

(1)

PDA e linguaggi deterministici

Def Un PDA $N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ è deterministico (DPDA) se

$$(1) \forall q \in Q \quad \forall z \in \Gamma, \text{ se } \delta(q, \epsilon, z) \neq \emptyset$$

$$\text{allora } \delta(q, a, z) = \emptyset \quad \forall a \in \Sigma$$

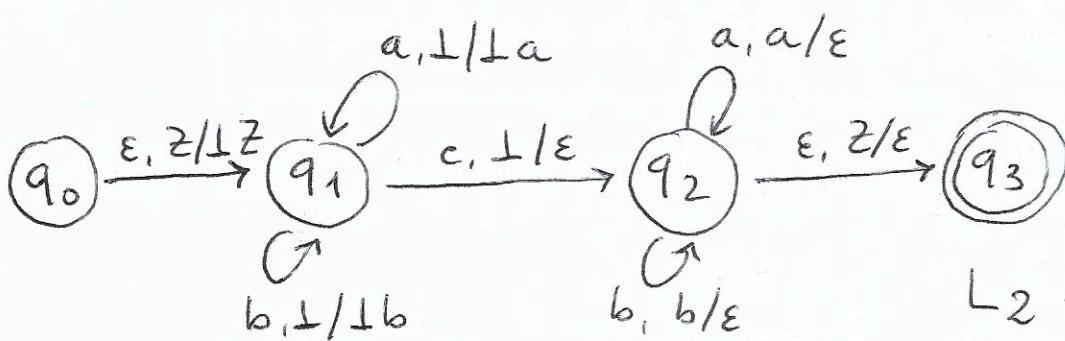
$$(2) \forall q \in Q \quad \forall z \in \Gamma \quad \forall a \in \Sigma \cup \{\epsilon\} \quad |\delta(q, a, z)| \leq 1$$

Def Un linguaggio è libero deterministico se è accettato per stato finale da un DPDA

Teorema La classe dei ling. liberi deterministici è inclusa propriamente nella classe dei ling. liberi.

Ese: $L_1 = \{ww^R \mid w \in \{a,b\}^*\}$ è libero, ma si può dimostrare non esistere un DPDA che lo riconosce

$L_2 = \{wcw^R \mid w \in \{a,b\}^*\}$ è libero deterministico



L_2 riconosciuto
da un DPDA
per stato finale

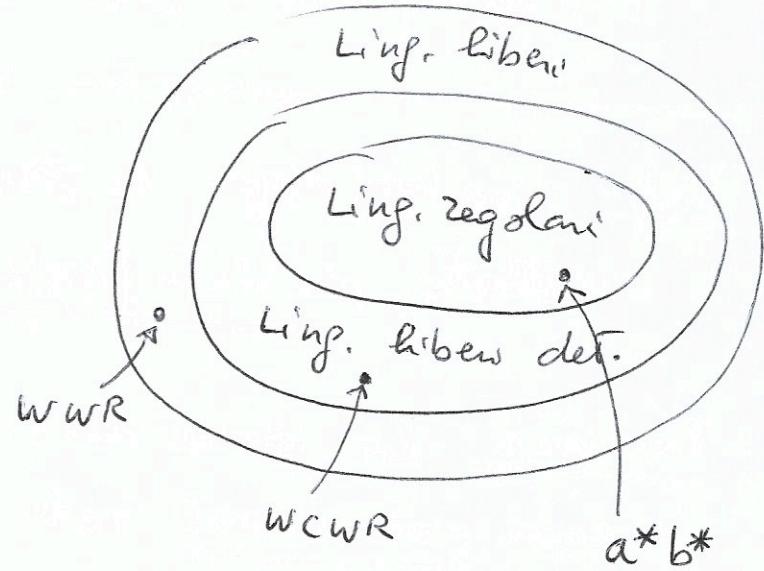
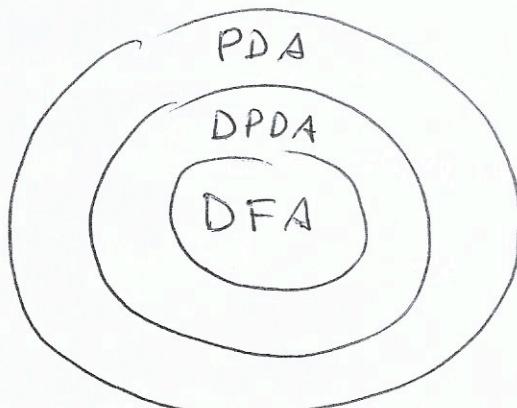
Proposizione Se L è regolare, allora \exists DPDA N
 tale che $L = L[N]$
 per stato finale

Dim: Se L è regolare, allora \exists DFA M tale che $L = L[M]$.

A partire da M , posso costruire un DPDA N che si comporta
 come M senza mai manipolare lo stack

$$\Rightarrow L = L[N]$$

(2)

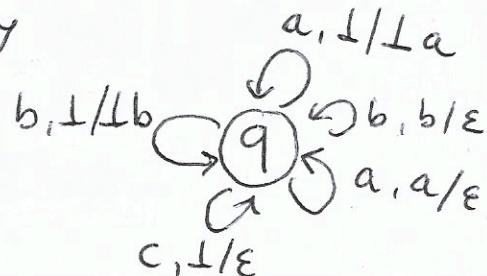


Fatto: Un linguaggio libero deterministico L è
 riconosciuto da un DPDA per pila vuota ~~se solo se~~
 L gode della "prefix property"

(condizione
 necessaria
 e sufficiente)

$\boxed{\forall x, y \in L \text{ tal che } x \text{ è prefisso di } y}$

Osserva che $L_2 = \{w \in \{a,b\}^* \mid w \in L\}$ gode della prefix property



DPDA che riconosce
 L_2 per pila vuota

Quindi

libero deterministico

(3)

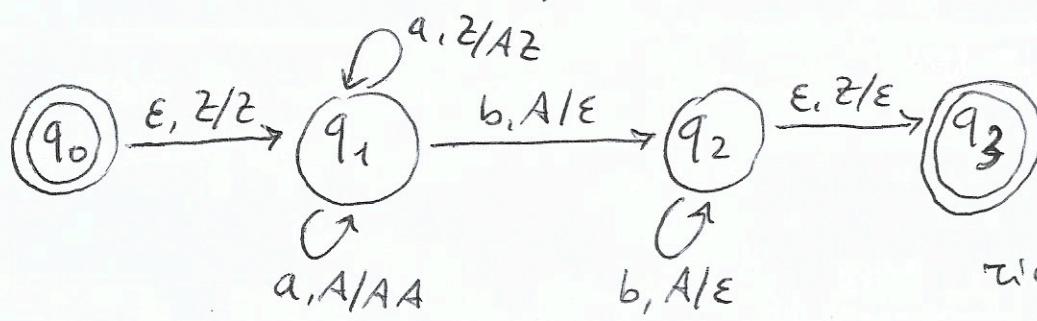
(1) Se $L \setminus \{ \}$ non gode della prefix property, non può essere riconosciuto da un DPDA per pila vuota.

(2) Se L libero det. gode della prefix property, allora può essere riconosciuto da un DPDA per pila vuota.

3) Se L è libero det., allora $L\# = \{ w\# \mid w \in L \}$ gode della prefix property.

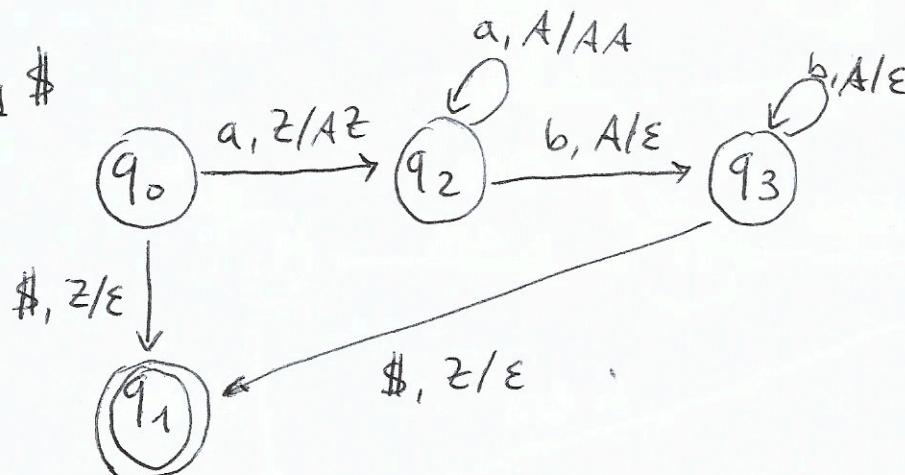
$\Rightarrow L\#$ può essere riconosciuto da un DPDA per pila vuota

$L_1 = \{ a^n b^n \mid n \geq 0 \}$ non gode della prefix property
perché $\epsilon \in L_1$ e ϵ è prefisso di ab



riconosce per stato finale

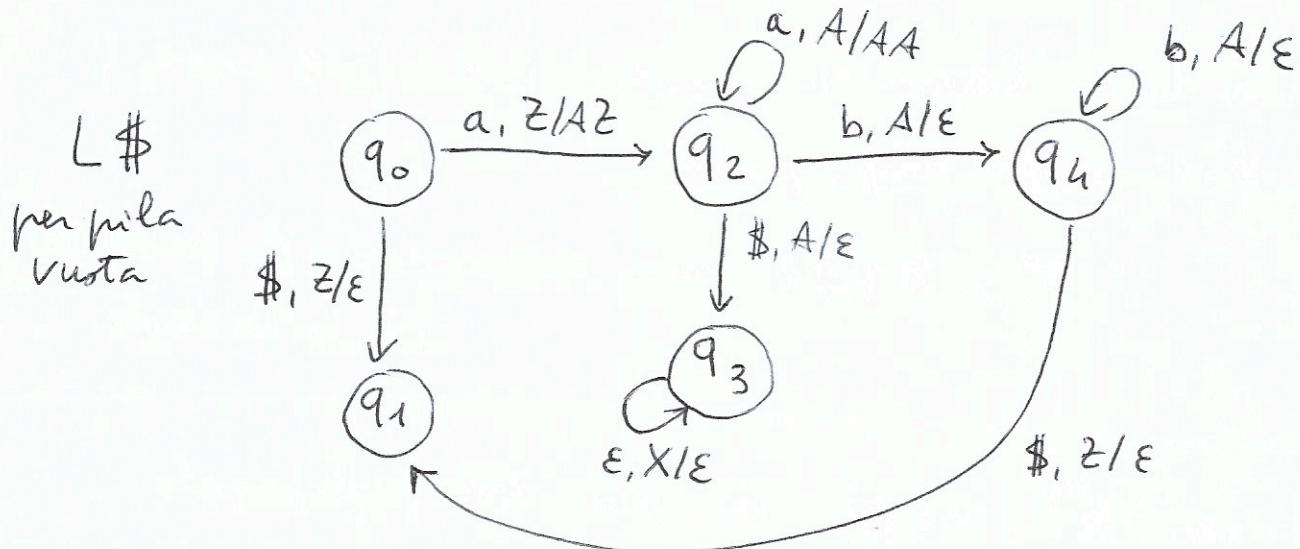
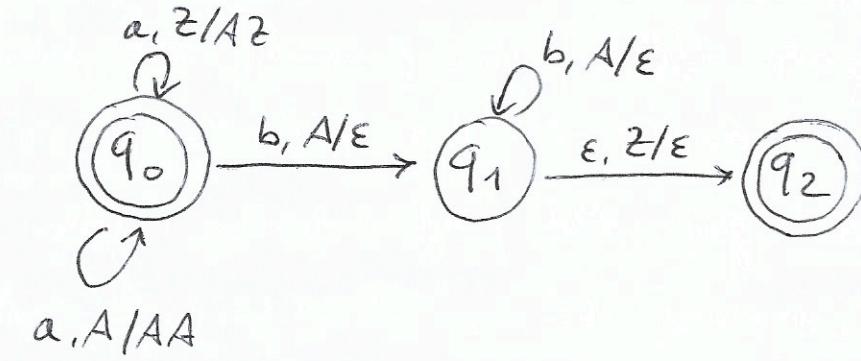
$L_1\#$



riconosce per pila vuota
(ed anche per stato finale)

$$L = a^* \cup \{a^n b^n \mid n \geq 1\}$$

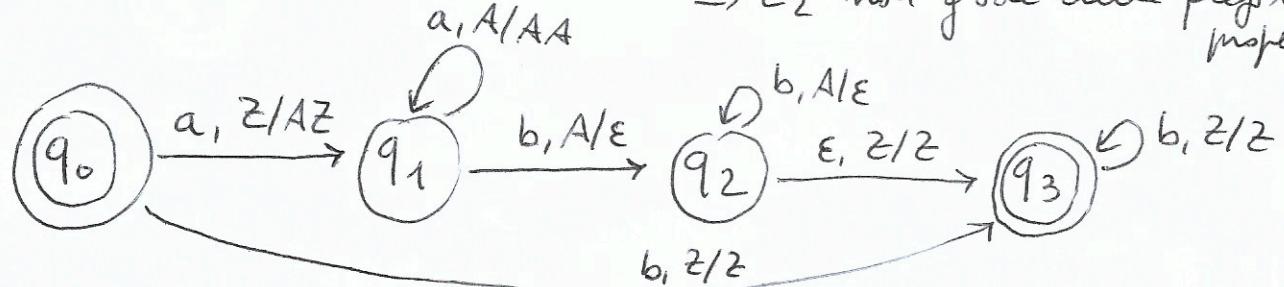
non gode della prefix property



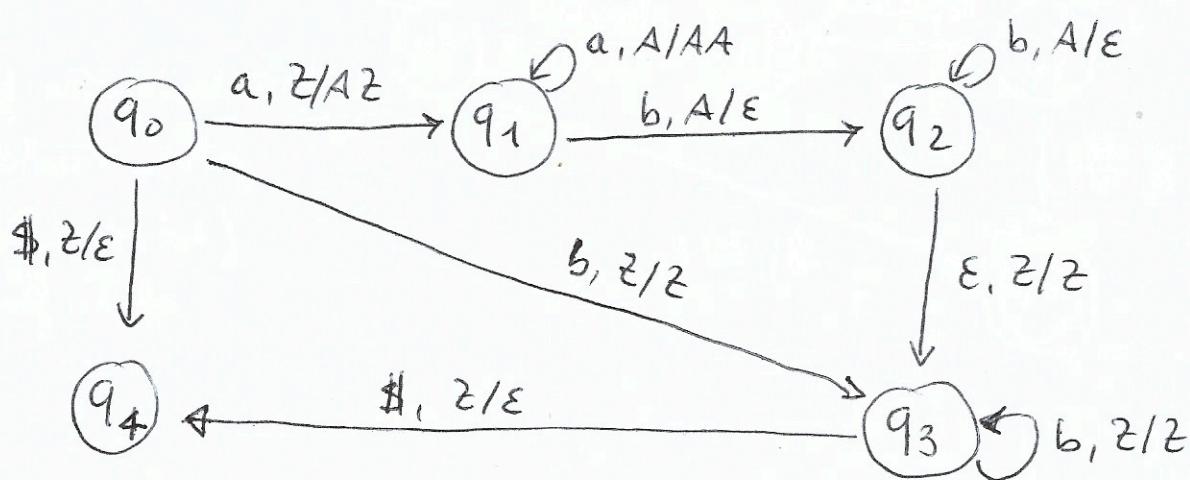
$$L_2 = \{a^n b^m \mid n \leq m\}$$

ab prefisso di abb

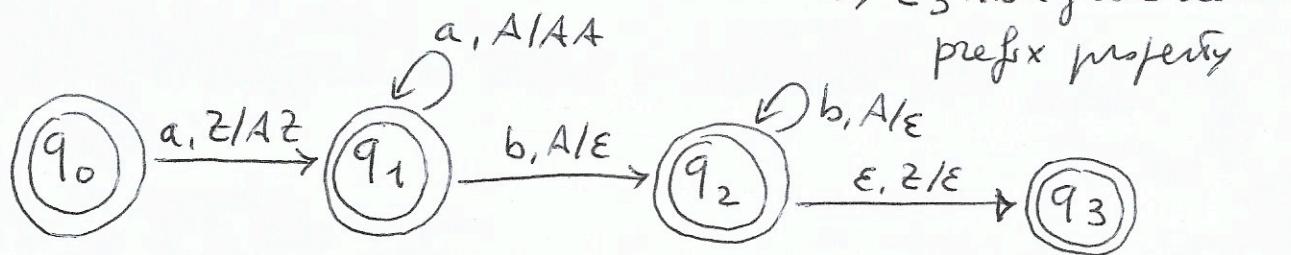
$\Rightarrow L_2$ non gode delle prefix property



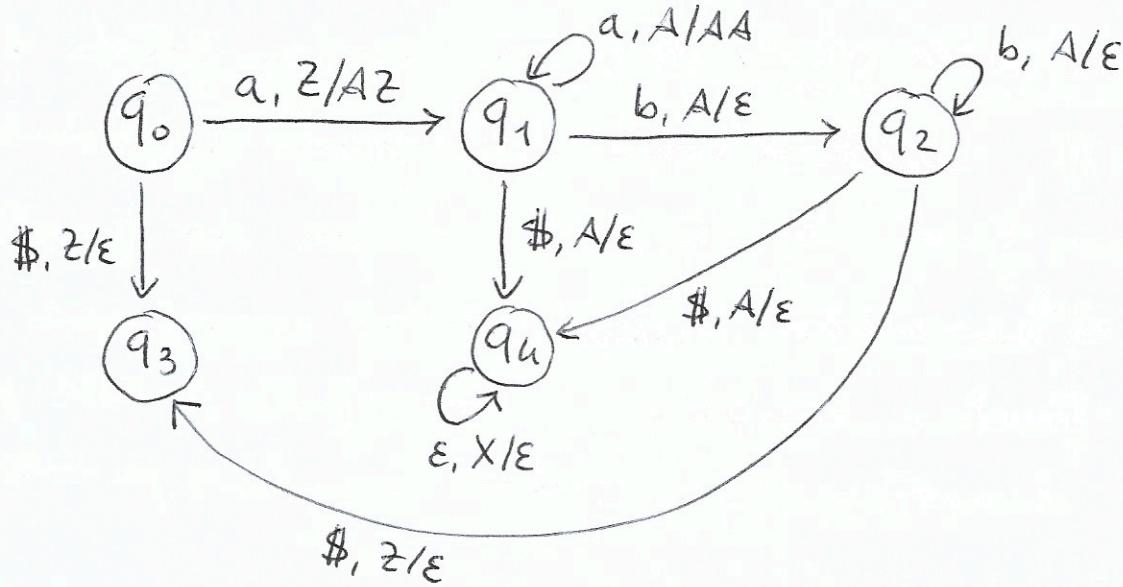
$L_2 \#$



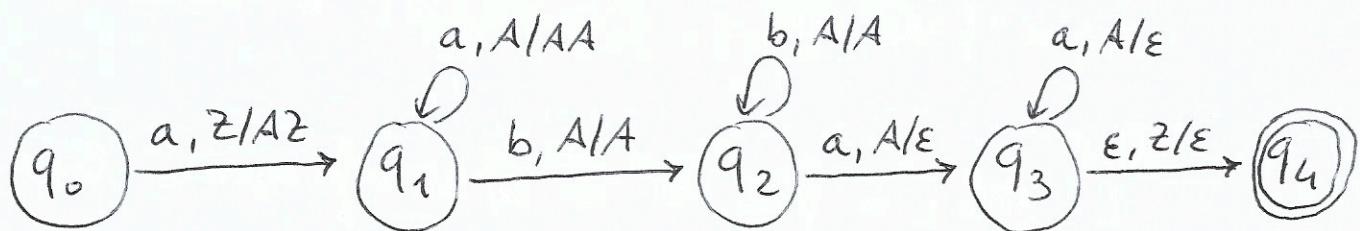
$$L_3 = \{ a^n b^m \mid n \geq m \geq 0 \} \quad \epsilon \in L_3 \quad (5)$$



$L_3 \$$
per nulla
vuota



$$L_4 = \{ a^n b^m a^n \mid n, m \geq 1 \} \quad \text{gode della prefix property !!}$$



Proposizione Se L è libero deterministico (cioè riconosciuto da un DPDA per stato finale), allora L è generabile da una grammatica libera

NON AMBIGUA

\Rightarrow I lang. liberi deterministici non sono ambigui !!

Proprietà dei lang. liberi deterministici

(6)

- Sono chiusi per complementazione, cioè se $\exists \text{DPDA } N \text{ tale che } L = L[N]$, allora esiste un DPDA N' tale che $\bar{L} = L[N']$, dove $\bar{L} = \Sigma^* \setminus L$.

Idea della prova: Bisogna rendere totale la δ di N , eventualmente aggiungendo degli stati non finali, e poi N' si ottiene da questo N "aumentato", semplicemente scambiando finali e non finali (come fatto con DFA per dimostrare che i regolari sono chiusi per complementazione)

- Non sono chiusi per intersezione

$L_1 = \{a^n b^m c^m \mid n, m \geq 0\}$ è libero det.

$L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$ è libero det.

ma

$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ non è libero!

- Non sono chiusi per unione

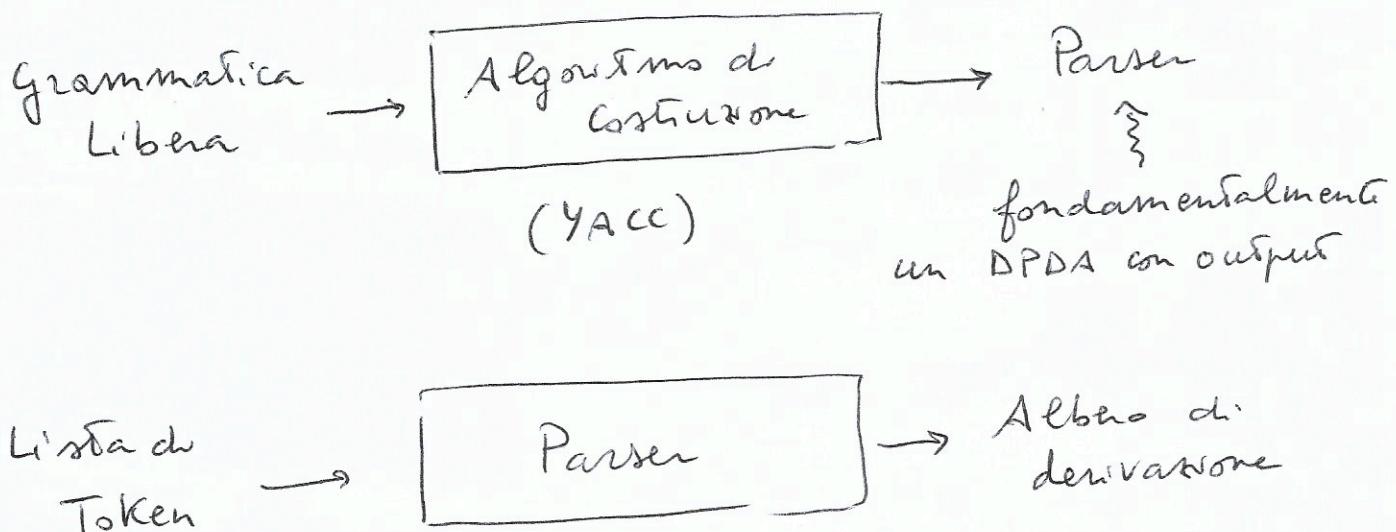
Se, per assurdo, lo fossero, allora

$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$$

e quindi risulterebbero chiusi per intersezione
 \Rightarrow impossibile!

Analizzatori Sintattici (Parser)

(7)



I Parser possono essere

- Nondeterministici: se, durante la ricerca di una derivazione, si scopre che una scelta è improduttiva e non porta a riconoscere l'input, il parser torna indietro (backtracking), disfa parte della derivazione appena costituita, e sceglie un'altra produzione, tornando a leggere (parte de) l'input
- Deterministici: leggono l'input una sola volta; ogni loro decisione è definitiva

Entrambi cercano di sfruttare informazioni dell'input per guidare la ricerca della derivazione

I Parser possono essere:

(8)

- Top-down

Ricostuiscono una derivazione leftmost per la parola a partire dal simbolo iniziale S (all'inizio della parola)

- Bottom-up

Ricostuiscono una derivazione rightmost (a rovescio) a partire dalla stringa w , cercando di ridurla al simbolo iniziale S (alla fine della parola)

Entrambi cercano di sfruttare quello che vedono dell'input per guidare la ricerca della derivazione

Noi vedremo Parser top-down deterministico (ottenuti a partire da grammatiche $LL(k)$, in particolare $LL(1)$) e Parser bottom-up deterministico (ottenuti a partire da grammatiche $LR(k)$, in particolare $LR(0)$, $SLR(1)$, $LR(1)$ e $LALR(1)$).

Introduzione al Top-down Parsing

(9)

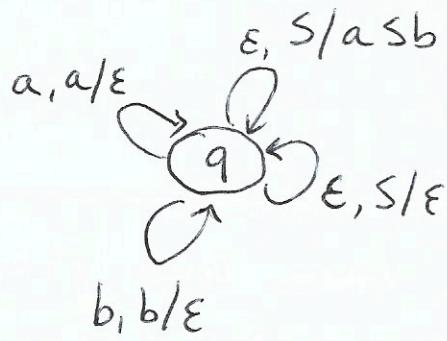
Data $G = (NT, T, S, R)$ libera, costruiamo il PDA

$M = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$, che riconosce per pila vuota, dove δ è definita:

- $(q, \beta) \in \delta(q, \epsilon, A)$ se $A \rightarrow \beta \in R$ (espandi)
- $(q, \epsilon) \in \delta(q, a, a)$ $\forall a \in T$ (consuma)

talché $L(G) = P[M]$
 è riconoscimento per pila vuota

Esempio $S \rightarrow aSb \mid \epsilon$ $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

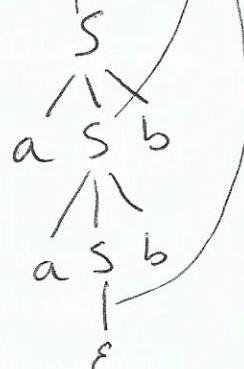
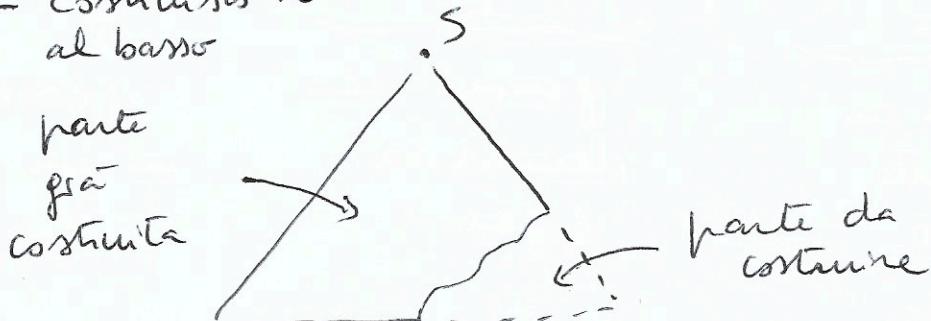


$(q, aabb, S) \xrightarrow{\quad} (q, aabb, aSb)$
 $\vdash (q, abb, Sb) \xrightarrow{\quad} (q, abb, aSbb)$
 $\vdash (q, bb, Sbb) \xrightarrow{\quad} (q, bb, bb)$
 $\vdash (q, b, b) \vdash (q, \epsilon, \epsilon)$

- Derivazione canonica SX

(leftmost)

- costruisce l'albero dall'alto
al basso



- Nondeterminismo! Può essere risolto scegliendo la produzione in base al simboli in lettura (look-ahead!) (10)

- se leggo "a", \Rightarrow espandsi $S \rightarrow aSb$
- se leggo "b", \Rightarrow espandsi $S \rightarrow \epsilon$

<u>input</u>	<u>stack</u>
a a bb \$	S
<u>a</u> bb \$	$a S b$
<u>a b</u> b \$	$S b$
<u>a S</u> b b \$	$a S b b$
<u>b b</u> \$	$S b b$
<u>b</u> \$	b
\$	ϵ

$G \left[S \rightarrow aSb \mid \epsilon \right]$ vedremo essere una grammatica di classe LL(1) nel senso che consente di costruire un Parser/DPDA

<u>left-to-right</u>	/	<u>left-derivation</u>	(1)
<u>L</u>		<u>L</u>	
modi in cui		genera una	
viene letto		derivaione	
l'input		leftmost	
			guardando
			un solo simbolo
			di look-ahead

$$G[S \rightarrow aSb | ab]$$

(11)

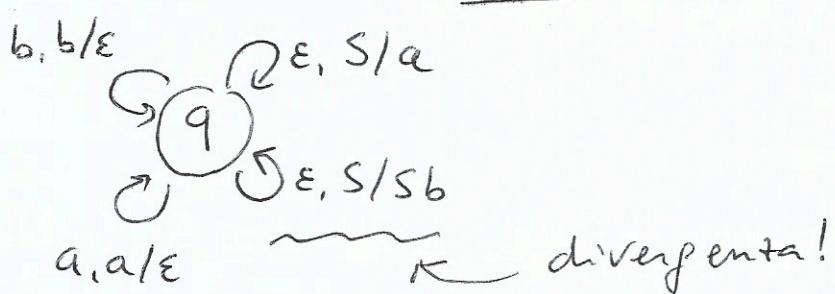
In questo caso devo guardare 2 simboli di look-ahead (in avanti)

- se leggo "aa", \Rightarrow espando $S \rightarrow aSb$
- se leggo "ab", \Rightarrow espando $S \rightarrow ab$

G è di classe LL(2)

Attenzione! Non tutte le grammatiche sono adatte per top-down parsing

Esempio 1 $G[S \rightarrow Sb | a] \quad L(G) = ab^*$
 G è left-recursive!



$(q_0, ab, S) \vdash (q_0, ab, Sb) \vdash (q_0, ab, Sbb) \vdash \dots$

- non consumo mai l'input
- continuo ad espandere S e a scrivere sulla pila

\Rightarrow bisogna manipolare la grammatica G per ottenerne una equivalente G' senza ricorso a sinistra

$$G' \left[\begin{array}{l} S \rightarrow aA \\ A \rightarrow bA / \epsilon \end{array} \right] \quad L(G) = L(G')$$

(12)

G' è adatta al top-down parsing

- se leggo "a" e sulla pila c'è S
 \Rightarrow espando con $S \rightarrow aA$
- se leggo "b" e sulla pila c'è A
 \Rightarrow espando con $A \rightarrow bA$
- se leggo "#" e sulla pila c'è A
 \Rightarrow espando con $A \rightarrow \epsilon$

G' è LL(1)

Esempio 2

$$S \rightarrow abB \mid acC$$

$$B \rightarrow \dots$$

$$C \rightarrow \dots$$

- se leggo "a" e sulla pila c'è S , allora come espando S ?

- o leggo due simboli ($ab \Rightarrow S \rightarrow abB$
 $ac \Rightarrow S \rightarrow acC$)

- oppure "Follow set"

$$S \rightarrow aE$$

$$E \rightarrow bB \mid cC$$

!

Ora non c'è mondet:
basta un solo simbolo
di look-ahead

Introduction al bottom-up parsing

(13)

Data una grammatica libera $G = (NT, T, R, S)$, costruiamo un PDA M che riconosce $L(G)$. $\#$

$$M = (T, \{q\}, T \cup NT \cup \{Z\}, \delta, q, Z, \emptyset)$$

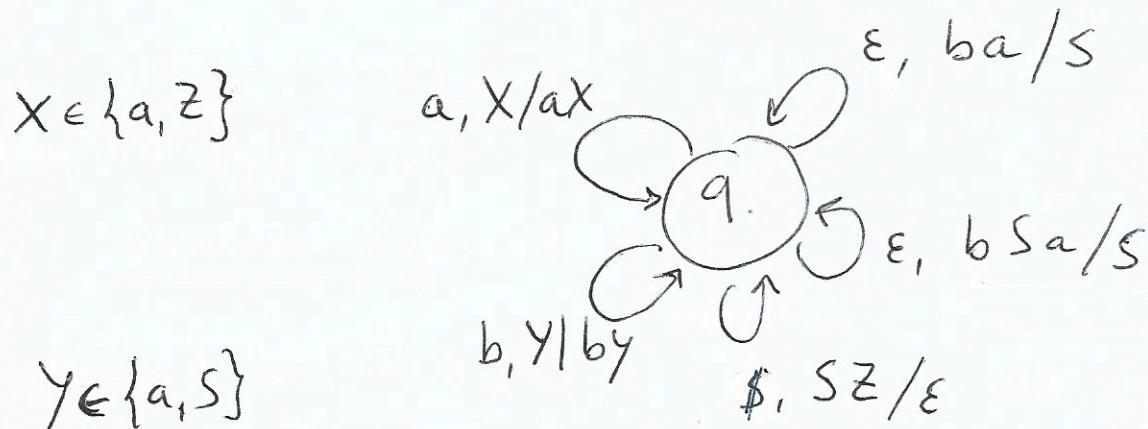
dove

- where

 1. $(q, ax) \in \delta(q, a, x)$ $\forall a \in T$ $\forall x \in \{a\}^*$ (SHIFT)
 2. $(q, A) \in \delta(q, \epsilon, \underline{\alpha^R})$ se $A \rightarrow \alpha \in R$ (REDUCE)
 3. $(q, \epsilon) \in \delta(q, \$, SZ)$ (ACCEPT)
 - \uparrow S deve essere alla fine sulla pile
 - \uparrow $\$$ simbolo di fine input

generalizzazione
dei PDA in cui
si consuma una
stringa sulla pile
anche sotto il top

$G[S \rightarrow aSb \mid ab]$

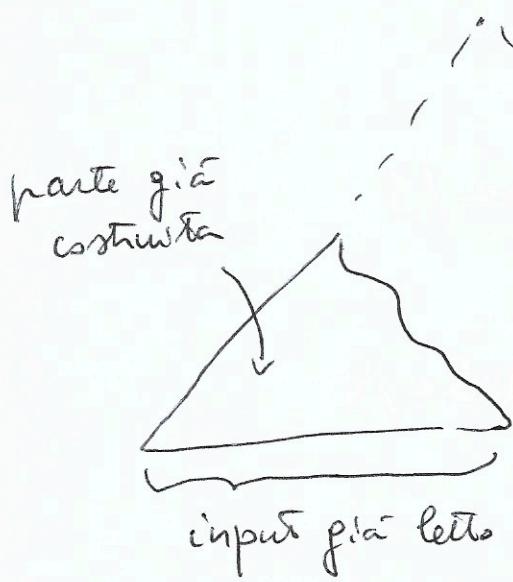
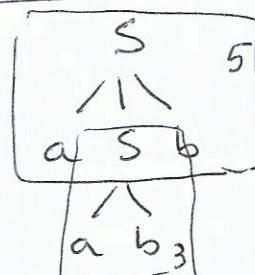


(14)

	<u>Stack</u> <small>cresce verso dx</small>	<u>Input</u>	<u>Action</u>
0)	ϵ	aabb \$	shift
1)	za	abb \$	shift
2)	zaa	bb \$	shift
3)	zaab	b \$	reduce $S \rightarrow ab$
4)	zaS	b \$	shift
5)	zaSb	\$	reduce $S \rightarrow aSb$
6)	zS	\$	<u>Accept</u>
7)	ϵ	ϵ	

$aabb \Leftarrow aSb \Leftarrow S$

derivazione canonica destra
a rovescio



parte da costituire
da leggere

L left-to-right
(come leggo l'input)
R right-derivation

LR(K) se l'automa
risultante è deterministico
usando K simboli di look-ahead

C'è molto nondeterminismo !

15

- confitti: shift-reduce

3) Zaab b\$ potrei fare shift

4') Zaabb \$ ma è un percorso
infruttuoso

- confitti: reduce-reduce (più possibile)

(non ci sono in quest'esempio, ma con grammatiche più complesse è possibile)

⇒ Per ottenere un DPDA, serve introdurre informazioni aggiuntive per risolvere i conflitti

- più stati (o strutture particolari di supporto alla decisione: DFA dei prefissi viabili)
- look-ahead (guardare l'input in avanti)

Vediamo un esempio più corposo relativo alla grammatica delle espressioni aritmetiche

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

$$T \rightarrow T * A \mid A$$

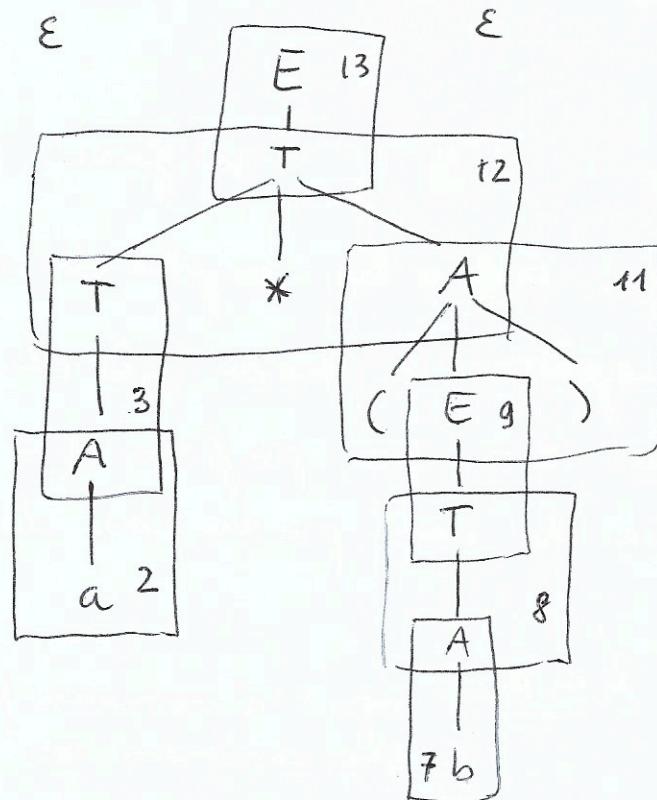
$$A \rightarrow a \mid b \mid (E)$$

G	M
	$t_0 \quad \delta(q, a, x) = (q, ax) \quad a \in X \quad \text{SHIFT}$
$(R_1) E \rightarrow T+E$	$t_1 \quad \delta(q, \epsilon, E+T) = (q, E)$
$(R_2) E \rightarrow T$	$t_2 \quad \delta(q, \epsilon, T) = (q, E)$
$(R_3) T \rightarrow T*A$	$t_3 \quad \delta(q, \epsilon, A*T) = (q, T)$
$(R_4) T \rightarrow A$	$t_4 \quad \delta(q, \epsilon, A) = (q, T)$
$(R_5) A \rightarrow (E)$	$t_5 \quad \delta(q, \epsilon,)E() = (q, A)$
$(R_6) A \rightarrow a$	$t_6 \quad \delta(q, \epsilon, a) = (q, A)$
$(R_7) A \rightarrow b$	$t_7 \quad \delta(q, \epsilon, b) = (q, A)$
	$t_8 \quad \delta(q, \$, EZ) = (q, \epsilon) \quad \text{ACCEPT}$

$$a * (b) \Leftarrow A * (b) \Leftarrow T * (b) \Leftarrow T * (A) \Leftarrow \\ \Leftarrow T * (T) \Leftarrow T * (E) \Leftarrow T * A \Leftarrow T \Leftarrow E$$

derivatione canonica dx a reverse
(rightmost)

	<u>Stack</u>	<u>Input</u>	<u>Action</u>
0)	Z	$a * (b) \$$	shift
1)	$Z a$	$* (b) \$$	reduce R6
2)	$Z A$	$* (b) \$$	reduce R4
3)	$Z T$	$* (b) \$$	shift
4)	$Z T *$	$(b) \$$	shift
5)	$Z T * ($	$b) \$$	shift
6)	$Z T * (b$	$) \$$	reduce R7
7)	$Z T * (A$	$) \$$	reduce R4
8)	$Z T * (T$	$) \$$	reduce R2
9)	$Z T * (E$	$) \$$	shift
10)	$Z T * (E)$	$\$$	reduce R5
11)	$Z T * A$	$\$$	reduce R3
12)	$Z T$	$\$$	reduce R2
13)	$Z E$	$\$$	<u>accept</u>
14)	ϵ	ϵ	



costruzione
dell'albero
bottom-up!

L'automa è nondeterministico perché

18

(1) chi ha precedenza tra shift e reduce?

Ad es:

1) $Za * (b)\$$ shift

2') $Za* (b)\$$ non buono!

qui reduce è meglio
di shift!

Altro es:

3) $ZT * (b)\$$ reduce R_2

4') $Z E * (b)\$$ non buono!

qui shift è meglio
di reduce!

(2) chi ha precedenza tra 2 diverse reduce?

Ad es:

(11) $ZT * A \$$ reduce R_3

(12) $Z T$

(12') $ZT * T \$$ se faccio reduce R_4

qui reduce R_3 meglio di reduce R_4 !

È buona norma scegliere in modo tale che ciò che si trova sulla pila sia un prefisso (a rovescio) di una parte destra di una produzione della gram.!!

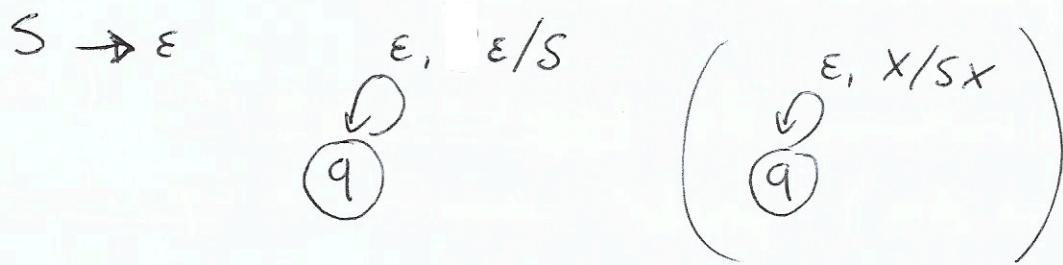
Ad es

scelgo 2) $\Leftrightarrow \begin{cases} 2') Z \underline{a*} & \text{non c'è nessuna produzione} \\ & \text{che comincia con } ax \\ 2) Z \underline{A} & \text{c'è una produzione che} \\ & \text{comincia con } A \end{cases}$

Vedremo che usando una struttura ausiliaria (19) (DFA dei prefissi riabinili) ed usando anche simboli di look-ahead, si può costruire un parser (DPDA a partire da una grammatica libera G di classe

- LR(0)
- SLR(1)
- LR(1)
- LALR(1) + tipo di parser prodotto da YACC, uno strumento che genera parser a partire da grammatiche

Problema con produzioni ε del tipo A → ε



riduzione sempre
applicabile!

⇒ cercare di evitare di usare grammatiche libere con produzioni ε quando si vuole usare la tecnica di bottom-up parsing (shift-reduce)