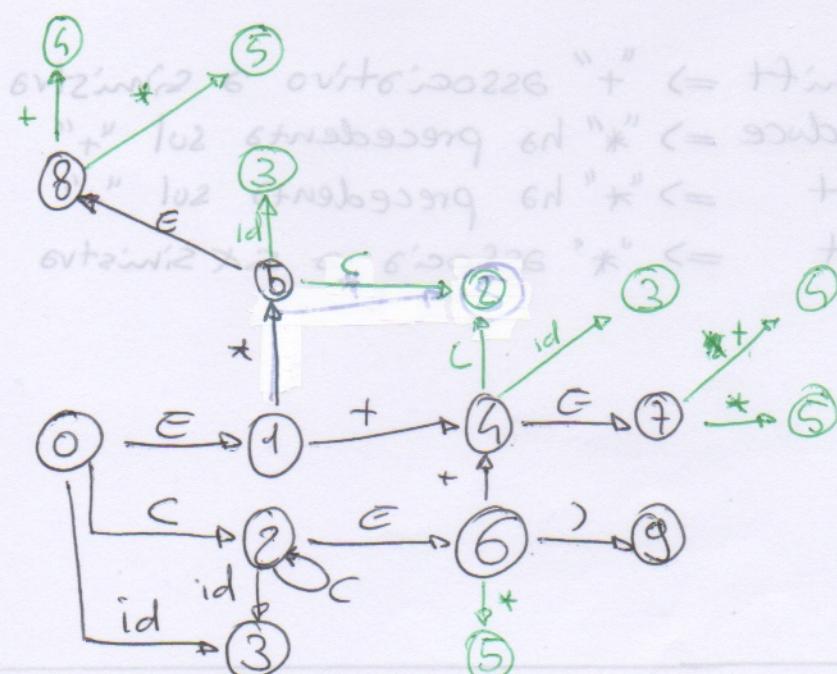
| 6 |
|-----------------------------|
| $E \rightarrow E$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 7 |
|-----------------------------|
| $E \rightarrow (E)$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 8 |
|-----------------------------|
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

| 9 |
|-----------------------------|
| $E \rightarrow (E)$ |
| $E \rightarrow E+E$ |
| $E \rightarrow E \cdot E+E$ |
| $E \rightarrow E \cdot (E)$ |
| $E \rightarrow E \cdot id$ |

$$FOLLOW(E) = \{ \$, +, *, (\},) \}$$



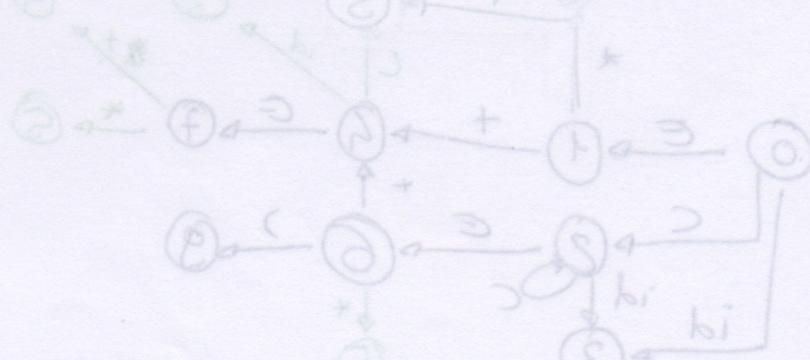
| | + | * | id | (|) | \$ | E |
|---|--------|--------|----|---|-----------------------------------|--------|--------|
| 0 | | | 3 | 2 | | | |
| 1 | 5 | 5 | | | | ACCEPT | |
| 2 | | | 3 | 2 | | | 6 |
| 3 | E->id | E->id | | | | E->id | E->id |
| 4 | | | 3 | 2 | | | 7 |
| 5 | | | 3 | 2 | | | 8 |
| 6 | | | | | bi () 9 - + - - + - - + - | | |
| 7 | E->E+E | E->E+E | | | | E->E+E | E->E+E |
| 8 | E->E+E | E->E+E | | | | E->E+E | E->E+E |
| 9 | E->(E) | E->(E) | | | | E->(E) | E->(E) |

Queste sono le entry che rendono la grammatica ambigua.
Ma c'è il modo di scegliere la mossa giusta.

Idea: Cosa ho visto quando arrivo nello stato 7? E+E
Devo riducere decidere se ridurre o leggere un altro simbolo
 - se riduco \Rightarrow "+" associa a sinistra ||
 - se shift \Rightarrow "+" associa a destra ||

Con quest'idea:

- in $[+, +]$ tolgo lo shift \Rightarrow "+" associativo a sinistra
- in $[7, *]$ tolgo il reduce \Rightarrow "*" ha precedenza sul "+"
- in $[8, +]$ tolgo shift \Rightarrow "*" ha precedenza sul "+"
- in $[8, *]$ tolgo shift \Rightarrow "*" associa a sinistra



NB: non-terminali: SHIFT , terminali: GOTO

Esercizio

$S \rightarrow \text{if } + \text{then } S \text{ else } S \mid \text{if then } S \mid \epsilon$

$S' \rightarrow S$

PROBLEMA

0

$S' \rightarrow D, S$

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S$

$S \rightarrow \cdot, \epsilon$

1

$S' \rightarrow S, \cdot$

$S \rightarrow S, \cdot$

3

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S$

5

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S$

5

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S$

$S \rightarrow \cdot, \text{if then } S$

$S \rightarrow \cdot, \epsilon$

6

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S$

7

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$S \rightarrow \cdot, \text{if then } S$

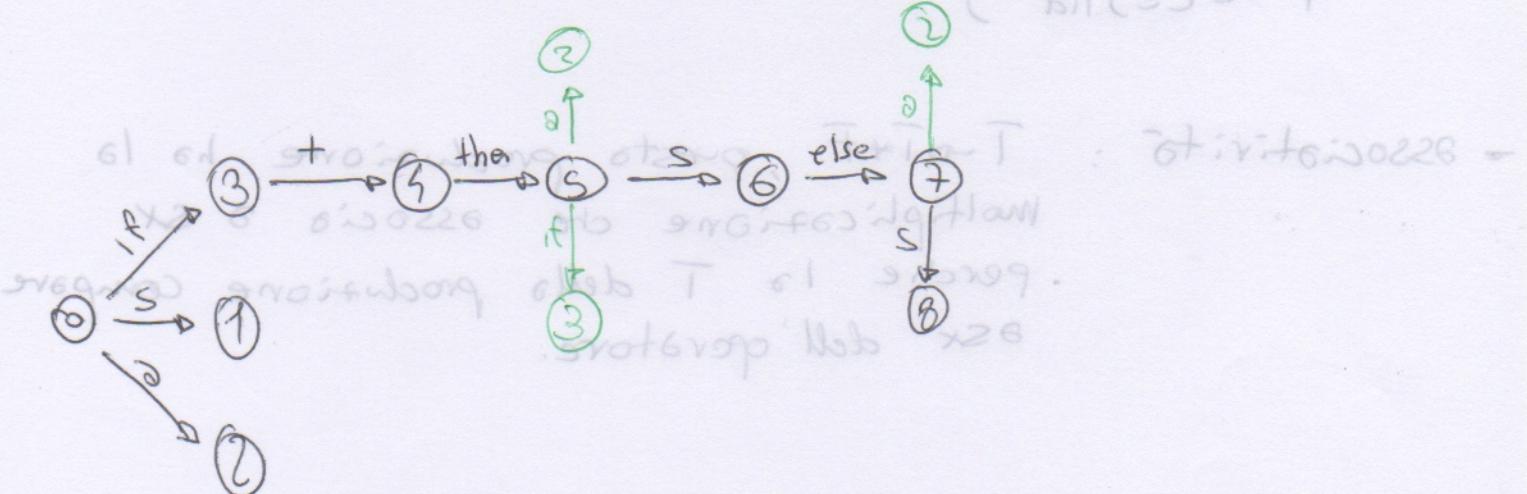
$S \rightarrow \cdot, \epsilon$

8

$S \rightarrow \cdot, \text{if then } S \text{ else } S$

$\text{Follow}(S) = \{\$, \text{else}\}$

stato di bloccaggio di inserimento blocc



| | if | + | then | else | 2 | \$ | S |
|---|----|---|---------------------------|----------------------|---|--------------------|---|
| 0 | 3 | | 6 2 . nft t, 2 s21s 2 | 2 nft + 7, a - 2 | 2 | | 1 |
| 1 | | | | | | ACCEPT | |
| 2 | | | 2 s21s 2 . nft + 7, a - 2 | | | S → a | |
| 3 | | 5 | 2 . nft + 7, a - 2 | | | | |
| 4 | | | 2 s21s 2 . nft + 7, a - 2 | | | | |
| 5 | 3 | | 6 . a - 2 | | 2 | 6 . a - 6 | |
| 6 | | | 2 s21s 2 . nft + 7, a - 2 | 7 S → if + then S | | | |
| 7 | | | 2 . nft + 7, a - 2 | | 2 | | 8 |
| 8 | | | 2 s21s 2 . nft + 7, a - 2 | S → if + then S | 2 | 2 . nft + 7, a - 2 | |

ASSOCIAZIONE E PRECEDENZA DEGLI OPERATORI

- precedenza degli operatori:

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array} \quad \left\{ \begin{array}{l} \text{la precedenza è data dalla cascata} \\ \text{delle produzioni:} \end{array} \right.$$

- associatività: $T \rightarrow T * F$, questa produzione ha la moltiplicazione che associa a sx perché la T della produzione compare a sx dell'operatore.

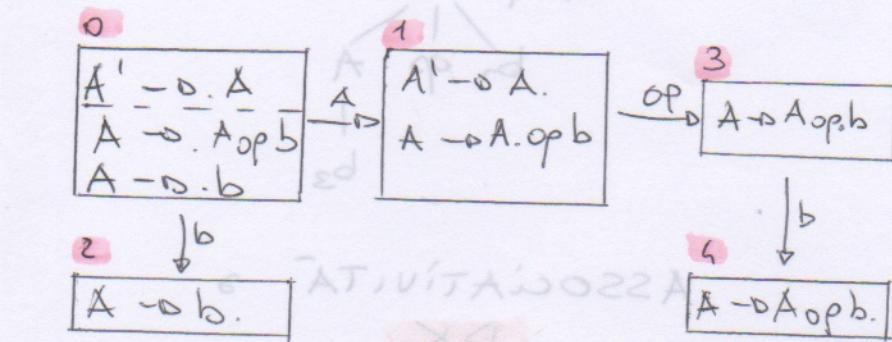
ESEMPIO PER DIMOSTRARE L'ASSOCIAIVITÀ A SX

Consideriamo due semplici grammatiche G_1, G_2 :

$$G_1: A \rightarrow A^{\text{op}} b/b$$

Gz: A → b op A lb → es analizziamo le con le strings bop bop b

$$G_1 \quad \text{Follow}(A) = \{ \$, \text{op} \}$$



| | b | op | \$ | A |
|---|---|-------|-------|---|
| 0 | 2 | | | 1 |
| 1 | | 3 | ACC | |
| 2 | | X-db | A-db | |
| 3 | 5 | | | |
| 4 | | A→Apb | A→Apb | |

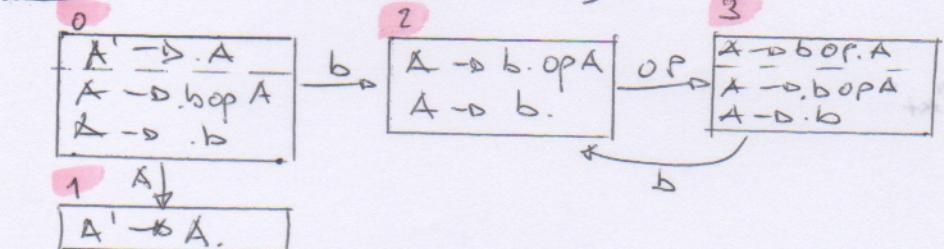
Pilo

Input

\emptyset
 $\emptyset b_2$
 $\emptyset A_1$
 $\emptyset A_1 op_3$
 $\emptyset A_1 op_3 b_4$
 $\emptyset A_1$
 $\emptyset A_1 op_3$
 $\emptyset A_1 op_3 b_4$
 $\emptyset A_1$
ACC

Output

G₂ FOLLOW(A) = { \$ }



$$A \xrightarrow{A \rightarrow b \oplus A}$$

Pilo

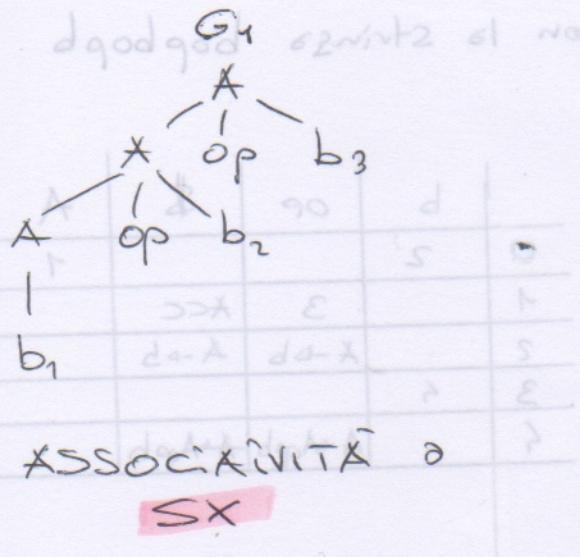
Input Output

\emptyset
 $\emptyset b_2 op_3 b_2 op_3 b_2$
 $\emptyset b_2 op_3 \underline{b_2 op_3 Ag}$
 $\emptyset \underline{b_2 op_3 Ag}$
 $\emptyset A_1$
Acc

bop bop bop A
A → b
A → bop A
A → bop A

| | b | OP | \$ | A |
|---|---|----|---------|---|
| 0 | 2 | | | 1 |
| 1 | | | XCC | |
| 2 | | 3 | X-ab | |
| 3 | 2 | | | 4 |
| 4 | | | X-abopt | |

Ricostituendo a vitoso gli alberi di elevazione delle stringe, si ottiene:



$$(b_1 \oplus b_2) \oplus b_3$$

perché la produzione
nella grammatica è
scritta:

$A \rightarrow A \oplus b \parallel b$

ESERCIZIO

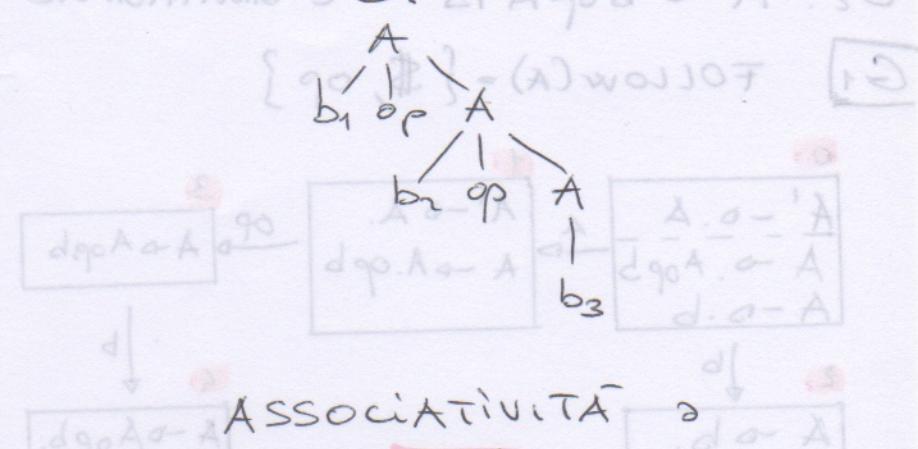
Scrivere una grammatica con gli operatori "+" e "·" (exp)

con
- "+" associa à SI
- "1" associa à DX
- "1" precedentes sul

$$E \rightarrow E + T | T$$

T- σ F \wedge T|F

$F \dashv\vdash (\equiv)$ id

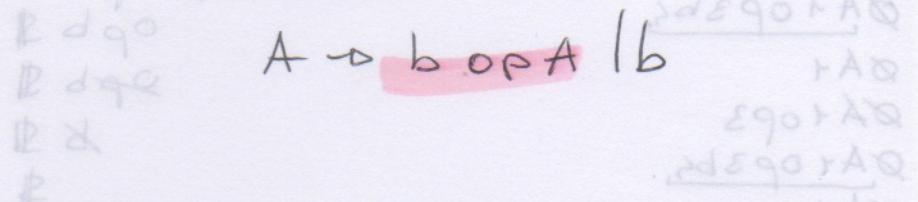


ASSOCIAТИВИТАТ

$$b_1 \circ_p (b_2 \circ_p b_3)$$

perché la produzione
nella grammatica è
scritta

$$A \rightarrow b \circ a \quad |b$$



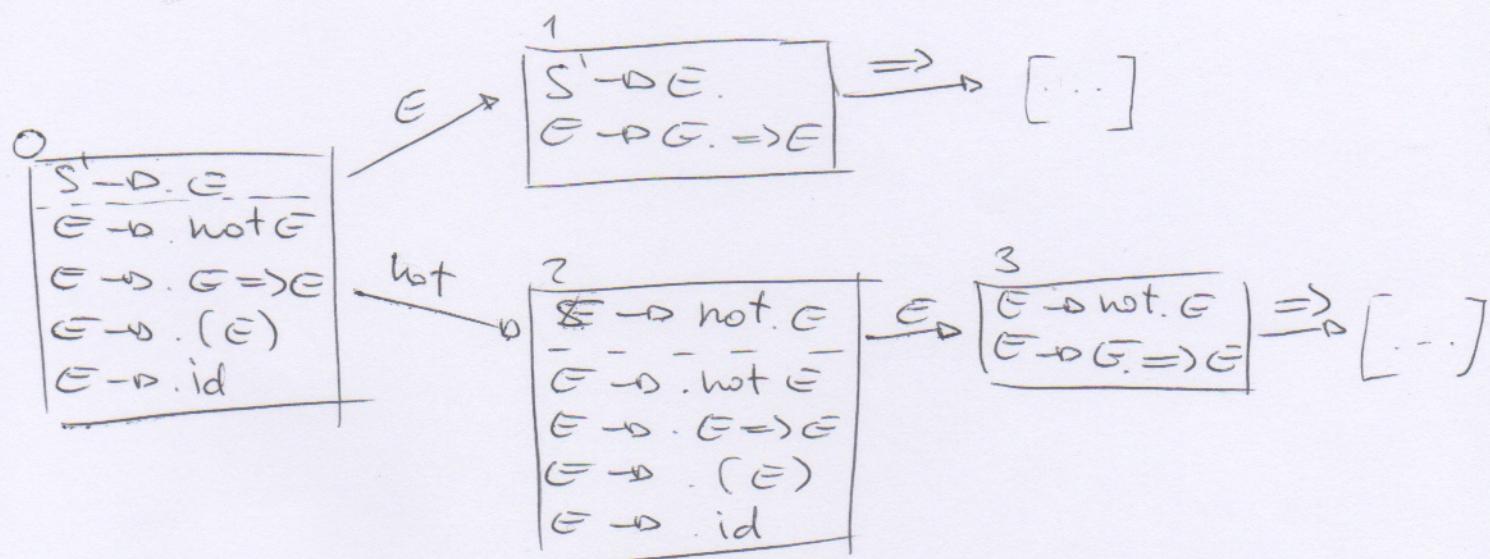
con gli operatori "+" e "·" (exp)

Esercizi

Sia G la seguente grammatica:

$$E \rightarrow \text{not } E \mid E \Rightarrow E \mid (E) \mid \text{id}$$

- Sia q_0 lo stato iniziale dell'LR(0) automata per G , dire quali stati dell'automa contengono conflitti, identificando tali stati come $\text{goto}^*(q_0, \cdot)$ dove \cdot possibile di lunghezza minima.
- per ogni conflitto trovato sopra, proporre una soluzione, supponendo che l'operatore ' \Rightarrow ' sia associativo a dx e il 'not' abbia precedenza su ' \Rightarrow '.



| FIRST | FOLLOW |
|----------------------------|------------------|
| $\text{not}, (, \text{id}$ | $\Rightarrow,)$ |

Gli stati 1 e 3 contengono un conflitto shift-reduce
 $\text{goto}^*(q_0, E \rightarrow E)$
 $\text{goto}^*(q_0, \text{not } E)$

Analisi lessicale: scopre e identifica i token del programma in input.

Analisi sintattica: dona output dell'analisi lessicale cerca di comprendere dalla grammatica l'input.
Se ci riesce genera un albero di derivazione

Analisi semantica: controlli sulla semanticità
controlli statici sul programma che non dipendono dall'esecuzione del programma.

position := initial + rate * 60

↓ analizzatore

lessicale

<id, p1> ASSIGN <id, p2> PLUS <id, p3> TIMES <num, p4>

↓ analizzatore
sintattico/semantico

↓ ASSIGN

id₁

PLUS

id₂

TIMES

id₃

n60

- grammatiche

- automati

stati finiti

- reg exp

↓ Thompson

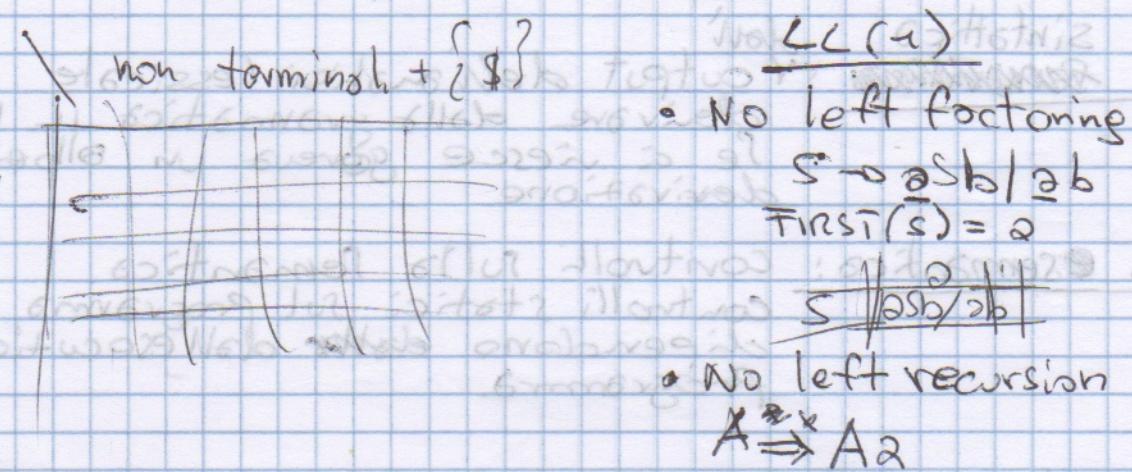
NFA

↓ subset construction

DFA

pila

TOP DOWN: ricostruzione della stringa, leggendo da sinistra a destra, usando le produzioni della grammatica.



BOTTOM UP: voglio ricostruire la stringa, partendo dalle produzioni che compaionebbero per ultime (bottom) del derivation tree. Si usano automi e pila.

- SLR(1) reduce & Follow dei Driver delle produzioni con punto in fondo
- CLR(1) reduce & lookahead delle produzioni con punto in fondo
- LALR(1)

E → E + E | (E) | id

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

$$\begin{array}{l} E \rightarrow T E' \\ E' \rightarrow D + T E' \mid E \\ T \rightarrow F T' \\ T' \rightarrow D * F T' \mid E \\ F \rightarrow (E) \mid \text{id} \end{array}$$

ambiguous left recursion/factoring

SLR, associatività e priorità
embeddable.

left recursion

LL(1) No left recursion

INTRODUZIONE AL PARSER LR(1)

NETI (1)R

Consideriamo la seguente grammatica:

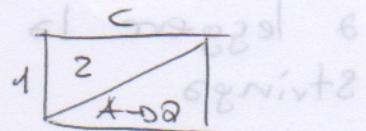
$$\begin{aligned} S &\rightarrow AB \mid \alpha c \mid xAc \\ A &\rightarrow \alpha \\ B &\rightarrow b \mid \epsilon \end{aligned}$$

Proviamo a costruire la SLR parsing table.

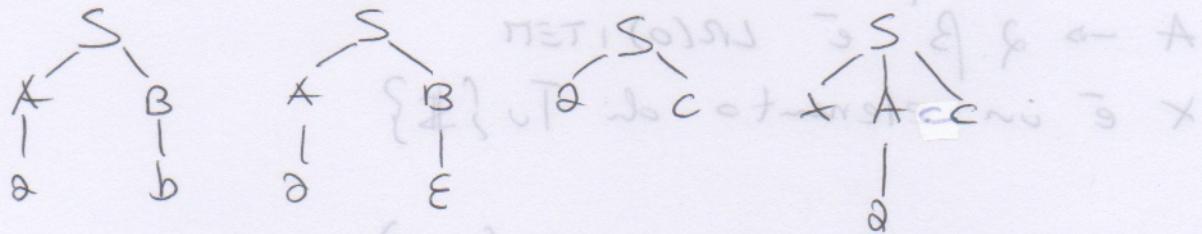
| 0 | 1 |
|--------------------------|--------------------------|
| $S' \rightarrow S$ | $S \rightarrow \alpha c$ |
| $S \rightarrow A B$ | $S \rightarrow \alpha$ |
| $S \rightarrow \alpha c$ | $S \rightarrow c$ |
| $S \rightarrow x A c$ | $S \rightarrow x A c$ |
| $A \rightarrow \alpha$ | |

Siccome "c" è FOLLOW(A)
 \Rightarrow GRAMMATICA NON SLR

Perché:



Se è ambigua dovrà trovare due alberi di derivazione per la stessa stringa ma distinti.



Come si può notare non risulta ambigua.

Questo accade perché in SLR mettiamo i passi di riduzione in tutti i FOLLOW; nel nostro caso di $A = \{b, c, \$\}$.

In SLR non si tiene conto di quale produzione derivano i FOLLOW di A , aggiungiamo nella tabella le riduzioni per tutti allo stesso modo.

Produzioni diverse descrivono il punto a cui siamo arrivati a leggere la stringa in input:

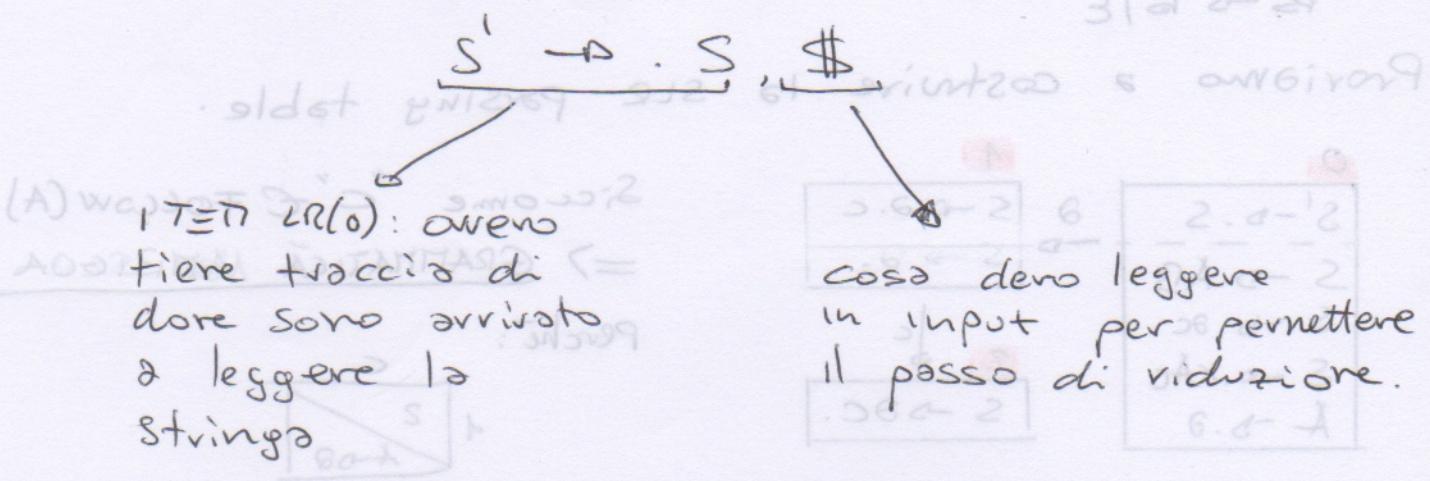
$$\begin{aligned} S &\Rightarrow AB \Rightarrow \textcircled{1} \Rightarrow \textcircled{2} \\ S &\Rightarrow AB \Rightarrow \textcircled{1} b \Rightarrow \textcircled{2} b \\ S &\Rightarrow \alpha c \\ S &\Rightarrow xAc \Rightarrow \textcircled{3} \end{aligned}$$

stessi FOLLOW
ma situazioni
diverse

Il problema sono i FOLLOW in SLR!

LR(0) IT \equiv I

Per contestualizzare i follow si deve cambiare $IT \equiv I$, usandone di più arricchiti: LR(1) ITEM



P:J formalmente:

$A \rightarrow 2. \beta, x$ dove

$A \rightarrow 2. \beta$ è LR(0) ITEM

x è un elemento di $T \cup \{ \$ \}$

Closure (I):

```

repeat
  foreach (A → 2. Bβ, x) ∈ I
    foreach (B → j in P)
      foreach (y ∈ FIRST(βx))
        [Add [B → j, y] to I]

```

until saturation
return I

goto (I, X)

$J = \emptyset$

```

foreach (A → 2. Xβ, x) ∈ I
  [Add [A → 2X. β, x] to J]
return closure(J)

```

Come si nota vengono modificate le funzioni goto e closure per tenere traccia anche delle informazioni aggiuntive

Si calcola first della parte di produzione successiva al primo simbolo dopo il punto

ESEMPPIO CANONICAL LR(1) PARSER CON LR(0) ITET

$$\left. \begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow * R \mid id \\ R \rightarrow L \end{array} \right\}$$

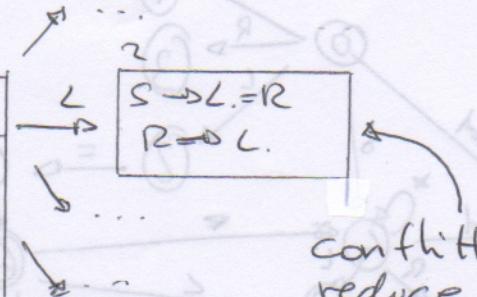
Grammatica per la gestione degli assegnamenti con i puntatori.

Esempio con SLR \subseteq LR(1)

SLR

| FOLLOW | |
|--------|------|
| S | \$ |
| L | \$ = |
| R | \$ = |

| |
|------------------------|
| $S' \rightarrow .S$ |
| $S \rightarrow .L = R$ |
| $S \rightarrow .R$ |
| $L \rightarrow . * R$ |
| $L \rightarrow .id$ |
| $R \rightarrow .L$ |



confitto shift/reduce!
Perché "=" \in FOLLOW(s)

LR(0)

| |
|----------------------------|
| $S' \rightarrow .S, \$$ |
| $S \rightarrow .L = S, \$$ |
| $L \rightarrow . * R, =$ |
| $L \rightarrow . * R, \$$ |
| $L \rightarrow .id, \$$ |
| $R \rightarrow .L, \$$ |

| |
|---------------------------|
| $L \rightarrow . * R, =$ |
| $L \rightarrow . * R, \$$ |
| $R \rightarrow .L, =$ |
| $R \rightarrow .L, \$$ |
| $L \rightarrow . * R, =$ |

| | |
|---|--|
| 1 | $S' \rightarrow .S, \$$ |
| 2 | $S \rightarrow .L = R, \$$ $R \rightarrow .L, \$$ |

| | |
|---|--|
| 6 | $S \rightarrow .L = R, \$$ $R \rightarrow .L, \$$ |
| 7 | $L \rightarrow . * R, \$$ $L \rightarrow .id, \$$ |
| 8 | $R \rightarrow .L, =$ $R \rightarrow .L, \$$ |

| | |
|---------|----------------------------|
| 10 ≠ 8! | $R \rightarrow .L, \$$ |
| 9 | $S \rightarrow .L = R, \$$ |

| | |
|---------|---------------------------|
| 12 ≠ 5! | $L \rightarrow .id, \$$ |
| 13 ≠ 7! | $L \rightarrow . * R, \$$ |

| | |
|---------|---------------------------|
| 11 ≠ 4! | $L \rightarrow . * R, \$$ |
| 10 | $R \rightarrow .L, \$$ |
| 12 | $L \rightarrow .id, \$$ |
| * | |

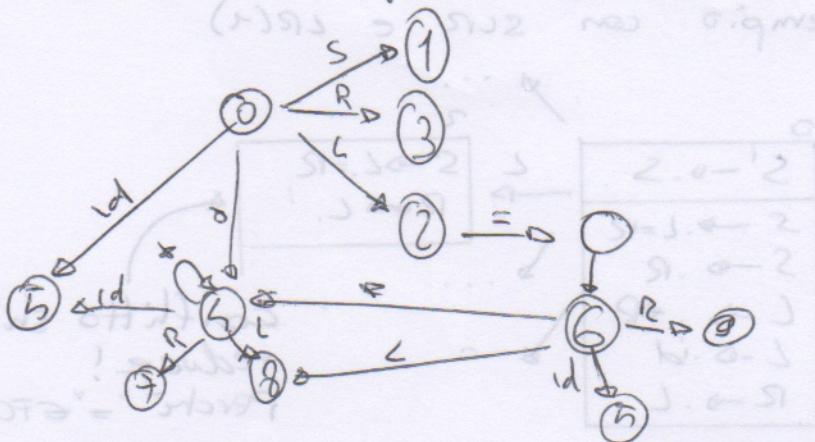
| |
|---------------------------|
| $L \rightarrow . * R, =$ |
| $L \rightarrow . * R, \$$ |
| $R \rightarrow .L, =$ |
| $R \rightarrow .L, \$$ |
| $L \rightarrow . * R, =$ |

| | |
|---|---|
| 7 | $R \rightarrow .L, =$ $R \rightarrow .L, \$$ |
| 8 | $R \rightarrow .L, =$ $R \rightarrow .L, \$$ |

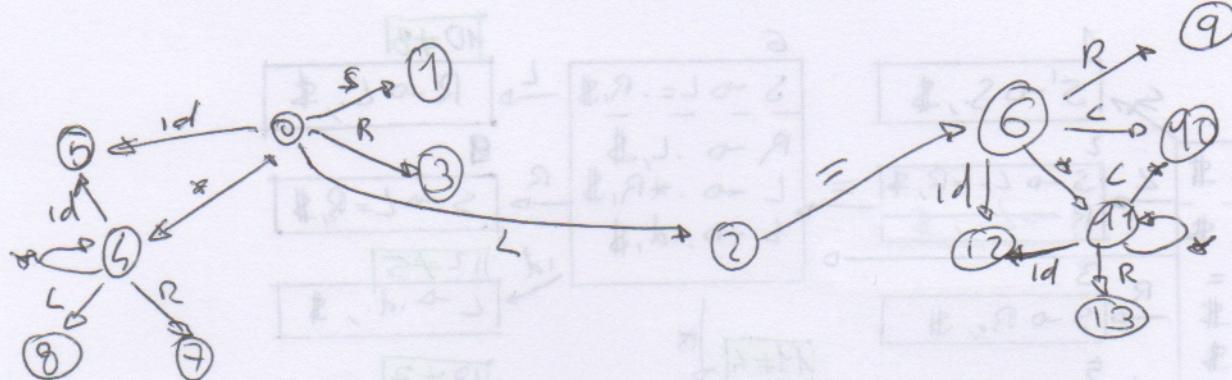
N.B.: è importante notare che alcuni ITET differiscono solo per la parte di informazione della reduce, tenendo la parte LR(0) in comune. Il compilatore LACR omisce questi ITET per ridurre la dimensione della parsing table. Successivamente viene mostrata la tabella LR(1) e poi le correzioni LACR.

Se si esplicitano gli automi SLR e LR(1) si nota come SLR metta insieme produzioni con lo stesso FOLLOW, mentre LR(1) distingua situazioni dirette a seconda delle produzioni scelte.

SLR



LR(1)



| | = | id | * | l | r | . | s | \$ |
|----|--------|--------|--------|----|----|---|---|-----------|
| 0 | | 55-12 | 54-11 | 2 | 3 | | 1 | |
| 1 | | | | | | | | ACCEPT |
| 2 | 6 | | | | | | | R → L |
| 3 | | | | | | | | S → R |
| 4 | | | | | | | | |
| 5 | L → id | | | | | | | L → id |
| 6 | | 125-12 | 124-11 | 10 | 9 | | | |
| 7 | L → *R | | | | | | | L → *R |
| 8 | R → L | | | | | | | R → L |
| 9 | | | | | | | | S → L = R |
| 10 | | | | | | | | R → L |
| 11 | | 12 | + | 40 | 13 | | | |
| 12 | | | | | | | | L → id |
| 13 | | | | | | | | L → *R |

- tabella LR(1)
- tabella ZALR: unisce le righe con core-SLR simili, si vedono dagli archi uscenti e entranti dell'automa LR(1)

LALR PARSING TABLE

C19T523

Algoritmo non efficiente per costruire la LALR(1) parsing table:

INPUT: grammatica avvicchita G'

OUTPUT: LALR(1) parsing table per G

ALGORITMO:

1 - costruire l'automa basato su LR(1) ITENS, chiamando $\{I_0, \dots, I_n\}$ i suoi stati.

2 - trovare $\{I_0, \dots, I_n\}$ gli stati con lo stesso SLR-core e li sostituiamo con la loro unione J_1, \dots, J_n

3 - costruire la parsing table relativa alle colonne $T \cup \{\$\}$ controllando il contenuto degli stati J_1, \dots, J_n .
Se ci sono conflitti \Rightarrow STOP, grammatica NO LALR

else 4 - costruire la restante parte della tabella VIT in questo modo.

$$\text{goto}(J_j, A) = J_\ell \quad \text{se } \text{goto}(I_0, A) = I_m \quad 8 \\ I_m \in I_\ell$$

ESEMPIO

PARSING TABLE

Data la grammatica arricchita G' costruiamo un petto della LR(1) parsing table per poi ricongordarlo ad una LALR(1) parsing table.

$$G: S \rightarrow aAd \mid bBd \mid aBe \mid bAe$$

$$A \rightarrow c$$

$$B \rightarrow c$$

0

| |
|-------------------------|
| $S' \rightarrow S, \$$ |
| $S \rightarrow aAd, \$$ |
| $S \rightarrow bBd, \$$ |
| $S \rightarrow aBe, \$$ |
| $S \rightarrow bAe, \$$ |

2

| |
|-------------------------|
| $S \rightarrow aAd, \$$ |
| $S \rightarrow aBe, \$$ |
| $A \rightarrow c, d$ |
| $B \rightarrow c, e$ |

4

| |
|----------------------|
| $A \rightarrow c, d$ |
| $B \rightarrow c, e$ |

3

| |
|-------------------------|
| $S \rightarrow bBd, \$$ |
| $S \rightarrow bAe, \$$ |
| $B \rightarrow c, d$ |
| $A \rightarrow c, e$ |

5

| |
|----------------------|
| $B \rightarrow c, d$ |
| $A \rightarrow c, e$ |

Usando l'algoritmo appena visto riceverebbe da unire gli stati 4 e 5 perché hanno lo stesso SLR-ONE:

| | d | c |
|---|-------------------|-------------------|
| 4 | $A \rightarrow c$ | $B \rightarrow c$ |
| 5 | $B \rightarrow c$ | $A \rightarrow c$ |

Il in questo caso l'unione di questi due stati porta ad un conflitto reduce-reduce

| | d | e |
|-----|-------------------|-------------------|
| 4-5 | $A \rightarrow c$ | |
| | $B \rightarrow c$ | $A \rightarrow c$ |

$\Rightarrow G \text{ è LR}(1) \text{ ma NON LALR}(1)$

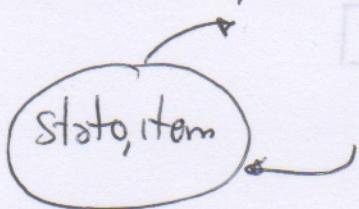
LALR : ALGORITMO EFFICIENTE

[YACC]

INPUT: Grammatica arricchita G'
OUTPUT: LALR(1) parsing table per G

ALGORITMO:

- 1 - costruzione automa LR(0)
- 2 - costruzione di un nuovo grafo della propagazione/generazione di lookahead, essendo che la tabella LALR(1) ha le stesse dimensioni della SCR, non si hanno le stesse informazioni sui lookahead nell'automa, in quanto è LR(0).



Quindi si costruisce un nuovo grafo dove ogni nodo è la coppia stato-item con lookahead.

Kernel-items dell'LR(0) automa: $S \rightarrow S, A \rightarrow ., A \rightarrow 2, B \rightarrow \epsilon$
compari alla stessa sibilla.

Propagazione/generazione di lookahead da Kernel items in K , dove K è l'insieme di tutti i kernel items in tutti gli stati LR(0) automa.

foreach ($A \rightarrow 2, \beta$) $\in J \in K$

$J := \text{closure}([A \rightarrow 2, B, \#])$

if ($B \rightarrow \gamma, X, \delta$) $\in J$ & $\# \neq \#$

diciamo che " δ " è generato per ($\text{goto}(I, X), B \rightarrow \gamma, X, \delta$)

if ($B \rightarrow \gamma, X, \delta$) $\in J$

diciamo che lookahead propagato da $(I, A \rightarrow 2, \beta)$ o $(\text{goto}(I, X), B \rightarrow \gamma, X, \delta)$

if ($B \rightarrow ., \delta$) $\in J$ & $\# \neq \#$

diciamo che " δ " è generato per $(I, B \rightarrow .)$

if ($B \rightarrow ., \#$) $\in J$

diciamo che lookahead propagato da $(I, A \rightarrow 2, \beta)$ o $(I, B \rightarrow .)$

è un segnale posto, in modo da controllare se la closure modifica o no il lookahead.

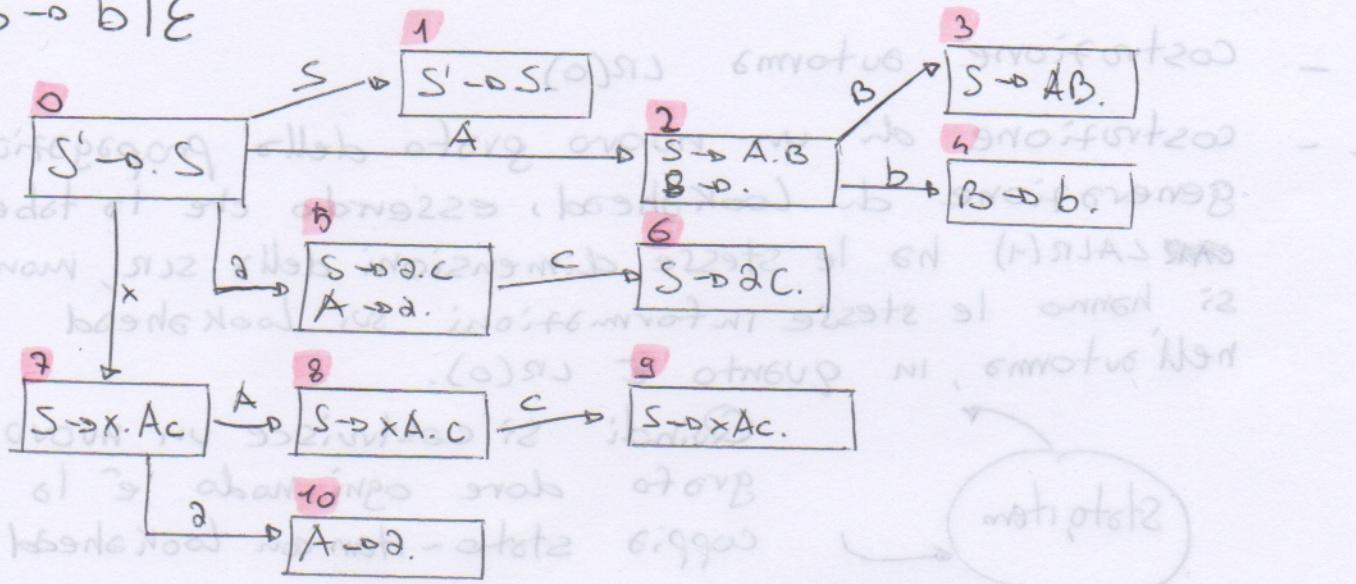
ESEMPPIO

Nell'automa ^{LR(0)} seguente vengono vi partiti in sottosezioni Kernel items.

$S \rightarrow AB/xAc$

$A \rightarrow a$

$B \rightarrow b/\epsilon$



1) Per ogni produzione di tutti i Kernel item, devo calcolare la closure LR(1) usando come simbolo il placeholder "#".

- per gli stati 1-3-4-6-9-10 non calcolo nulla in quanto il punto è in fondo alla produzione.

- closure LR(1) di I_0 :

$$\begin{cases} S' \rightarrow S \# \\ S \rightarrow A.B \# \\ S \rightarrow A.C \# \\ S \rightarrow x.Ac \# \\ A \rightarrow a.b \\ A \rightarrow a.\# \end{cases}$$

L'item \circledast propaga il simbolo # all'item corrispondente al goto della lettera che ha dietro al punto.

Quando si verifica una generazione \circledast viene salvato il lookahead generato nello stato corrispondente al goto della lettera che trova dietro al punto.

- closure LR(1) di I_2 :

$$\begin{cases} S \rightarrow A.B \# \\ B \rightarrow b \# \\ B \rightarrow . \# \end{cases}$$

- : propagation
- △ : generation.

- closure LR(1) di I_5 :

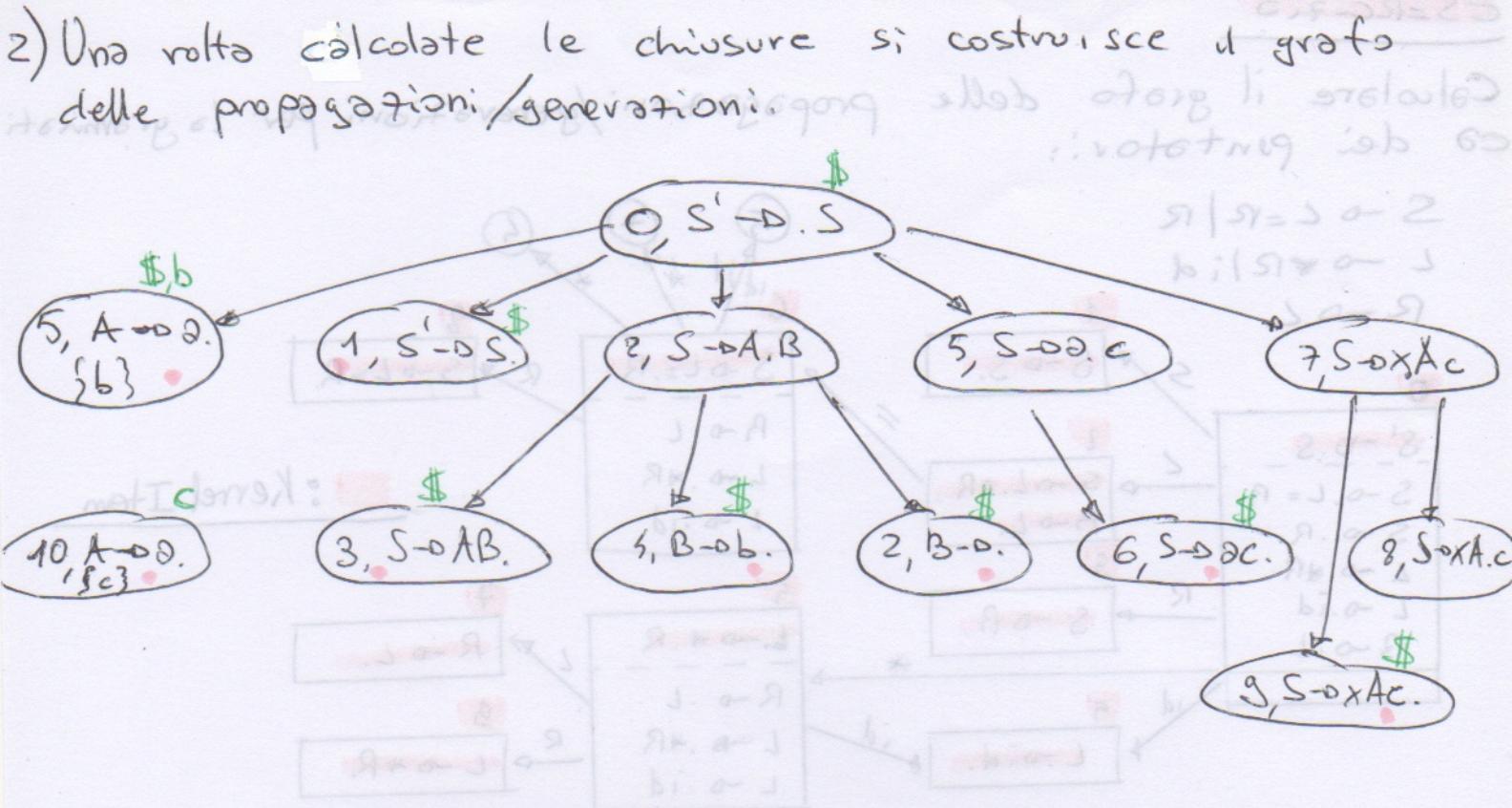
$$\begin{cases} S \rightarrow A.C \# \\ A \rightarrow a.C \triangle \end{cases}$$

- closure LR(1) di I_7

$$\begin{cases} S \rightarrow x.Ac \# \\ A \rightarrow a.C \triangle \end{cases}$$

- closure LR(1) di I_8 :

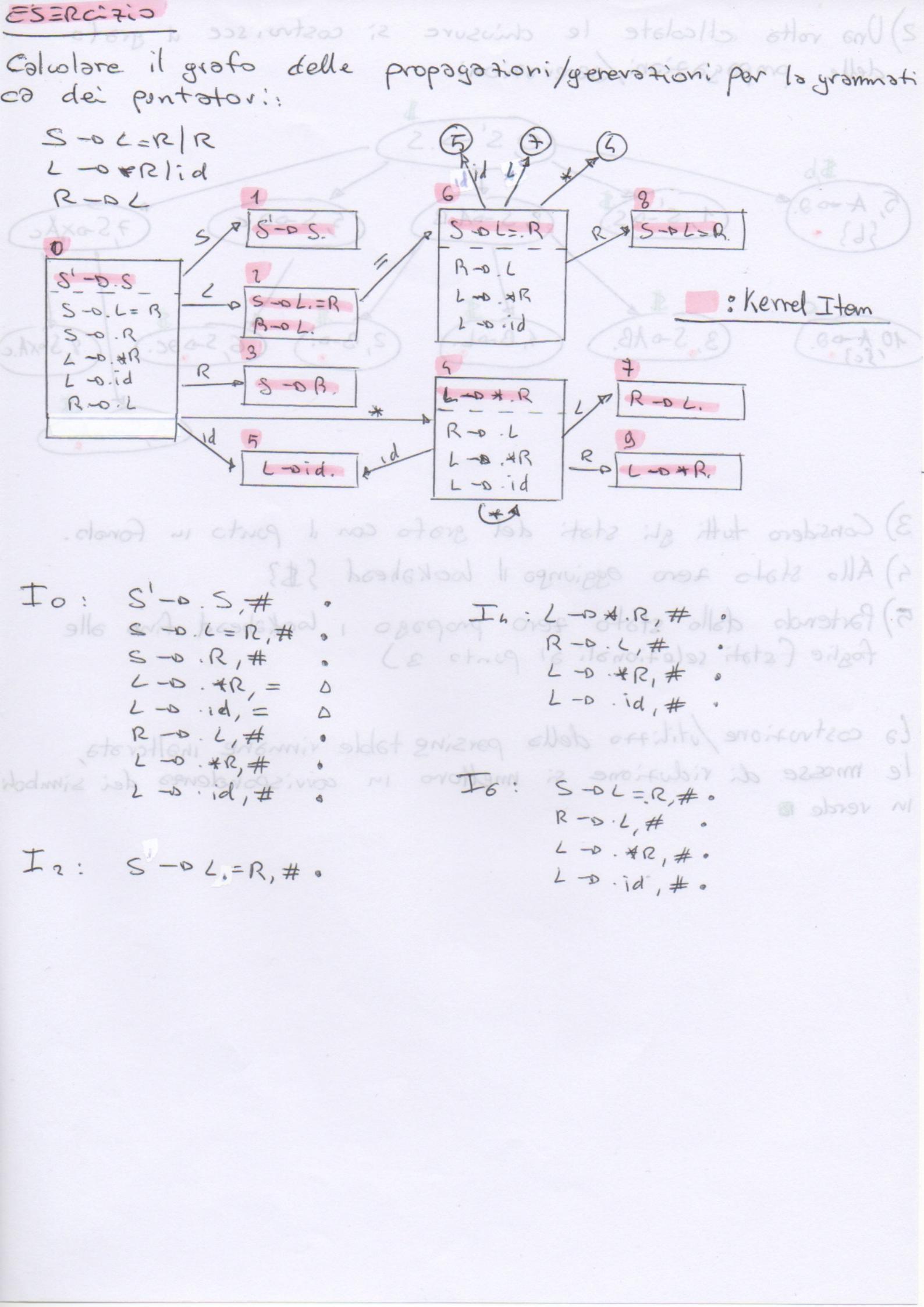
$$\begin{cases} S \rightarrow x.A.c \# \end{cases}$$

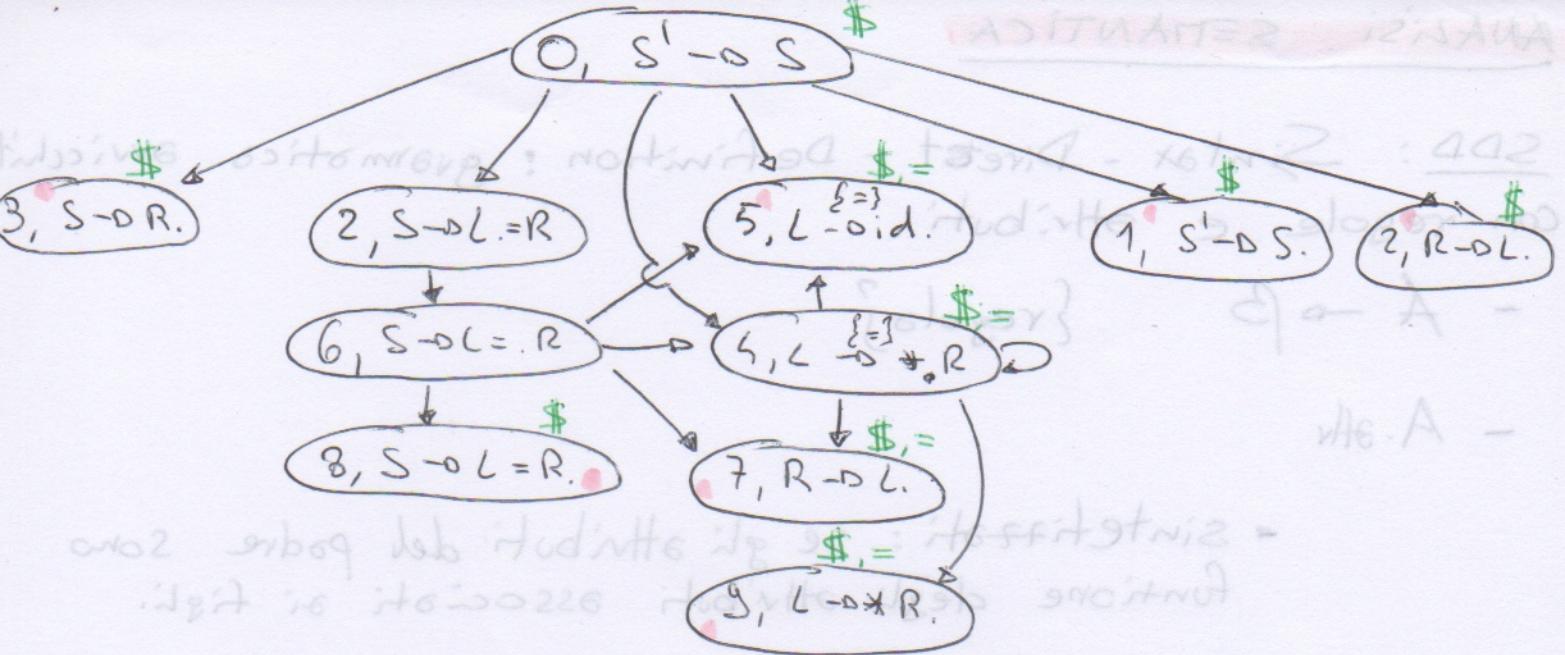


- 3) Considero tutti gli stati del grafo con il punto in fondo.
 4) Allo stato zero aggiungo il lookahead $\{\$\}$
 5) Partendo dallo stato zero propago i lookahead fino alle foglie (stati selezionali al punto 3)

La costruzione/utilizzo della parsing table rimane inalterata, le mosse di riduzione si mettono in corrispondenza dei simboli in verde.

ESERCIZIO



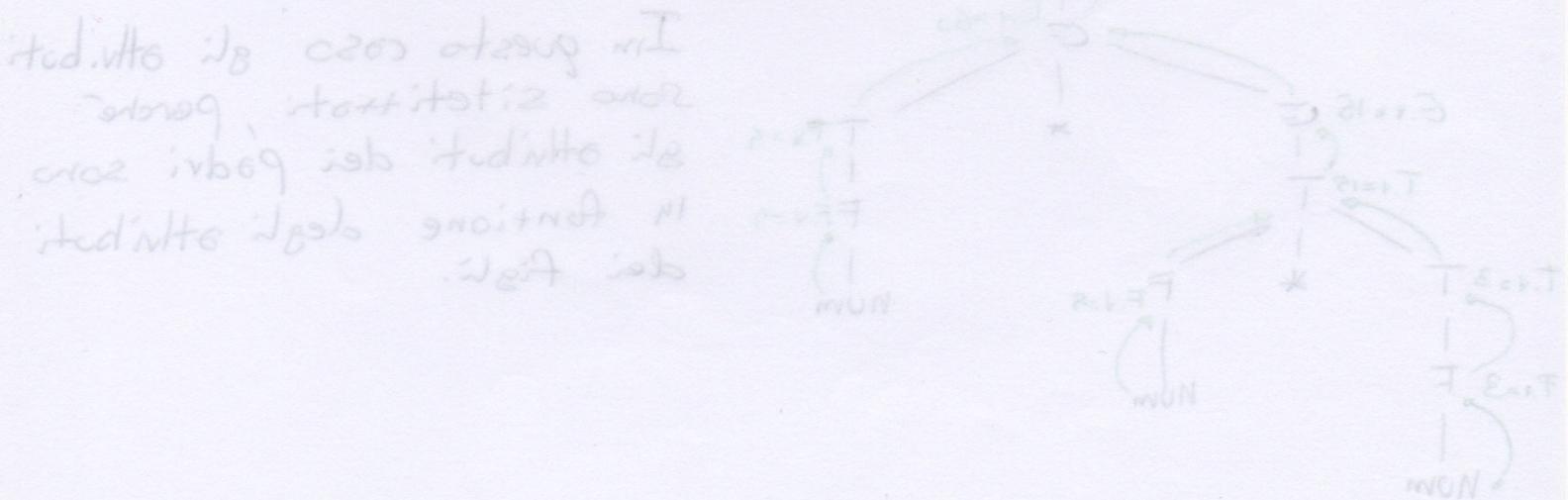


- produzioni col punto in fondo.

Nell'esempio dei spontatori si può notare che non si tratta di un albero ma di un grafo, il quale può avere dei cicli.

$$\begin{array}{lll}
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \end{array} \right\} & T \vdash v_1 = v_3 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \end{array} \right\} & T, T \vdash v_4 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \\ v_4 = v_5 \end{array} \right\} & T \vdash v_5 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \\ v_4 = v_5 \\ v_5 = v_6 \end{array} \right\} & T \vdash v_6 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \\ v_4 = v_5 \\ v_5 = v_6 \\ v_6 = v_7 \end{array} \right\} & T \vdash v_7 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \\ v_4 = v_5 \\ v_5 = v_6 \\ v_6 = v_7 \\ v_7 = v_8 \end{array} \right\} & T \vdash v_8 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \\ v_4 = v_5 \\ v_5 = v_6 \\ v_6 = v_7 \\ v_7 = v_8 \\ v_8 = v_9 \end{array} \right\} & T \vdash v_9 & \\
 \left\{ \begin{array}{l} v_1 = v_2 \\ v_2 = v_3 \\ v_3 = v_4 \\ v_4 = v_5 \\ v_5 = v_6 \\ v_6 = v_7 \\ v_7 = v_8 \\ v_8 = v_9 \\ v_9 = v_1 \end{array} \right\} & T \vdash v_1 & \\
 \end{array}$$

$(v_1, v_2) * (v_2, v_3) * (v_3, v_4) * (v_4, v_5) * (v_5, v_6) * (v_6, v_7) * (v_7, v_8) * (v_8, v_9)$



ANALISI SEMANTICA

SDD: Sintax - Direct - Definition: grammatica avvicinata con regole e attributi

- $A \rightarrow \beta$ {regola?}
- $A \cdot \text{attr}$

- sintetizzati: se gli attributi del padre sono funzione degli attributi associati ai figli.
- ereditati: se sono ereditati dal padre.

Ci può essere circolarità negli attributi:

$A \rightarrow \beta$ { $(A.a = B.b)$, $(B.b = A.a + c)$ } ||

sintetizzato ereditato

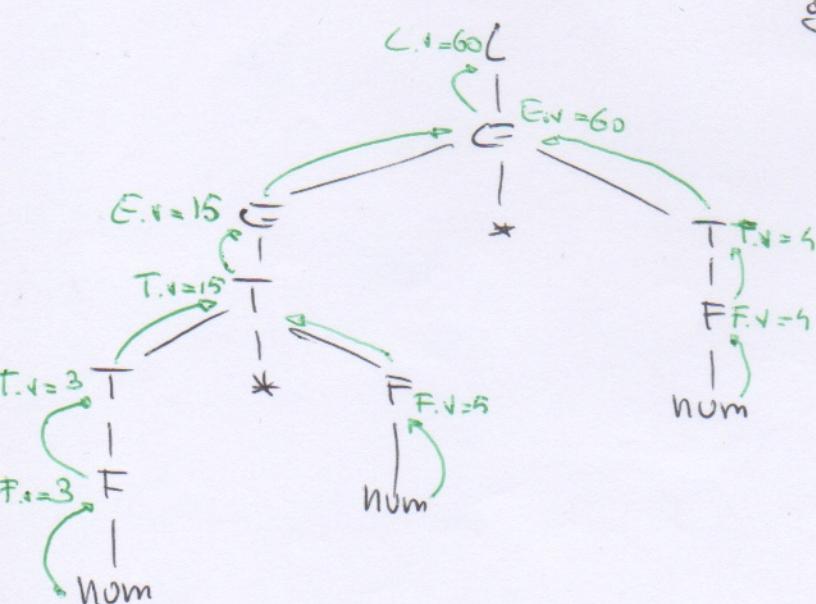
ESEMPIO: Si descrivano le regole della seguente grammatica:

| | |
|----------------------------|----------------------------|
| $C \rightarrow CE$ | { $C.v = E.v$ } |
| $E \rightarrow E_1 + T$ | { $E.v = E_1.v + T.v$ } |
| $E \rightarrow T$ | { $E.v = T.v$ } |
| $T \rightarrow T_1 * F$ | { $T.v = T_1.v * F.v$ } |
| $T \rightarrow F$ | { $T.v = F.v$ } |
| $F \rightarrow (E)$ | { $F.v = E.v$ } |
| $F \rightarrow \text{num}$ | { $F.v = \text{num.lex}$ } |

N.B.: si usano i indici per definire quali argomenti della produzione si parla.

Con queste regole proviamo ad analizzare la stringa "3*5*5".

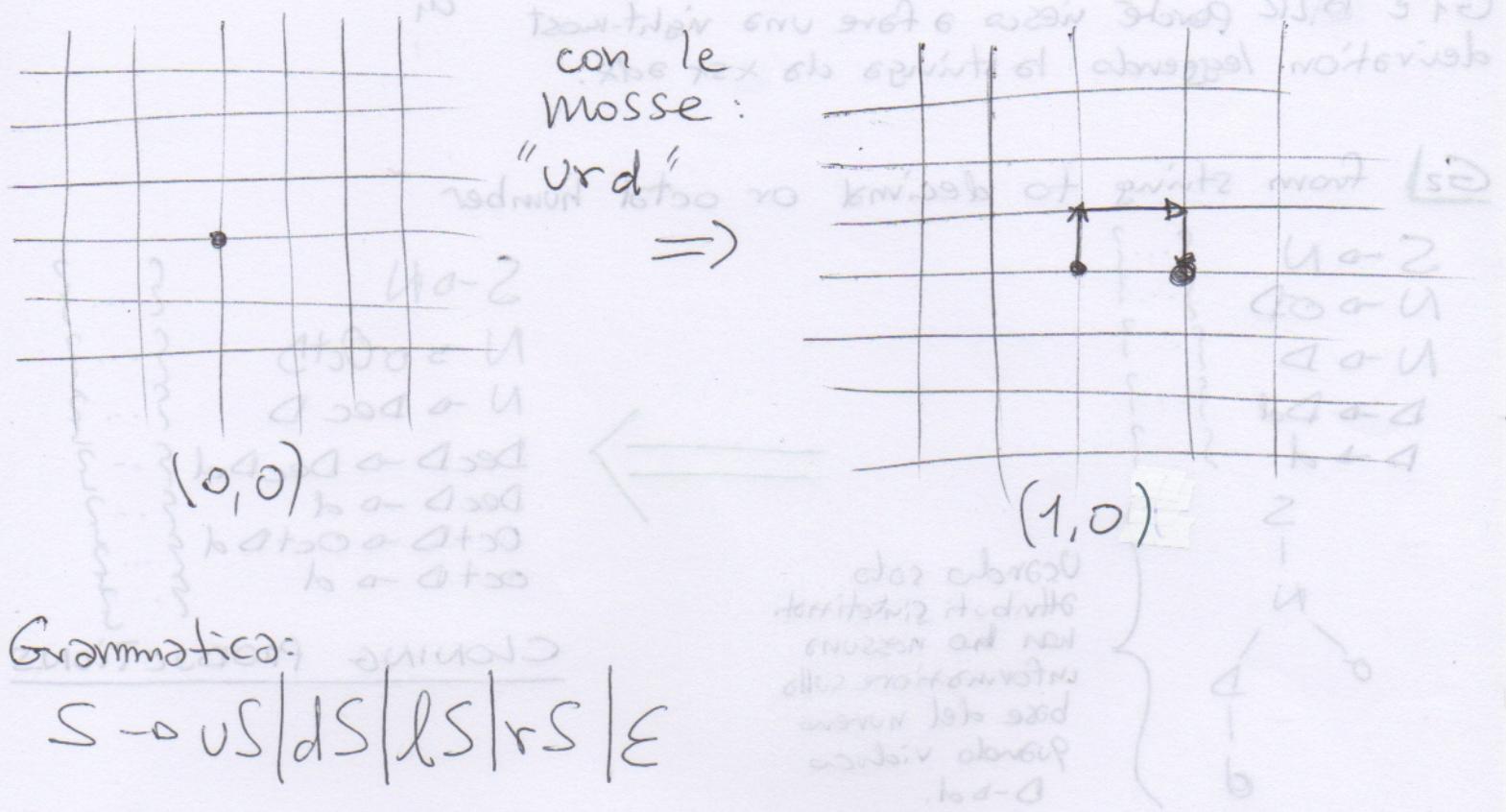
L'analisi lessicale ci fornisce già (num, 3)*(num, 5)*(num, 5)



In questo caso gli attributi sono sintetizzati, perché gli attributi dei padri sono in funzione degli attributi dei figli.

ESEMPIO

Scrivere una SDD (usando solo attributi sintetizzati) che data una locazione $(0,0)$ di partenza e una sequenza di mosse $u/d/l/r$, trova la posizione finale su una griglia unitaria.



Grammatica:

$$S \rightarrow uS | dS | lS | rS | \epsilon$$

SDD

$$L \rightarrow S$$

$$\{ L.x = S.x, L.y = S.y \}$$

$$(0, +1) \text{ up}$$

$$S \rightarrow uS_1$$

$$\{ S.x = S_1.x, S.y = 1 + S_1.y \}$$

$$(1, +0) \text{ right}$$

$$S \rightarrow dS_1$$

$$\{ S.x = S_1.x; S.y = S_1.y - 1 \}$$

$$(-1, +0) \text{ left}$$

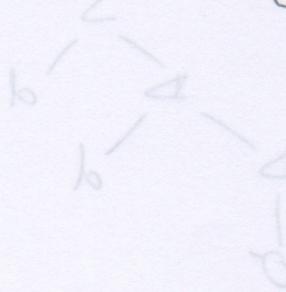
$$S \rightarrow lS_1$$

$$\{ S.x = S_1.x - 1, S.y = S_1.y \}$$

$$(0, -1) \text{ down}$$

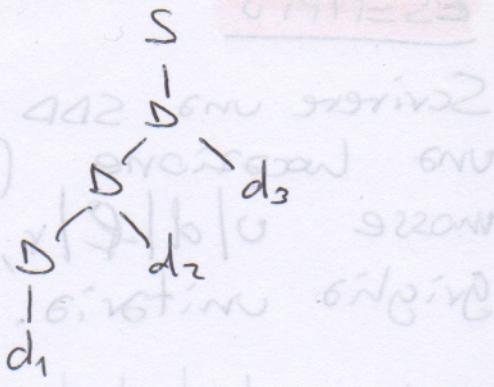
$$S \rightarrow \epsilon$$

$$\{ S.x = 0, S.y = 0 \}$$



SDD : string to number

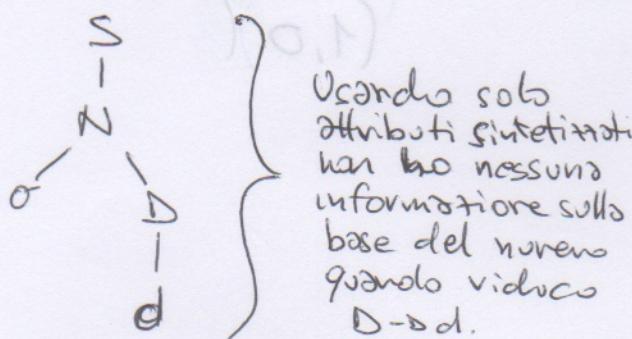
G1: $S \rightarrow D$ $\{ \text{print}(D.\text{val}) \}$
 $D \rightarrow D_1 d$ $\{ D.\text{val} = D_1.\text{val} \cdot 10 + d.\text{lex} \}$
 $D \rightarrow d$ $\{ D.\text{val} = d.\text{lex} \}$



G1 è LALR perché riesce a fare una right-most derivation leggendo la stringa da sinistra ad dx.

G2: from string to decimal or octal number

$S \rightarrow N$ $\{ \dots \}$
 $N \rightarrow oD$ $\{ \dots \}$
 $N \rightarrow D$ $\{ \dots \}$
 $D \rightarrow Dd$ $\{ \dots \}$
 $D \rightarrow d$ $\{ \dots \}$



$S \rightarrow N$ $\{ \dots \}$
 $N \rightarrow o \text{Oct} D$ $\{ \dots \}$
 $N \rightarrow \text{Dec} D$ $\{ \dots \}$
 $\text{Dec} D \rightarrow \text{Dec} D d$ $\{ \dots \}$
 $\text{Dec} D \rightarrow o d$ $\{ \dots \}$
 $\text{Oct} D \rightarrow \text{Oct} D d$ $\{ \dots \}$
 $\text{Oct} D \rightarrow o d$ $\{ \dots \}$

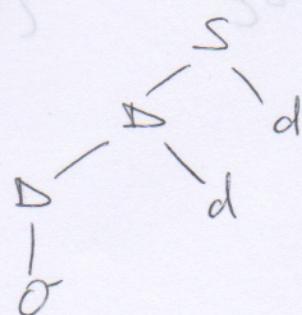
CLONING PRODUCTIONS

G3: G2 senza cloning productions, con variabili globali.

$S \rightarrow N$ $\{ \text{print}(N.\text{val}) \}$
 $N \rightarrow \text{Oct} D$ $\{ N.\text{val} = D.\text{val} \}$
 $N \rightarrow \text{Dec} D$ $\{ N.\text{val} = D.\text{val} \}$
 $\text{Oct} \rightarrow o$ $\{ \text{base} = 8 \}$
 $\text{Dec} \rightarrow e$ $\{ \text{base} = 10 \}$
 $D \rightarrow D_1 d$ $\{ \text{if } d.\text{lex} > \text{base} \text{ then error() else } D.\text{val} = D_1.\text{val} \cdot \text{base} + d.\text{lex} \}$
 $D \rightarrow d$ $\{ \text{if } d.\text{lex} > \text{base} \text{ then error() else } D.\text{val} = d.\text{lex} \}$

G4: G3 senza variabili globali

$S \rightarrow Dd$ $\{ S.\text{val} = D.\text{val} \cdot D.\text{base} + d.\text{lex}, \text{print}(S.\text{val}) \}$
 $D \rightarrow D_1 d$ $\{ D.\text{val} = D_1.\text{val} \cdot D_1.\text{base} + d.\text{lex} \}$
 $D \rightarrow d$ $\{ D.\text{base} = 10, D.\text{val} = d.\text{lex} \}$
 $D \rightarrow o$ $\{ D.\text{base} = 8, D.\text{val} = 0 \}$

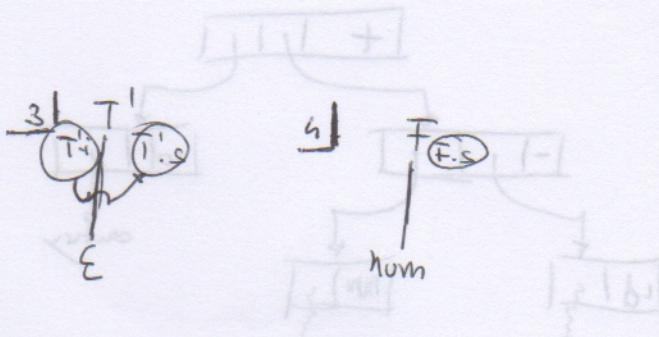
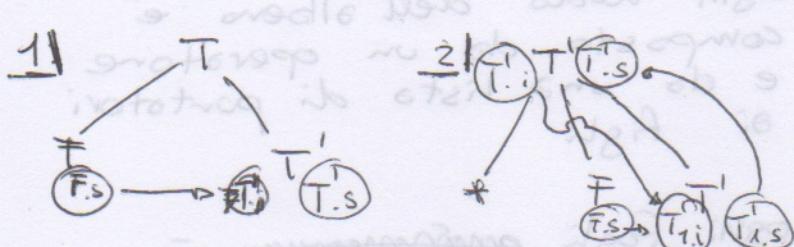


ESEMPIO

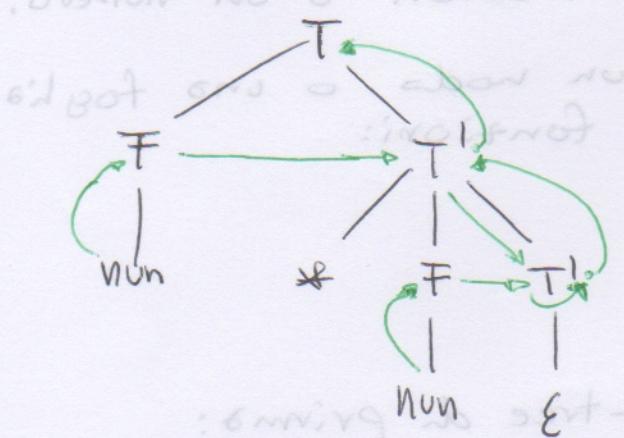
Scrivere una SDD per la grammatica dei puntatori:

- ① $T \rightarrow FT^i$ $\{T.i = F.s, T.s = T'.s\}$
- ② $T' \rightarrow *FT'_i$ $\{T'_i.i = T'.i * F.s, T'.s = T'_i.s\}$
- ③ $T' \rightarrow \epsilon$ $\{T'.s = T'_i.s\}$
- ④ $F \rightarrow \text{num}$ $\{F.s = \text{num}.lex\}$

i: creditato
s: sintetizzato.



Per la stringa num * num si ottiene:



Quello che in generale risulta è un grafo delle dipendenze, dove ogni nodo sorgente è necessario al nodo destinazione. Serve per valutare gli attributi e dargli un valore.

Per computarlo basta numerare i nodi in modo tale che il nodo sorgente abbia un valore

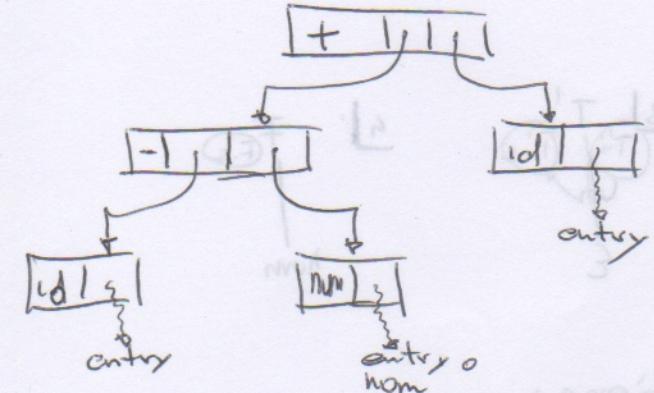
numerico inferiore alla destinazione; ~~e~~ successivamente si lancia un topological - sort.

ANALISI SEMANTICA - SDD con SYNTAX-TREE

L'albero di derivazione non è usato per parsenizzare la grammatica perché contiene troppe informazioni: sono inutili, che lo rendono troppo difficile da usare.

Per questo si usa il syntax-tree, che tiene conto solo delle informazioni essenziali.

Per esempio un syntax-tree per la stringa "2-5+c"



- ogni nodo dell'albero è composto da un operatore e da una lista di puntatori ai figli
- ogni fogli ~~paragrafo~~ è composta da un operando e un puntatore alla tabella degli identificatori o un numero.

Per specificare la creazione di un nodo o una foglia nelle SDD rules, si usano due funzioni:

Node(op, c₁, ..., c_n)

Leaf(operand, value)

Esempio che genera il syntax-tree di prima:

| | |
|-------------------------|---|
| $E \rightarrow E_1 + T$ | { G-node = new Node(+, E ₁ .node, T.node) ? |
| $E \rightarrow E_1 - T$ | { G-node = new Node(-, E ₁ .node, T.node) ? |
| $E \rightarrow T$ | { G-node = T.node ; |
| $T \rightarrow (=)$ | { T.node = E-node ; |
| $T \rightarrow id$ | { T.node = new Leaf(id, id.entry) ; |
| $T \rightarrow num$ | { T.node = new Leaf(num, num.value) ; |

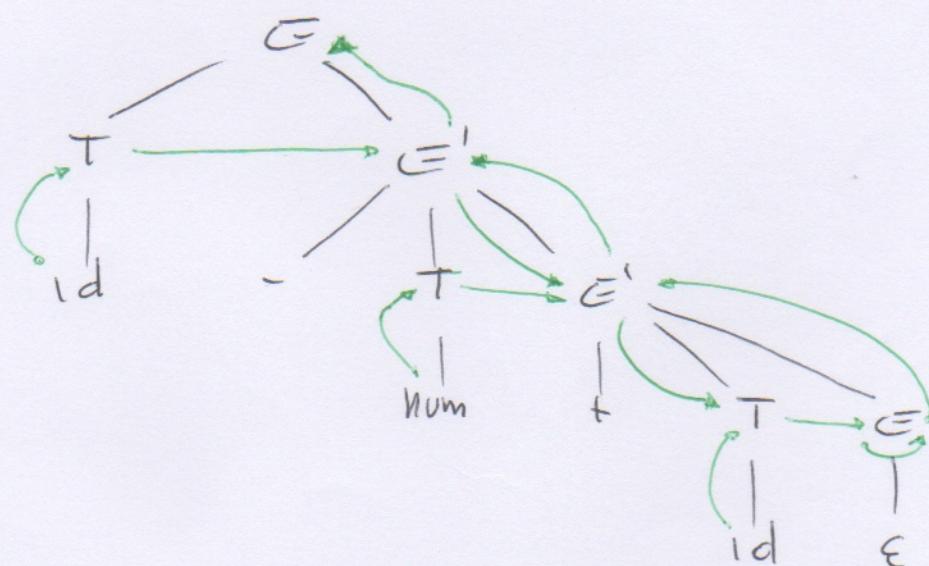
ESEMPIO

Scrivere una SDD per la generazione del syntax tree per la grammatica precedente senza la left-recursion:

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \\
 E' &\rightarrow \epsilon \\
 T &\rightarrow (\epsilon) \\
 T &\rightarrow id \\
 T &\rightarrow num
 \end{aligned}$$

$E.\text{node} = E'.s$, $E'.e = T.\text{node}$
 $E'.er = \text{new Node}(+, E'.er, T.\text{node})$, $E'.s = E'_1.s$
 $E'.s = E'.er$
 $T.\text{node} = E.\text{node}$
 $T.\text{node} = \text{new } \cancel{\text{Node}} \text{Leaf(id, id.entry)}$
 $T.\text{node} = \text{new Leaf(num, num.value)}$

stringa: 2 - 4 + c



SDT - SYNTAX DRIVEN TRANSLATION

che si può utilizzare

È una generalizzazione delle SDA; la differenza sta nel spostare alcune regole semantiche all'interno della produzione. Le regole che vengono spostate sono generalmente associate ad attributi ereditati.

Tutto questo ha il vantaggio di inserire nel syntax-tree anche le regole semantiche, in modo da eseguirle al momento opportuno.

Esempio:

$$A \rightarrow DBC \quad \{B.b = f(x)\} \quad // b \text{ ereditato}$$

↓

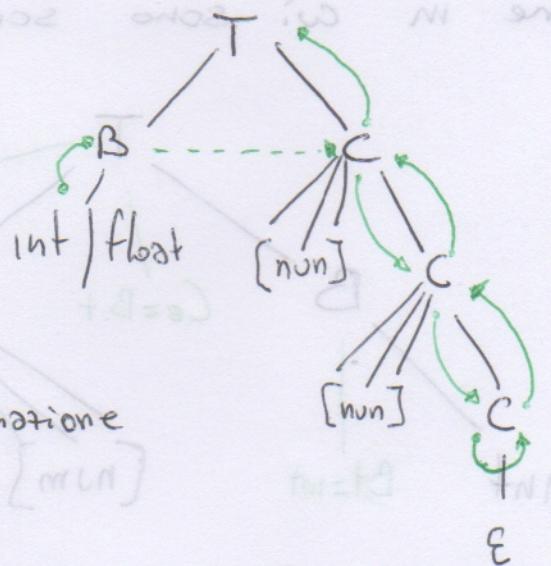
$$A \rightarrow D\{B.b = f(x)\}BC \quad \xleftarrow{\text{SDT}}$$

Esempio: con espressioni dichiarazioni di array.

$$T \rightarrow BC$$

$$B \rightarrow \text{int} / \text{float}$$

$$C \rightarrow [\text{num}] C / \epsilon$$



- con la stringa `int[2][3]` l'informazione che vogliamo ottenere è `array(2, array(3, int))`

- B avrà un attributo sintetizzato che contiene il tipo letto in input. Ma per completare l'informazione che vogliamo ottenere deve passarla a C! \Rightarrow Attributo ereditato.

Infatti con la SDD otterremo:

$$T \rightarrow BC \quad \{ T.t = C.t, C.e = B.t \}$$

$$B \rightarrow \text{int} \mid \text{float} \quad \{ B.t = \text{int}, B.t = \text{float} \}$$

$$C \rightarrow [\text{num}] C_1 \quad \{ C_1.e = C.e, C.t = \text{array}(\text{num.lex}, C_1.t) \}$$

$$C \rightarrow \emptyset \quad \{ C.t = C.e \}$$

Ma sono attributi ereditati.

Trasformazione in SAT

$$T \rightarrow B \{ C.e = B.t \} C \quad \{ T.t = C.t \}$$

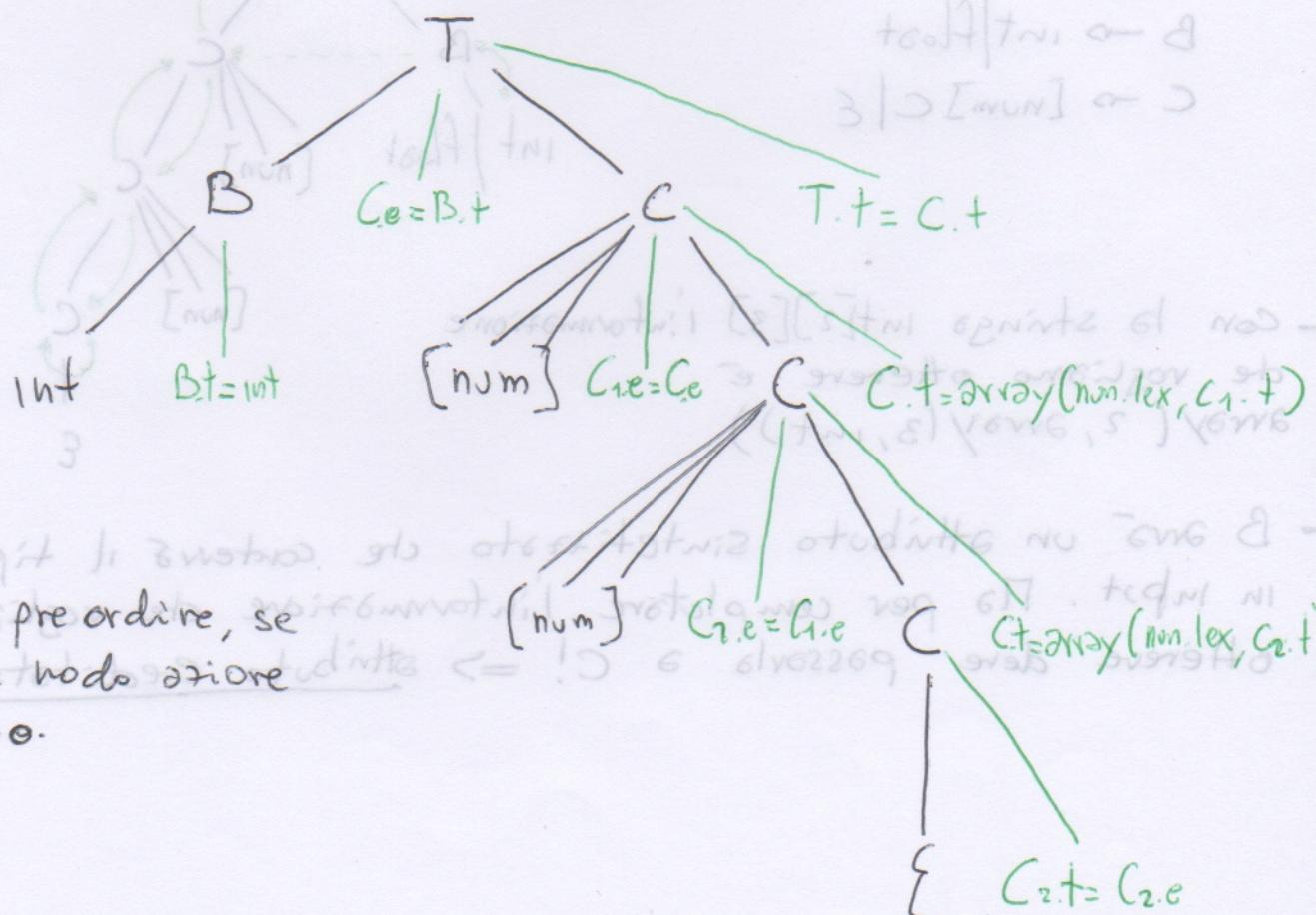
$$B \rightarrow \text{int} \quad \{ B.t = \text{int} \}$$

$$B \rightarrow \text{float} \quad \{ B.t = \text{float} \}$$

$$C \rightarrow [\text{num}] \{ C_1.e = C.e \} C_1 \quad \{ C.t = \text{array}(\text{num.lex}, C_1.e) \}$$

$$C \rightarrow \emptyset \quad \{ C.t = C.e \}$$

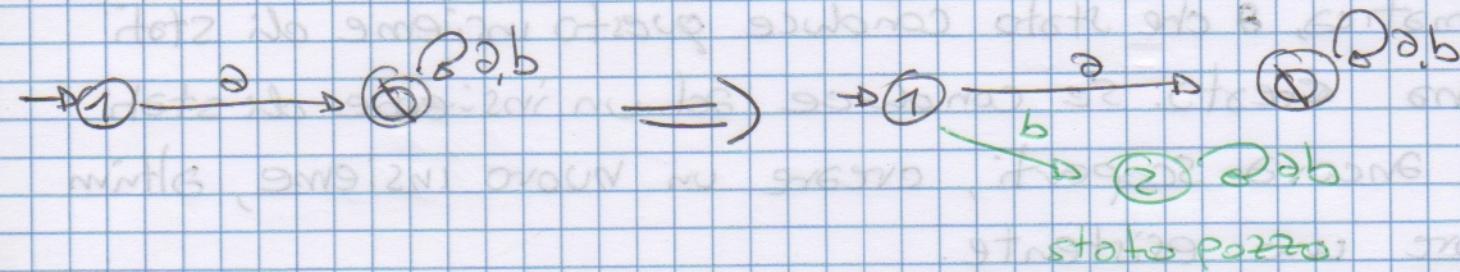
- 1) Scrivere il parse tree della stringa considerata
- 2) Aggiungere nodi al parse tree per ogni produzione nell'ordine in cui sono scritte nella grammatica.



- 3) Visito in preordine, se trovo un nodo azione lo eseguo.

DFA \rightarrow minDFA : PARTITION REFINEMENT

Scopo: raggruppare gli stati in classi di equivalenza.
 NB: funzione di transizione deve essere totale, ovvero ogni stato del DFA deve avere un arco uscente per ogni simbolo del linguaggio.



Algoritmo: creare due gruppi: G_1 e G_2 dove G_1 contiene tutti gli stati terminali e G_2 , rimanenti. Per ogni gruppo ci possono stare stati con caratteristiche simili, ovvero se la transizione con un non-terminali conduce sempre a stati dello stesso gruppo. In caso contrario si crea un nuovo gruppo con lo stato "diverso". In un gruppo possono stare stati anche che hanno transizioni per non-terminali tutte rivolte verso un altro gruppo.

$$G_1 = \{E\} \quad G_2 = \{A, B, C, D\}$$

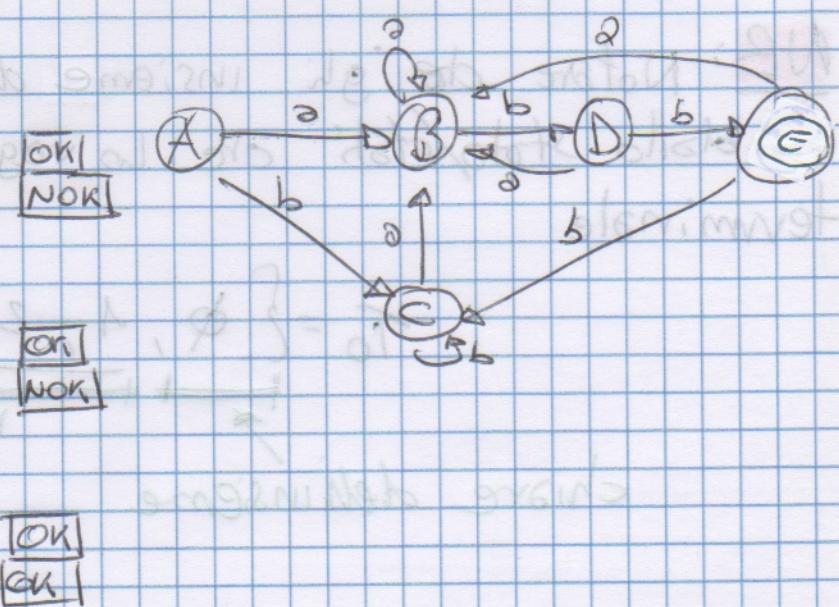
a-trans: $A \xrightarrow{a} B, B \xrightarrow{a} B, D \xrightarrow{a} B, C \xrightarrow{a} B$ OK!
 b-trans: $A \xrightarrow{b} C, B \xrightarrow{b} D, C \xrightarrow{b} C, D \xrightarrow{b} E$ NOK!

$$G_1 = \{E\} \quad G_2 = \{A, B, C\} \quad G_3 = \{D\}$$

a-trans: $A \xrightarrow{a} B, B \xrightarrow{a} B, E \xrightarrow{a} B$
 b-trans: $A \xrightarrow{b} C, C \xrightarrow{b} C, B \xrightarrow{b} D$ NOK!

$$G_1 = \{E\}; G_2 = \{A, C\}, G_3 = \{D\} \quad G_4 = \{B\}$$

a-trans: $A \xrightarrow{a} B, C \xrightarrow{a} B$
 b-trans: $A \xrightarrow{b} C, C \xrightarrow{b} C$ OK!



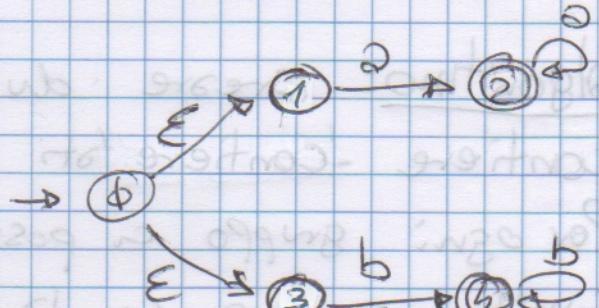
NFA \rightarrow DFA : SUBSET CONSTRUCTIONS

Scopo: eliminazione dell'indeterminismo dell'automa
Algoritmo: ~~per~~ stato dell'NFA (partendo da quello iniziale),
 si crea un insieme di stati raggiungibili con una ϵ -mossa,
 successivamente controllare, ~~se~~ simbolo terminale della
 grammatica, se che stato conduce questo insieme di stati
 appena creato: se conduce ad un insieme di stati
 non ancora scoperti, creare un nuovo insieme, altrimenti
 uno esistente.

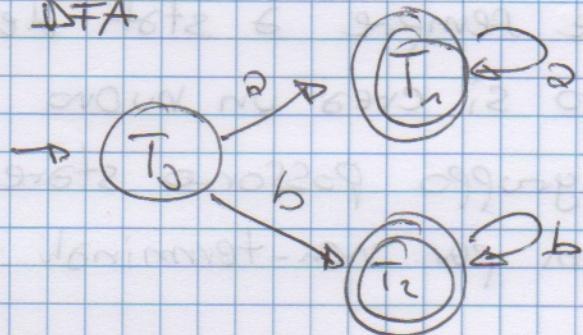
REG EXP: $aa^* \mid bb^*$

| | a | b |
|-----------------------------|-------|-------|
| $T_0 = \{\emptyset, 1, 2\}$ | T_1 | T_2 |
| $T_1 = \{2\}$ | T_3 | |
| $T_2 = \{1\}$ | / | T_4 |

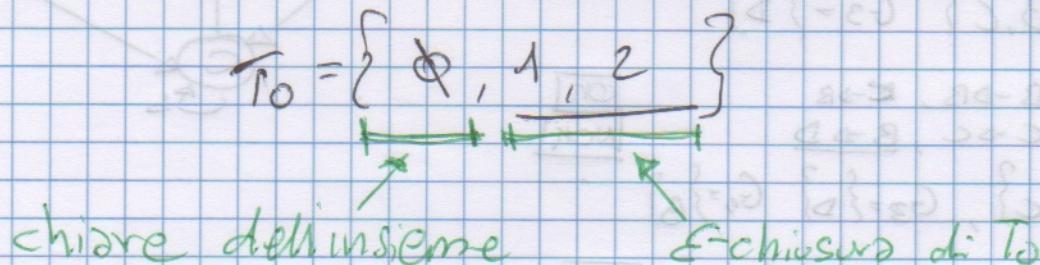
NFA



DFA



NB: Notare che gli insiemi di stati vengono identificati dallo stato/stati che lo raggiungono tramite un non-terminale



LL(1) - compitare i FIRST e FOLLOW per ogni non-terminal
(top-down) costruire una tabella con RIGHE \rightarrow non-terminali e
COLUMNI \rightarrow terminali + $\{\$\}$ in questo modo:

for each $A \rightarrow \alpha \in P$

$\begin{cases} \forall b \in \text{FIRST}(\alpha) \rightarrow \text{add } A \rightarrow \alpha \text{ to } T[A, b] \\ \text{if } (\epsilon \in \text{FIRST}(\alpha)) \\ \quad \text{add } A \rightarrow \alpha \text{ to } T[A, x] \text{ where } x \in \text{Follow}(A) \end{cases}$

NB: $A \rightarrow \epsilon$ va messo nella tabella di parsing.

LR(0) - costruire il grafo degli LR(0)-ITEM
(SLR(1)) costruire la tabella con RIGHE = numero dello stato del grafo e COLUMNI \rightarrow terminali + non-term (con starting symbol) + $\{\$\}$, in questo modo:

- transazione $\phi \xrightarrow{\alpha} z \Rightarrow T[\phi, \alpha] = z$ [SHIFT]
- transazione $\phi \xrightarrow{A} z \Rightarrow T[\phi, A] = z$ [GOTO]

per ogni stato del grafo degli LR(0)-ITEM con almeno una produzione col punto in fondo:

- $T[K, x] = A \rightarrow \alpha$. dove:

K : numero dello stato del grafo degli LR(0)-ITEM

x : $x \in \text{FOLLOW}(A)$

NB: $A \rightarrow .$ NON deve essere considerato in questo ultimo passaggio.

$LR(1)$ - costruire il grafo degli $LR(1)$ -ITEMS.
(bottom-up) costruzione della tabella come $LR(0)$ ad eccezione delle reduce che vanno messe in corrispondenza del lookahead dell' $LR(1)$ -ITEM.

NB: $A \rightarrow .$, e NON vengono considerati.