PDA e linguaggi deterministici

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

(1) $\forall q \in Q\ \forall Z \in \Gamma$, se $\delta(q, \varepsilon, Z) \neq \emptyset$ allora $\delta(q, a, Z) = \emptyset \quad \forall a \in \Sigma$

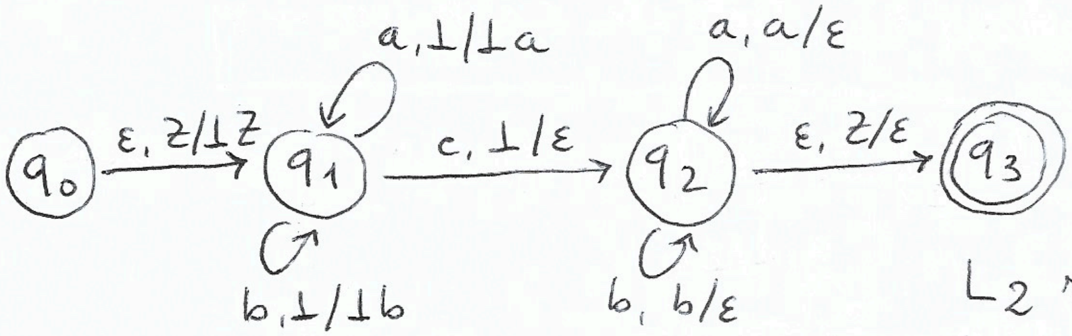
(2) $\forall q \in Q\ \forall Z \in \Gamma\ \forall a \in \Sigma \cup \{\varepsilon\} \quad |\delta(q, a, Z)| \leq 1$

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.

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

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

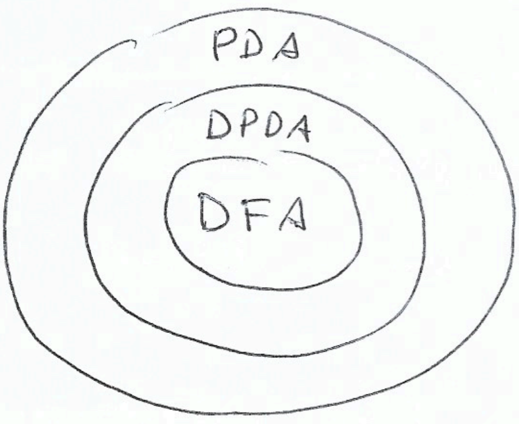


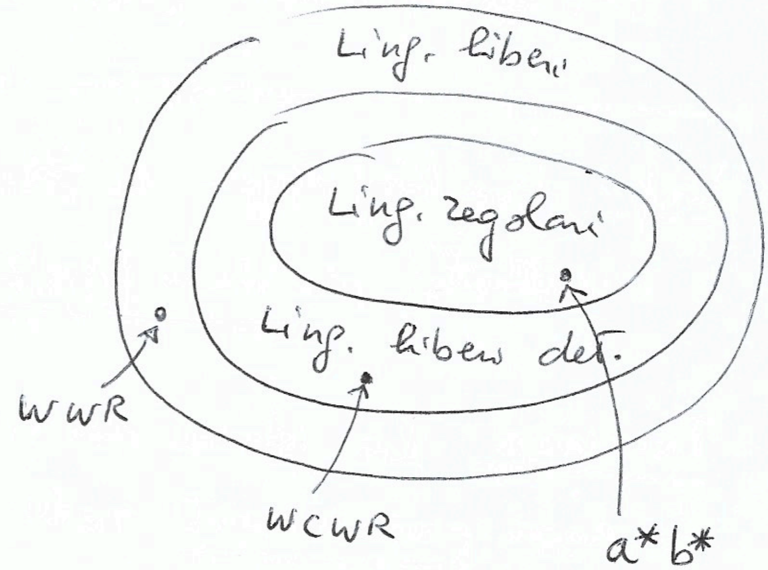
L_2 riconosciuto da un DPDA per stato finale

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

<u>Dim</u>: 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

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

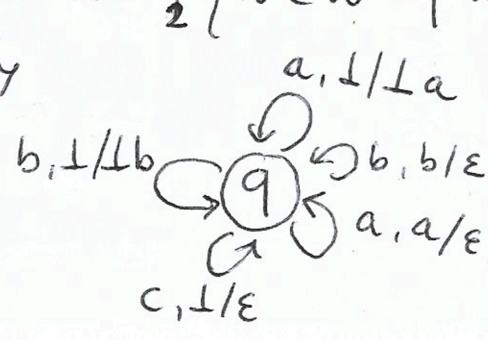




<u>Fatto</u>: Un linguaggio libero deterministico L è riconosciuto da un DPDA per pila vuota se solo se L gode della "prefix property"
(condizione necessaria e sufficiente)

$$\boxed{\nexists\, x, y \in L \text{ tali che } x \text{ è prefisso di } y}$$

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



DPDA che riconosce L_2 per pila vuota

Quindi

(1) Se L liberodeterministico non gode della prefix property, non può essere riconosciuto da un DPDA per pila vuota.

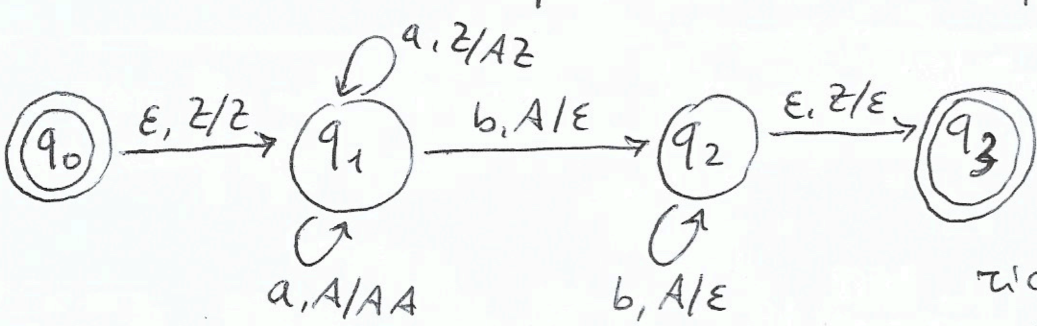
(2) Se L libero det. gode della prefix property, allora può essere riconosciuto da un DPDA per pila vuota

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

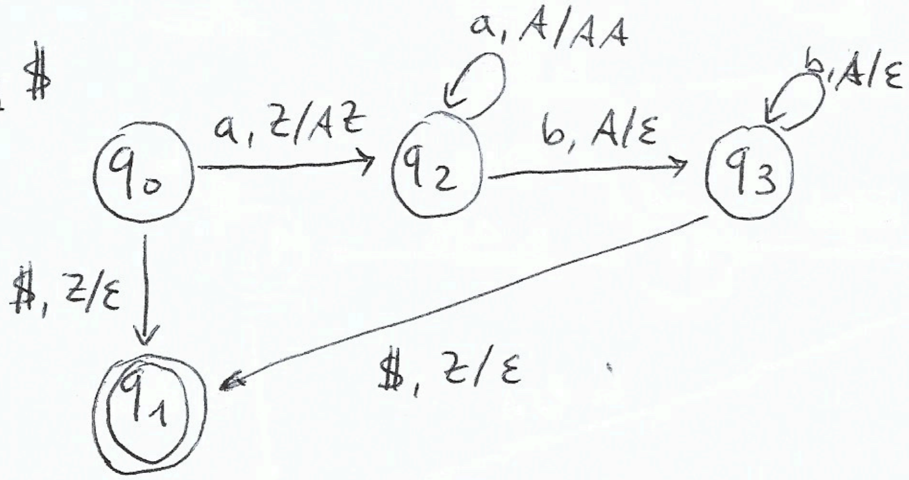
---

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



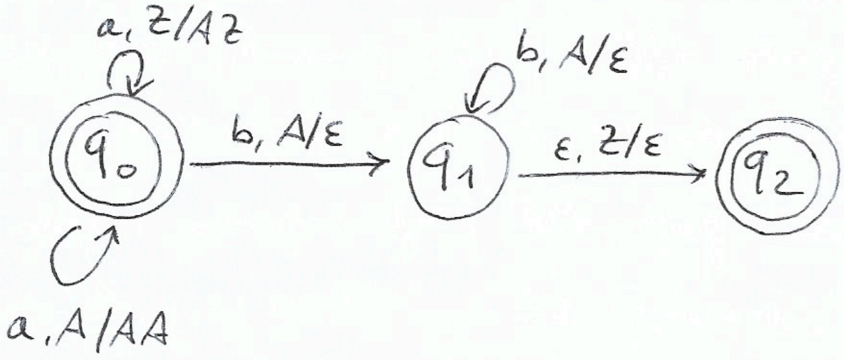
riconosce per stato final

$L_1 \#$

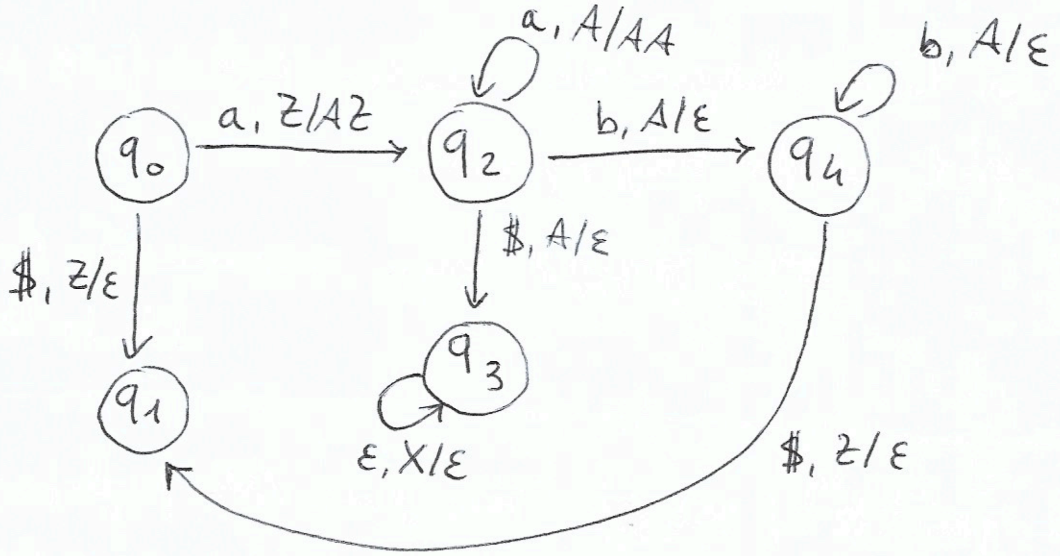


riconosce per pila vuota (ed anche per stato finale)

$L = a^* \cup \{a^n b^n \mid n \geq 1\}$ non gode della prefix property

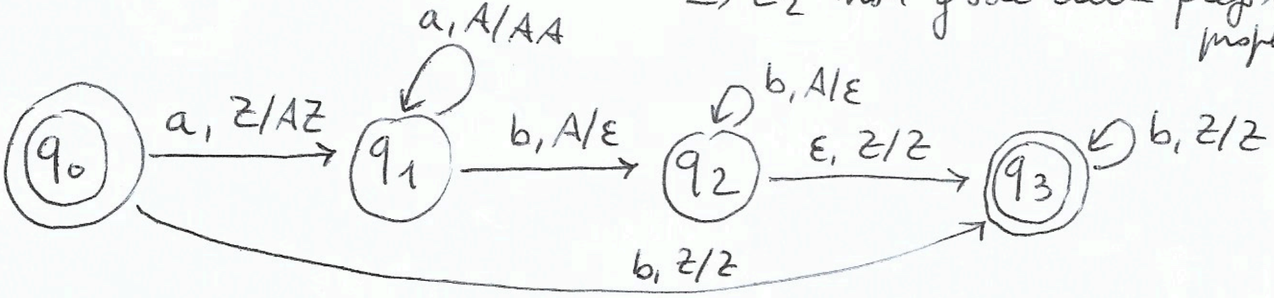


L$
per pila vuota

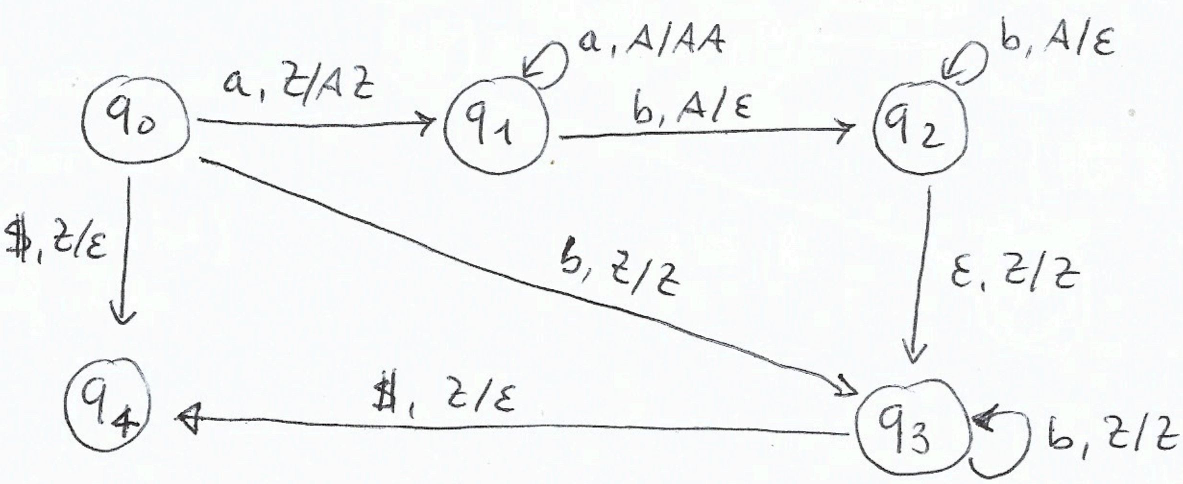


$L_2 = \{a^n b^m \mid n \leq m\}$ ab prefisso di abb

$\Rightarrow L_2$ non gode della prefix property

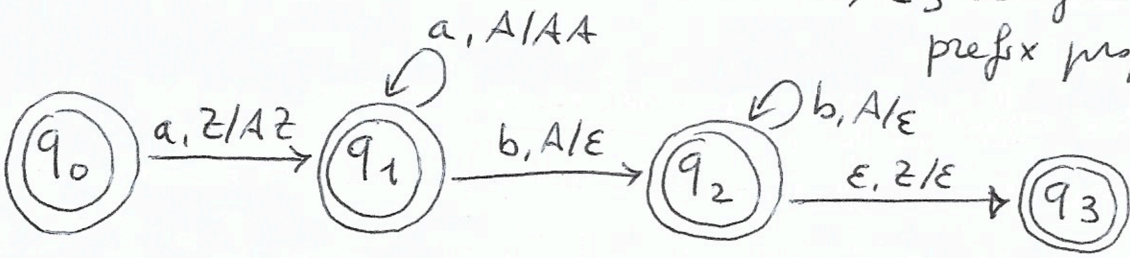


L_2 $

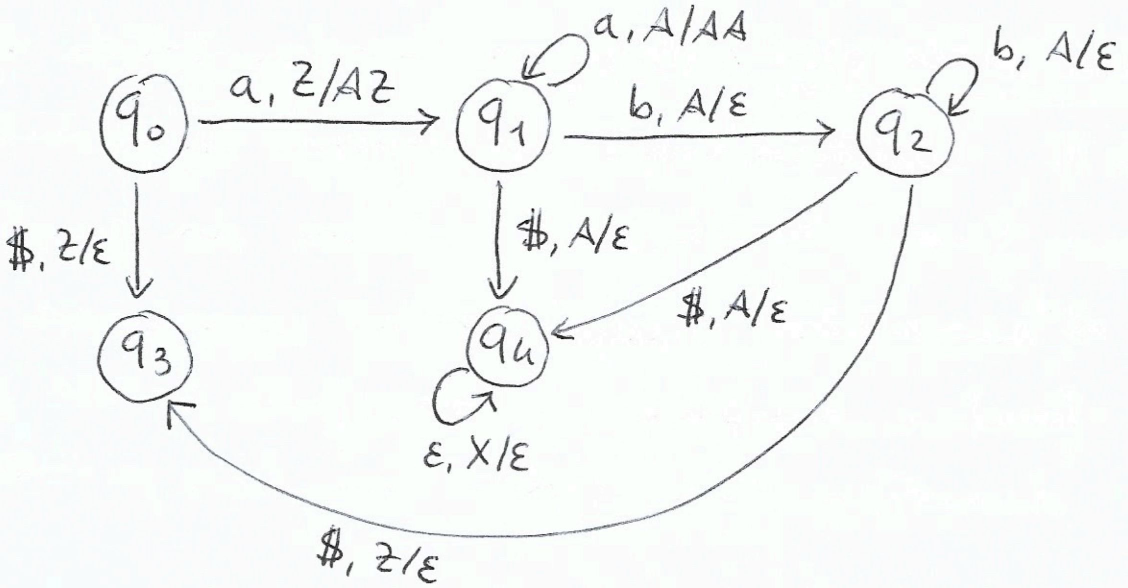


$L_3 = \{a^n b^m \mid n \geq m \geq 0\}$ $\varepsilon \in L_3$

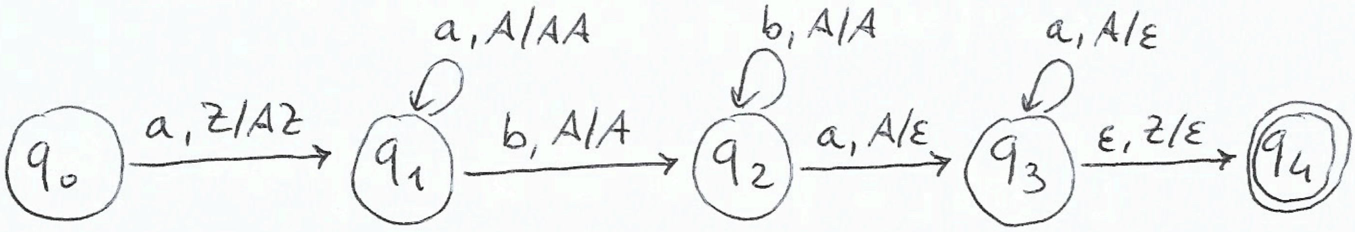
$\Rightarrow L_3$ non gode della prefix property



$L_3\$$ per pila vuota



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



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

<u>NON</u> <u>AMBIGUA</u>

$\Rightarrow$ I ling. liberi deterministici non sono ambigui!!

Proprietà dei ling. liberi deterministici

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

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 nonfinali (come fatto coi DFA per dimostrare che i regolari sono chiusi per intersezione)

- Non sono chiusi per intersezione

$L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ 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)

Grammatica Libera → [Algoritmo di Costruzione] → Parser

(YACC)

Parser: fondamentalmente un DPDA con output

Lista di Token → [Parser] → Albero di derivazione

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 costruita, 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 dall'input per guidare la ricerca della derivazione

I Parser possono essere:

- Top-down

  Ricostruiscono una derivazione leftmost però a partire dal simbolo iniziale S (all'inizio sulla pila)

- Bottom-up

  Ricostruiscono una derivazione rightmost (a rovescio) a partire dalla stringa w, cercando di ridurla al simbolo iniziale S (alla fine sulla pila)

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

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

Introduzione al Top-down Parsing

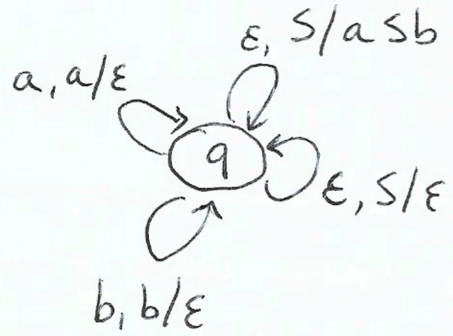
Data $G = (NT, T, S, R)$ libera, costruiamo il PDA $M = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$, che riconosce per pila vuota, dove δ è definita:

- $(q, \beta) \in \delta(q, \varepsilon, A)$ se $A \to \beta \in R$ (<u>espandi</u>)
- $(q, \varepsilon) \in \delta(q, a, a)$ $\forall a \in T$ (<u>consuma</u>)

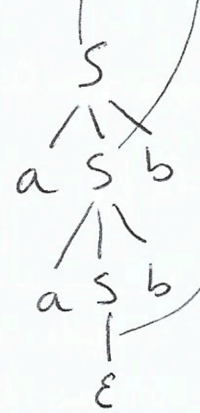
tale che $L(G) = P[M]$

↑ riconoscimento per pila vuota

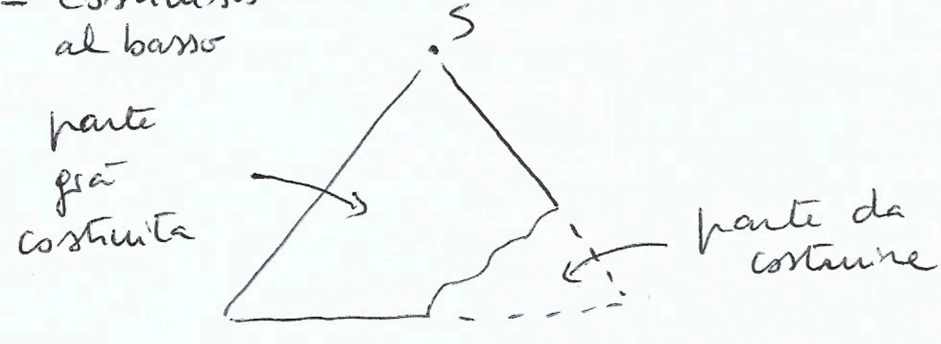
<u>Esempio</u> $S \to aSb \mid \varepsilon$ $\quad S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$



$$
\begin{aligned}
&(q, aabb, S) \vdash (q, aabb, aSb) \\
&\vdash (q, abb, Sb) \vdash (q, abb, aSbb) \\
&\vdash (q, bb, Sbb) \vdash (q, bb, bb) \\
&\vdash (q, b, b) \vdash (q, \varepsilon, \varepsilon)
\end{aligned}
$$



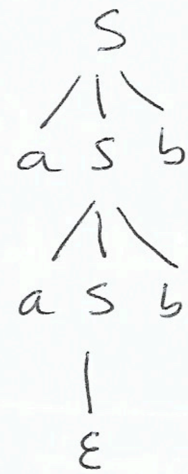
- Derivazione canonica sx (leftmost)
- costruisce l'albero dall'alto al basso



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

  - se leggo "a", $\Rightarrow$ espando $S \to aSb$
  - se leggo "b", $\Rightarrow$ espando $S \to \varepsilon$

input	stack
aabb\$	S
	aSb
abb\$	Sb
	aSbb
bb\$	Sbb
	bb
b\$	b
\$	ε



$G \left[S \to aSb \mid \varepsilon \right.$ Vedremo essere una grammatica di classe LL(1) nel senso che consente di costruire un Parser/DPDA

left-to-right	/ left-derivation	(1)
L	L	↑
modo in cui viene letto l'input	genera una derivazione leftmost	guardando un solo simbolo di look-ahead

$G\,[\, S \to aSb \mid ab$

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

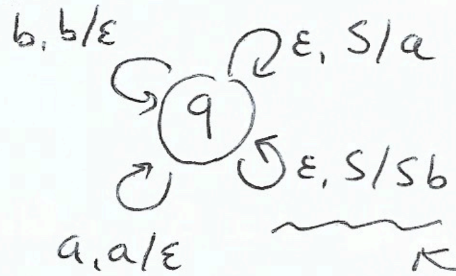
- se leggo "aa", $\Rightarrow$ espando $S \to aSb$
- se leggo "ab", $\Rightarrow$ espando $S \to ab$

G è di classe LL(2)

<u>Attenzione!</u> Non tutte le grammatiche sono adatte per top-down parsing

<u>Esempio 1</u> $G\,[\, S \to Sb \mid a \qquad L(G) = ab^*$

G è <u>left-recursive</u>!



divergenta!

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

- 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 ricorsione sinistra

$$G' \left[\begin{array}{l} S \to aA \\ A \to bA \mid \varepsilon \end{array} \right. \qquad L(G) = L(G')$$

G' è adatta al top-down parsing

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

G' è LL(1)

Esempio 2

$$S \to abB \mid acC$$
$$B \to \dots$$
$$C \to \dots$$

- se leggo "a" e sulla pila c'è S, allora come espando S?
  - o leggo due simboli ($ab \Rightarrow S \to abB$, $ac \Rightarrow S \to acC$)
  - oppure "Fattorizzo"

$$\left. \begin{array}{l} S \to aE \\ E \to bB \mid cC \\ \vdots \end{array} \right]$$

ora non c'è nondet: basta un solo simbolo di look-ahead

Introduzione al bottom-up parsing

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

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

dove

1. $(q, aX) \in \delta(q, a, X)$ $\quad \forall a \in T$, $\forall X \in T \cup NT \cup \{Z\}$ (<u>SHIFT</u>)

2. $(q, A) \in \delta(q, \varepsilon, \underbrace{\alpha^R})$ se $A \to \alpha \in R$ (<u>REDUCE</u>)

3. $(q, \varepsilon) \in \delta(q, \$, SZ)$ (<u>ACCEPT</u>)

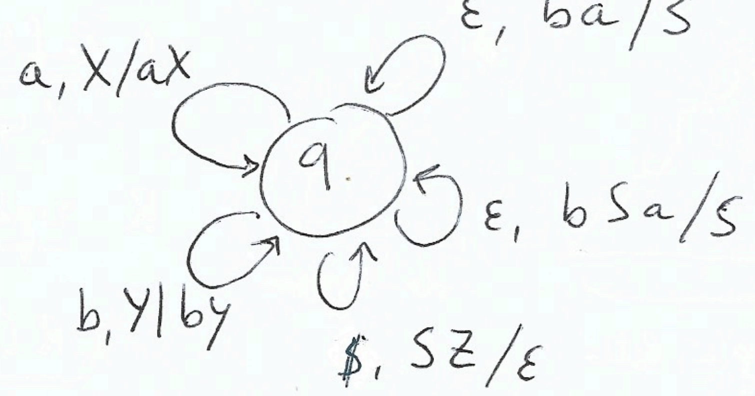
S deve essere alla fine sulla pila

\$ simbolo di fine input

generalizzazione dei PDA in cui si consuma una stringa sulla pila anziché solo il top

$$G \left[\; S \to aSb \mid ab \right.$$

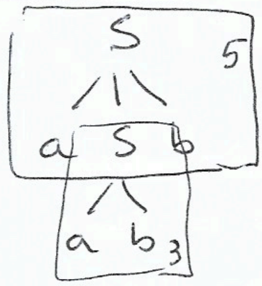
$X \in \{a, Z\}$



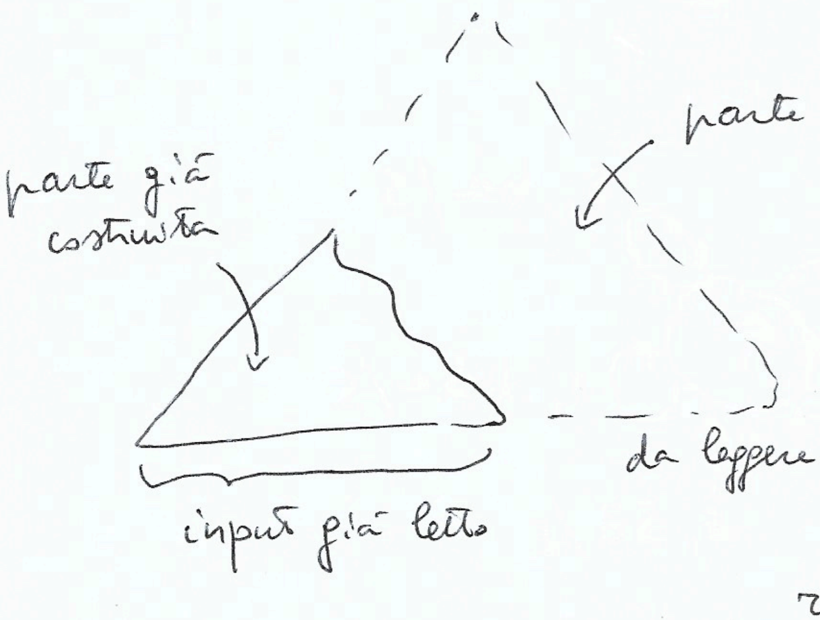
$Y \in \{a, S\}$

	Stack (cresce verso dx)	Input	Azione
0)	Z	aabb\$	shift
1)	Za	abb\$	shift
2)	Zaa	bb\$	shift
3)	Zaab	b\$	reduce $S \to ab$
4)	ZaS	b\$	shift
5)	ZaSb	\$	reduce $S \to aSb$
6)	ZS	\$	Accept
7)	ε	ε	

$$aabb \Leftarrow aSb \Leftarrow S$$



derivazione canonica destra a rovescio



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 !

- <u>conflitti</u> <u>shift - reduce</u>

3) $Z\,aab$ $b\$$ potrei fare <u>shift</u>

4') $Z\,aabb$ $\$$ ma è un percorso infruttuoso

- conflitti <u>reduce - reduce</u> (più reduce possibili)
  (non ci sono in quest'esempio, ma con grammatiche più complesse è possibile)

$\Rightarrow$ 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

$$E \to T+E \mid T$$
$$T \to T*A \mid A$$
$$A \to a \mid b \mid (E)$$

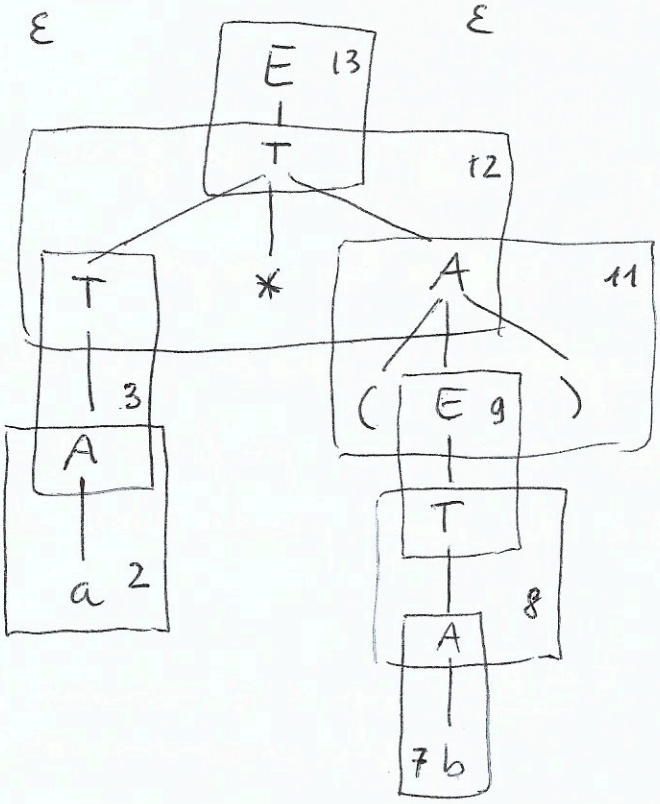
G	M	
	t_0 $\delta(q, a, X) = (q, aX)\ \forall a \forall X$	SHIFT
$(R_1)\ E \to T + E$	t_1 $\delta(q, \varepsilon, E+T) = (q, E)$	REDUCE
$(R_2)\ E \to T$	t_2 $\delta(q, \varepsilon, T) = (q, E)$	REDUCE
$(R_3)\ T \to T * A$	t_3 $\delta(q, \varepsilon, A * T) = (q, T)$	REDUCE
$(R_4)\ T \to A$	t_4 $\delta(q, \varepsilon, A) = (q, T)$	REDUCE
$(R_5)\ A \to (E)$	t_5 $\delta(q, \varepsilon,)E() = (q, A)$	REDUCE
$(R_6)\ A \to a$	t_6 $\delta(q, \varepsilon, a) = (q, A)$	REDUCE
$(R_7)\ A \to b$	t_7 $\delta(q, \varepsilon, b) = (q, A)$	REDUCE
	t_8 $\delta(q, \$, EZ) = (q, \varepsilon)$	ACCEPT

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

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

derivazione canonica dx a rovescio
(rightmost)

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



costruzione dell'albero <u>bottom-up</u>!

L'automa è nondeterministico perché

(1) chi ha precedenza tra <u>shift</u> e <u>reduce</u>?

<u>Ad es</u>:

1) Za   * (b)$ shift

2') Za*   (b)$ non buono! qui <u>reduce</u> è meglio di <u>shift</u>!

<u>Altro es</u>:

3) ZT   * (b)$ reduce R_2

4') ZE   * (b)$ non buono! qui <u>shift</u> è meglio di <u>reduce</u>!

(2) Chi ha precedenza tra <u>2</u> diverse <u>reduce</u>?

<u>Ad es</u>:

(11) ZT*A $ reduce R_3

(12) ZT

(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.!!

<u>Ad es</u>

2') Za<u>*</u>   non c'è nessuna produzione che comincia con a*

scelgo 2) $\Leftarrow$

2) Z<u>A</u>   c'è una produzione che comincia con A

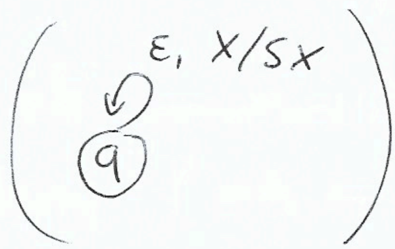
Vedremo che usando una struttura ausiliaria (DFA dei prefissi viabili) ed usando anche simboli di look-ahead, si può costruire un parser (DPDA a partire da una grammatica libera G di classe

- 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 $A \to \varepsilon$

$S \to \varepsilon$



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)