

(1)

Parser Bottom-up

(o Shift-Reduce)

- Shift: un simbolo terminale viene spostato dall'input sulla pila
- reduce: una serie di simboli (terminali e non terminali) sulla cima della pila corrisponde al "reverse" di una parte destra di una produzione
 $A \rightarrow \alpha \in R \quad - \quad \alpha^R \text{ sulla pila}$
 La stringa α^R viene rimossa dalla pila e sostituita con A
 (" α viene ridotta ad A^n ")

Riprendiamo la presentazione di un parser shift-reduce nondeterministico, che è essenzialmente un PDA con un solo stato che riconosce la pila vuota il language $L \cdot \$$

Parser LR :

- L (leggo da sx a dx)
- R (derivazione rightmost)

Parser shift-reduce Nondeterministico

(2)

- Input: - una grammatica libera G con simboli interni S
- una stringa $w \in T^*$

Output: se $w \in L(G)$, allora restituisce la sua derivazione rightmost a rovescio

- Inizializziamo la pila a $\$$;
- Inizializziamo l'input con $w\$$;
- Usiamo il PDA seguente per trovare la derivazione per $w\$$

$$M = (T, \{q\}, \underbrace{T \cup T\{ \$ \}}_{\Gamma}, \delta, \$, \phi)$$

dove

$$1. (q, aX) \in \delta(q, a, X) \forall a \in T \forall X \in \Gamma \quad (\text{SHIFT})$$

$$2. (q, A) \in \delta(q, \epsilon, \underline{\alpha^R}) \text{ se } A \rightarrow \alpha \in R \quad (\text{REDUCE})$$

$$3. (q, \epsilon) \in \delta(q, \$, S\$) \quad (\text{Accept})$$

- ogni volta che facciamo "Reduce", forniamo in output la produzione usata

- alla fine, $S\$$ sulla pila, con $\$$ in input.
 \Rightarrow ok, accettiamo

Oss: generalizzazione

della def. di PDA dove non si consuma solo il top della pila, ma una serie di caratteri contiguous a cominciare dal top

(3)

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

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

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

<u>Pila</u>	<u>Input</u>	<u>Azione</u>	<u>Output</u>
1. \$	a+b*b\$	shift	
2. \$a	+b*b\$	reduce	$A \rightarrow a$
3. \$A	+b*b\$	reduce	$T \rightarrow A$
4. \$T	+b*b\$	shift	
5. \$T+	b*b\$	shift	
6. \$T+b	*b\$	reduce	$A \rightarrow b$
7. \$T+A	*b\$	shift	
8. \$T+A*	b\$	shift	
9. \$T+A*b	\$	reduce	$A \rightarrow b$
10. \$T+A*A	\$	reduce	$T \rightarrow A$
11. \$T+A*T	\$	reduce	$T \rightarrow A*T$
12. \$T+T	\$	reduce	$E \rightarrow T$
13. \$T+E	\$	reduce	$E \rightarrow T+E$
14. \$E	\$	<u>accept</u>	
15. \$\epsilon\$	\$		

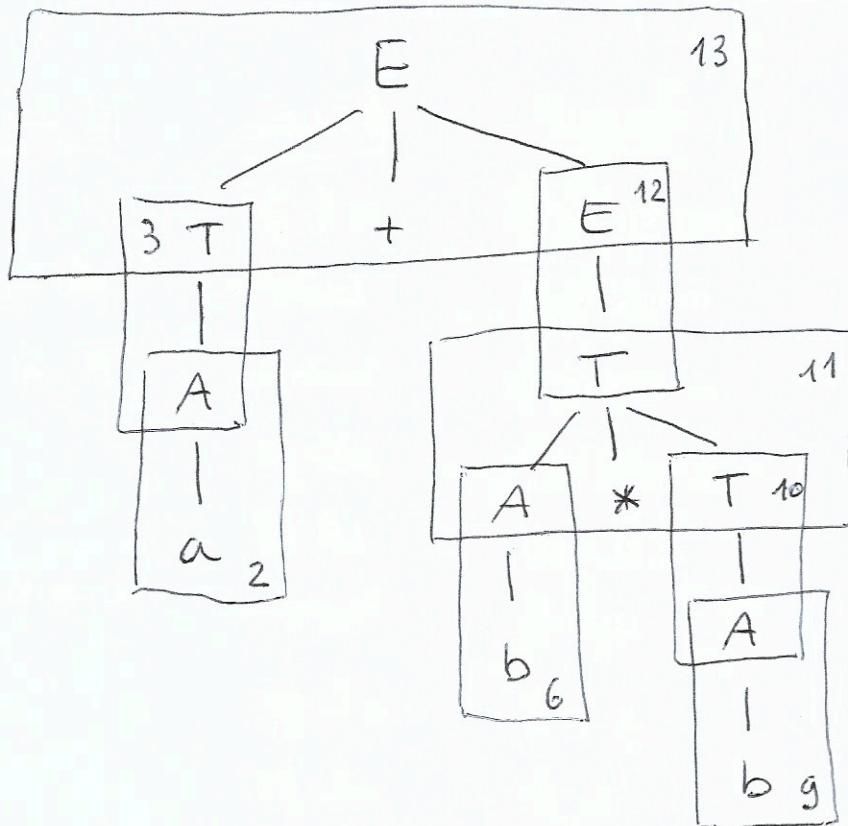


facevamo crescere la pila verso destra, così leggo
 α (da sinistra a destra), invece di α^R (da destra a
 sinistra) sulla pila per $A \rightarrow \alpha$

(4)

$$E \Rightarrow T + E \Rightarrow T + T \Rightarrow T + A * T \Rightarrow T + A * A$$

$$\Rightarrow T + A * b \Rightarrow T + b * b \Rightarrow A + b * b \Rightarrow a + b * b$$



- costruzione dell'albero di derivazione bottom-up
- derivazione canonica destra a rovescio
- enorme nondeterminismo:

- confitti shift-reduce

2') \$ a + b * b \$ shift

3') \$ a + b * b \$

- confitti reduce-reduce

11') \$ T + A * T \$ reduce $E \rightarrow T$

12') \$ T + A * E \$

Come risolvere i conflitti?

Strategia: bisogna scegliere l'azione giusta
in modo che sulla pila ci sia un
prefisso viabile

Prefisso viabile: è una sequenza $\in (T \cup NT)^*$
che può apparire sulla pila di un parser
bottom-up per una configurazione che accetta
un input

In pratica... bisogna che la parte top della
pila sia un prefisso di una parte dx di una
produzione

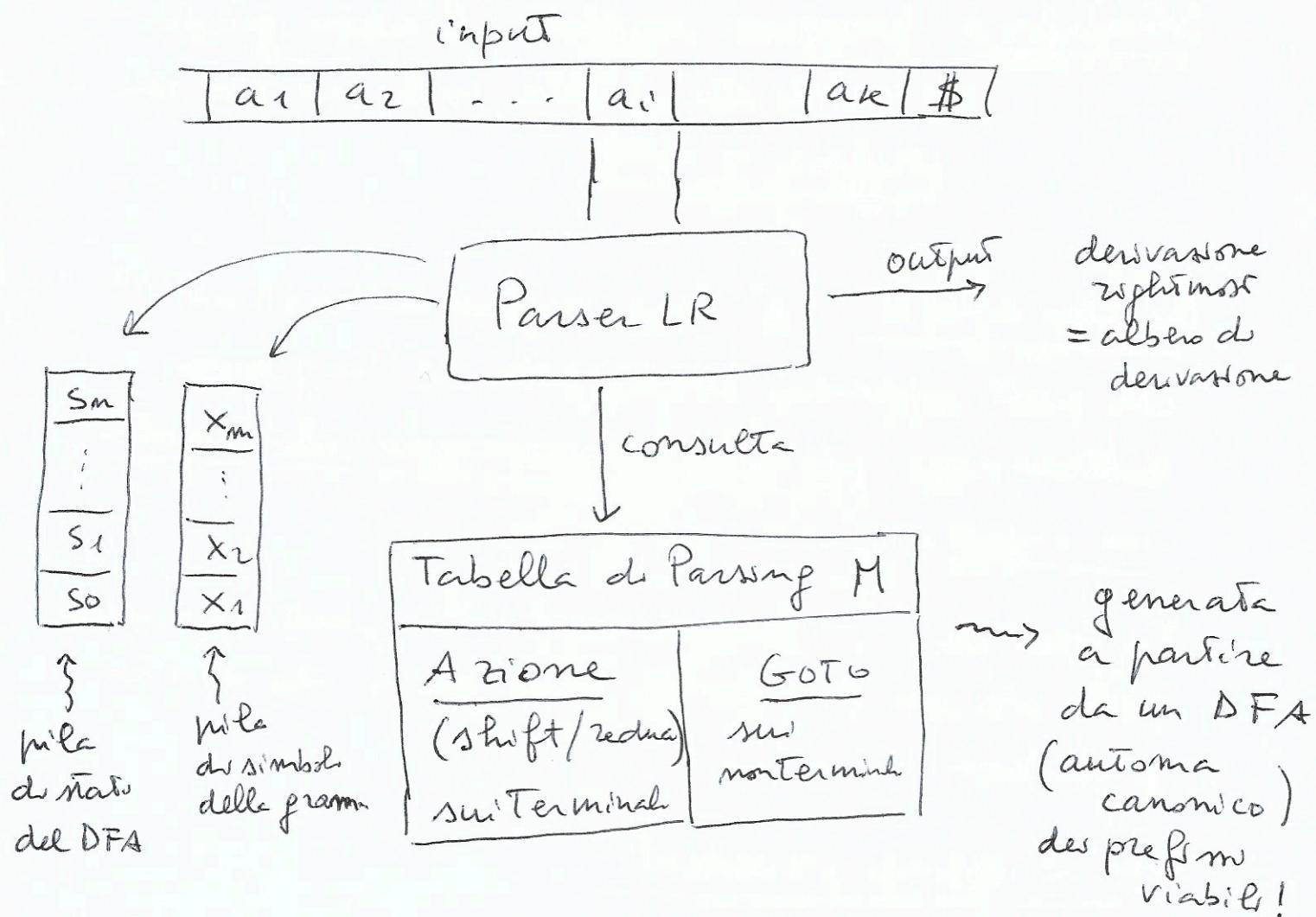
Nei 2 esempi precedenti:

- $\$ a +$ non è un prefisso di parte dx
di una produzione, mentre
 $\$ A$ lo è!

- $\$ T + A * E$ non è prefisso di una parte dx
di una produzione, mentre
 $\$ T + T$ lo è!

Quindi dovremo fornire al PDA, per renderlo deterministico, una struttura di controllo (tabelle di parsing) che ci aiuti a scegliere l'azione giusta.

Come è fatto un Parser LR ?



Una configurazione di un parser LR è

$$(S_0 \dots S_m, X_1 \dots X_m, a_i \dots a_k \$)$$

Stack degli stati Stack dei simboli Resto dell'input

Oss: " $X_1 \dots X_m a_i \dots a_k$ " è una stringa intermedia della derivazione canonica destra

Mosse del Parser LR

(7)

- (1) Prima legge: [stato top] e [simbolo corrente dell'input]
 s_n a_i

- (2) Consulta la tabella di parsing LR $M[s_n, a_i]$

- se $M[s_n, a_i] = \underline{\text{shifts}}$, allora la nuova configurazione è

$$(s_0 \dots s_n \underline{s}, x_1 \dots x_m \underline{a_i}, a_{i+1} \dots a_k \#)$$

- se $M[s_n, a_i] = \underline{\text{reduce } A \rightarrow \beta}$, allora la nuova configurazione è

$$(s_0 \dots s_{n-r} \underline{s}, x_1 \dots x_{m-r} A, a_i \dots a_k \#)$$

dove $r = |\beta|$ e $M[s_{n-r}, A] = \underline{\text{gotos}}$

cioè fa tre passi:

(1) faccio "pop" di r elementi dal stack

(2) metto A in cima alla pila dei simboli

(3) calcolo il nuovo stato top, guardando

$M[s_{n-r}, A] = \text{goto } S$ e metto S in cima alla pila degli stati.

- se $M[s_n, a_i] = \text{accept} \Rightarrow \text{FINE!}$

- se $M[s_n, a_i] = \text{"biancos"} \Rightarrow \text{errore!}$

Un esempio

G

(1) $S' \rightarrow S$
(2) $S \rightarrow (S)$
(3) $S \rightarrow ()$



grammatica aumentata con
un nuovo simbolo invalide!
(assunzione che faremo sempre
per parser LR!)

generata a partire
dal DFA
dei prefissi
viablei

→ Tabella di Parsing LR(0) non usa look-ahead!

State	Actions			GOTO
	()	\$	
0	S2			g1
1			Accept	
2	S2	S5		g3
3		S4		
4	r2	r2	r2	
5	r3	r3	r3	

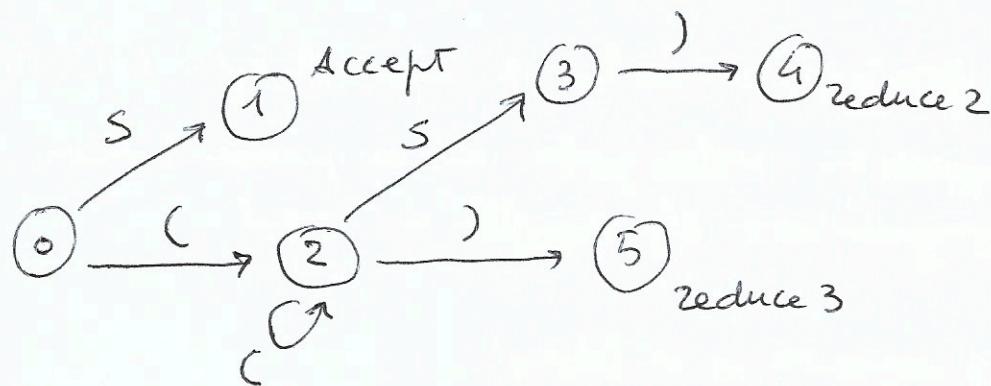
s2 = shift
state2

r3 = reduce
 $S \rightarrow ()$

g1 = goto 1

Stack state: (0, ϵ , (()) \$) stack symbol:	input	Action	Output
		$\frac{S2}{S2}$	-
(02, (, (()) \$)		S2	-
(022, ((,)) \$)		S5	-
(0225, ((),) \$)		r3	$S \rightarrow ()$ \uparrow
(023, (S,) \$)		S4	(S) \uparrow
(0234, (S), \$)		r2	$S \rightarrow (S)$ \uparrow
(01, S, \$)		Accept	S

DFA dei prefissi vivibili / Automa canonico LR(0) (9)



- da questo DFA (che non è esattamente un DFA!)
 - si ricava la tabella di parsing LR(0)
- Ma come si ricava questo DFA a partire dalla grammatica? (Lo vedremo tra poco)

Le operazioni shift/reduce devono far sì che rimanga in cima alla pila un prefisso di una parte dx di una produzione!

Nel suo complesso, la pila deve contenere un prefisso di derivazione canonica dx!

Nell'esempio:

- appena sulla pila compare ")", allora dobbiamo fare una reduce
Quale reduce fare, dipende dalla "stato", ovvero dallo stato su cui siamo finiti nel DFA

Prefisso Viable

(10)

Def(1) Un prefisso viable è una stringa $\gamma \in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom-up per una configurazione che accetta un input.

Def(2) (su grammatica G libera)

Una stringa $\gamma \in (T \cup NT)^*$ è un prefisso viable per G se esiste una derivazione rightmost

$$S \Rightarrow^* S A y \Rightarrow S \alpha \beta y = \gamma \beta y$$

per qualche $y \in T^*$, $\delta \in (T \cup NT)^*$ e per una produzione $A \rightarrow \alpha \beta$. Inoltre S è un prefisso viable per definizione.
Un prefisso viable è completo se $\beta = \epsilon$; in tal caso α è detta maniglia (Handle) per γy ,

(ovvero in cosa alla pila trovo α^R e posso fare una reduce)

Come fare a:

- (i) riconoscere le maniglie sulla cima della pila e ridurle?
- (ii) scegliere, tra più riduzioni, solo quelle che producono sulla pila un nuovo prefisso viable?
- (iii) scegliere spontaneamente che completino i prefissi viablei sulla pila e facciano comparire una nuova maniglia?

Teorema Data G libera, i prefissi viabili di G costituiscono un lang. regolare e può essere descritto con un DFA.

\Rightarrow Il parser (cioè un PDA) può consultare il DFA dei prefissi viabili (ovvero la tabella di parsing) per decidere cosa fare

- se la pila contiene un prefisso viabile completo, allora il parser produce;
- se la pila contiene un prefisso viabile incompleto, allora il parser shifts;
- se la pila non contiene un prefisso viabile, allora errore.

A seconda di come è fatto il DFA, il parser può risultare deterministico o meno, (eventualmente sfruttando informazioni di look-ahead e sui follow dei nonterminal per ottenere determinismo)

Anche se, concettualmente, ad ogni modifica (12) della pila il DFA la riscandisce da capo per determinare come sia fatto il prefisso viabile corrente, questo non è necessario perché ogni prefisso di un prefisso viabile è un prefisso viabile!

La pila viene modificata in 2 modi:

1) Shift: la pila passa da $\$j$ a $\$jx$. In tal caso il DFA si trova nello stato s dopo aver elaborato $\$j$ \Rightarrow basta far ripartire il DFA da s con input x

2) Reduce $A \rightarrow \alpha$: la pila passa da $\$j\alpha$ a $\$jA$.
In tal caso il DFA si trova nello stato s dopo aver elaborato $\$j\alpha$; non c'è bisogno di far ripartire il DFA dalla base delle pile; basta ripristinare lo stato in cui si trovava subito prima di elaborare il primo simbolo di α e fornirgli il simbolo A in input.

\Rightarrow mi serve lo stack degli stati del DFA!
(anzi, mi basterebbe solo lo stack degli stati, ma noi useremo anche lo stack dei simboli per ragioni di didattica)

(13)

Uno stato del DFA dei prefissi viabili
(chiamato automa canonico LR(0)) è costituito
da un insieme di item LR(0)

ITEM LR(0) : è una produzione con indicata,
con un punto, una posizione delle
sua parte destra

Ese: $A \rightarrow X Y Z$ genera 4 item

$$A \rightarrow . X Y Z$$

$$A \rightarrow X . Y Z$$

$$A \rightarrow X Y . Z$$

$$A \rightarrow X Y Z .$$

Il punto ":" indica quanta parte della produzione
è già stata analizzata

- se $A \rightarrow \alpha . \beta$ è nello stato del DFA in cima
alla pile, allora vuol dire che α è sulla pile
dei simboli e che ci si aspetta che l'input da
leggere contenga (o possa venir letto) β
- se $A \rightarrow \alpha .$ è nello stato del DFA in cima
alla pile, allora sulla pile dei simboli c'è
la maniglia α e possiamo fare la reduce

Come costruire l'NFA dei prefissi viabili? (14)

Data $G = (NT, T, S, R)$ libera, prendiamo la grammatica aumentata con un nuovo simbolo iniziale S' ed una produzione $S' \rightarrow S$

L'NFA dei prefissi viabili di G (primo passo verso la costruzione del DFA!) si ottiene così:

- $[S' \rightarrow .S]$ è lo stato iniziale
- dallo stato $[A \rightarrow \alpha . X \beta]$ c'è una transizione allo stato $[A \rightarrow \alpha X, \beta]$ etichettata X , per $X \in T \cup NT$
- dallo stato $[A \rightarrow \alpha . X \beta]$, per $X \in NT$ e per ogni produzione $X \rightarrow \gamma$, c'è una ϵ -transizione verso lo stato $[X \rightarrow .\gamma]$

OSS: non serve definire degli stati finali, perché l'NFA serve solo come ausilio al parser

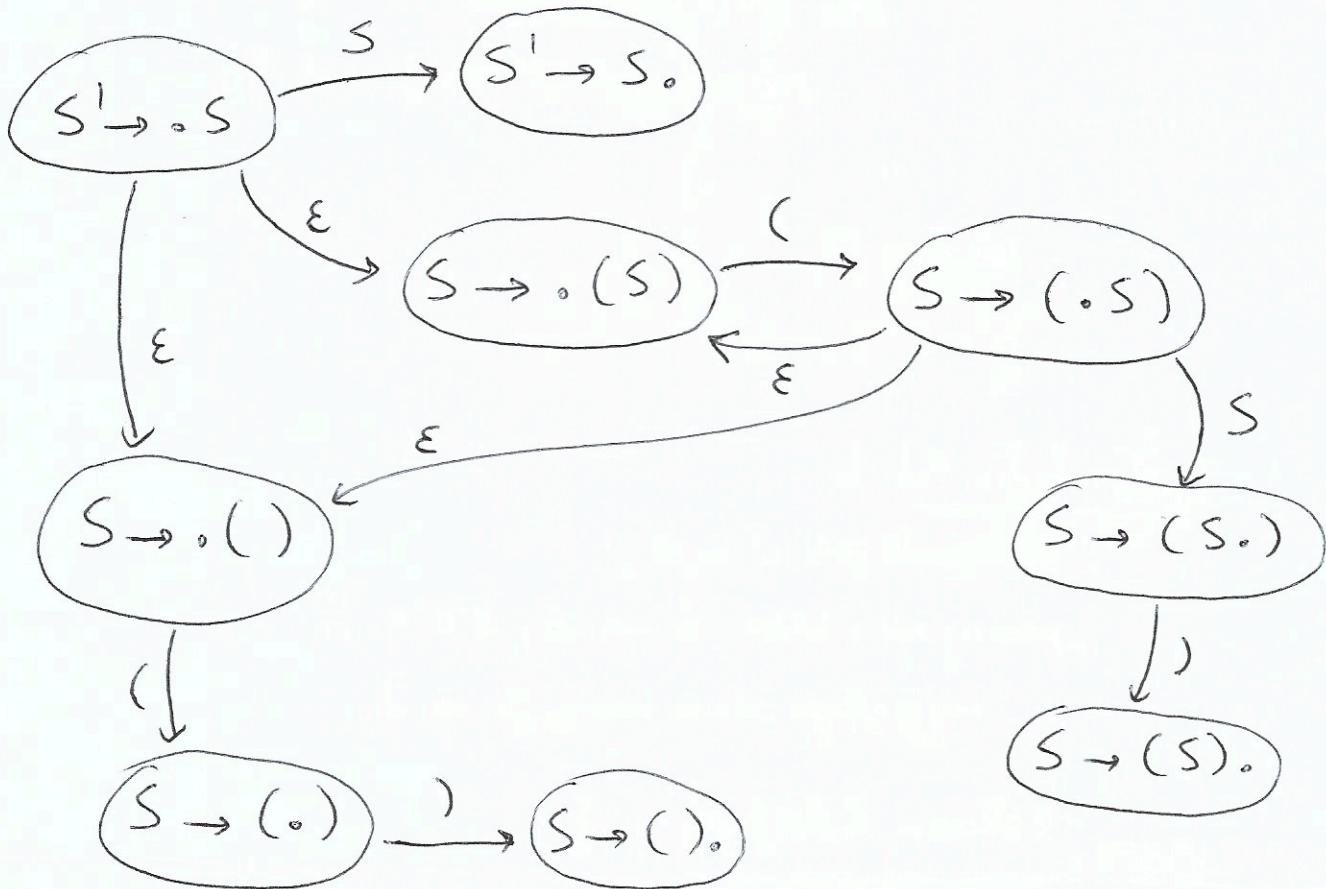
Esempio

(15)

- (1) $S' \rightarrow S$
- (2) $S \rightarrow (S)$
- (3) $S \rightarrow ()$

$$G \quad \left[\begin{array}{l} S' \rightarrow .S | S. \\ S \rightarrow .(S) | (.S) | (S.) | (S). \\ S \rightarrow .() | (.) | (). \end{array} \right]$$

possibili item LR(0) per G



Automa Canonico LR(0)

- 1) È il DFA che si ottiene dall'NFA dei prefissi viabili mediante la costruzione per sottoinsiemi oppure
- 2) in modo diretto usando le funzioni ausiliarie $Clos(I)$ e $Goto(I, X)$

dove I è un insieme di item e $X \in T \cup NT$

Costruzione diretta dell'Automa Canonico LR(0)

(16)

Class(I) {

 ripeti finché I è modificato {

 per ogni item $A \rightarrow \alpha \cdot X \beta \in I$

 per ogni produzione $X \rightarrow \gamma$

 aggiungi $X \rightarrow \cdot \gamma$ ad I;

}

 return I;

}

Goto(I, X) {

 inizializza J = \emptyset ;

 per ogni item $A \rightarrow \alpha \cdot X \beta \in I$

 aggiungi $A \rightarrow \alpha \cdot X \cdot \beta$ a J;

 return Class(J); }

Inizializza $S' = \{ \text{Class}(\{ S' \rightarrow \cdot S \}) \}$;

Inizializza $\delta = \emptyset$;

Ripeti finché S' o δ vengono modificati:

 per ogni $I \in S'$

 per ogni item $A \rightarrow \alpha \cdot X \beta \in I$ {

 sia $J = \text{Goto}(I, X)$;

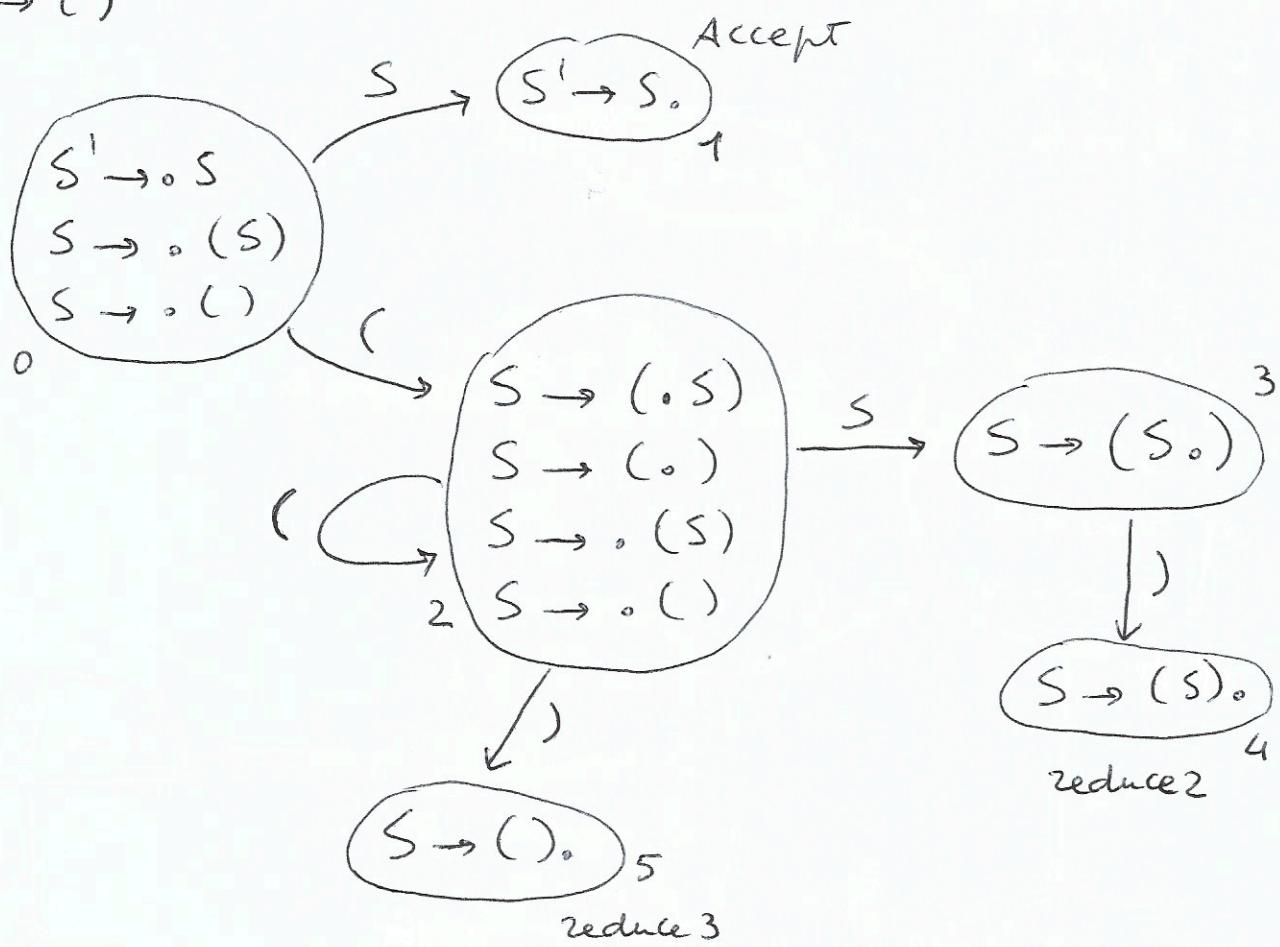
 aggiungi J a S' ;

 aggiungi $\delta(I, X) = J$ a δ ;

}

 return S' e δ .

Automa Canonic LR(0)

↑ $S' \rightarrow S$ 2 $S \rightarrow (S)$ 3 $S \rightarrow ()$ 

$$\text{Clos}(\{S' \rightarrow .S\}) = \underbrace{\{S' \rightarrow .S, S \rightarrow .(S), S \rightarrow .()\}}_{I_0}$$

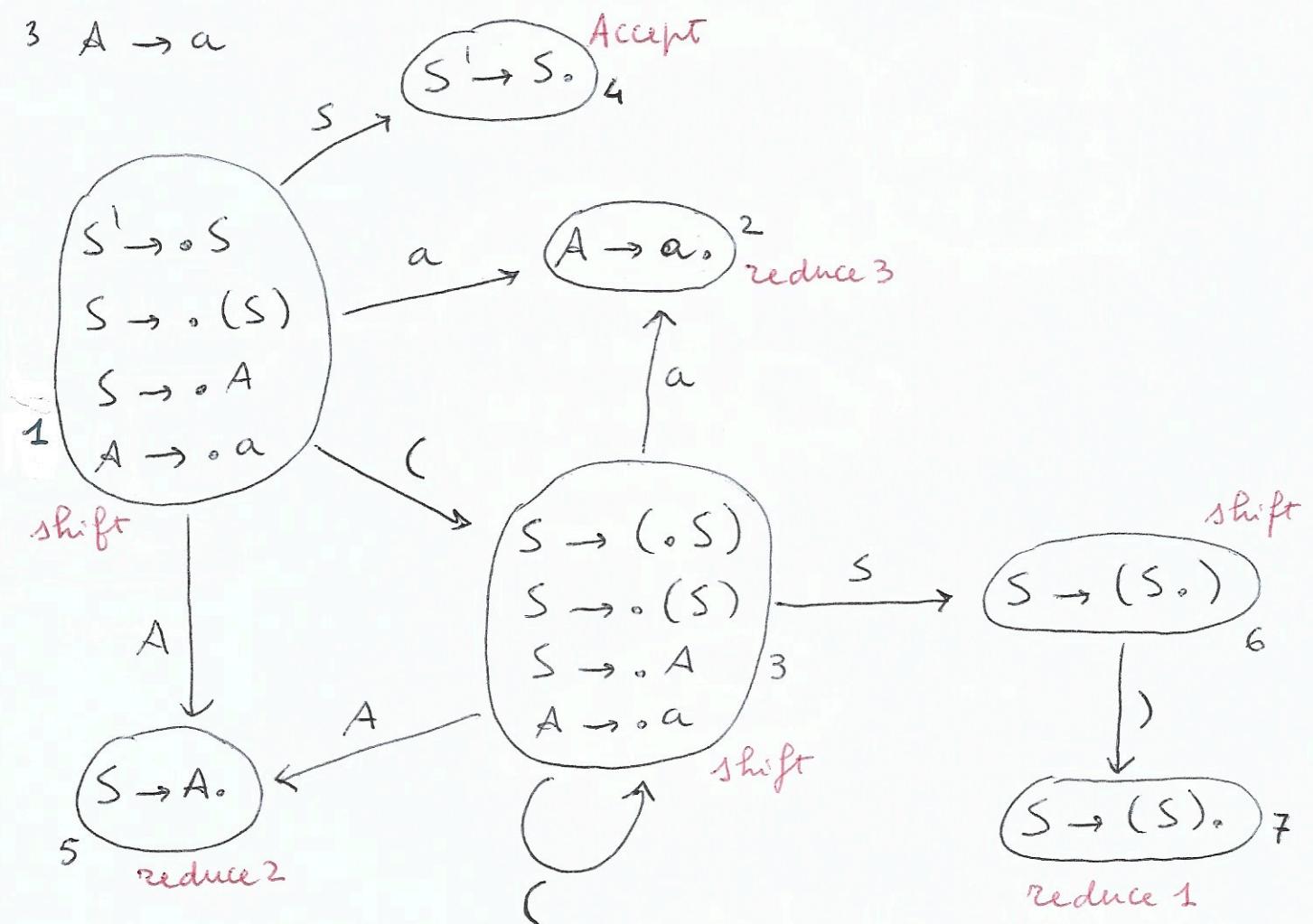
$$\text{Goto}(I_0, () = \text{Clos}(\{S \rightarrow (.S), S \rightarrow (.)\})$$

$$= \underbrace{\{S \rightarrow (.S), S \rightarrow (.), S \rightarrow .(S), S \rightarrow .()\}}_{I_2}$$

Esempio

- 0 $S' \rightarrow S$
- 1 $S \rightarrow (S)$
- 2 $S \rightarrow A$
- 3 $A \rightarrow a$

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



In questo caso (ed è così sempre per grammatiche LR(0)), possiamo associare una azione ad ogni stato

- stato di shift, come 1, 3 e 6
- stato di reduce, come 2, 5, 7
- stato di accept, come 4

Ma non sarà sempre così ---

Tabella di Parsing LR

Matrice bidimensionale M

- righe = stati dell'automa canonico $LR(0)$ / $LR(1)$
- colonne = $\overbrace{TV \{ \$ \}}^{\text{Azione}}$ $\underbrace{UNT}_{\text{GOTO}}$
- $M[s, X]$ contiene le azioni che può compiere un parser LR con s in cima alla pila degli stati e X simbolo in input (o non terminale)
- Se $M[s, X]$ è "bianca" / vuota, allora ERRORE
- Se $M[s, X]$ contiene più azioni, allora CONFLITTO
(il parser non è deterministico)

Caso $LR(0)$

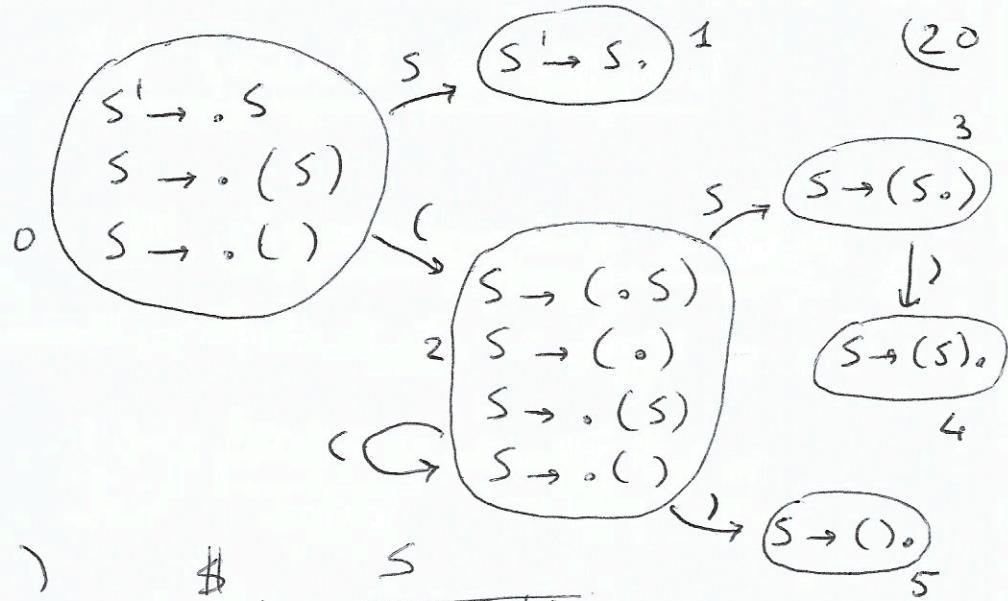
Per ogni stato s dell'automa canonico $LR(0)$

- se $x \in T$ e $s \xrightarrow{x} t$ nell'automa $LR(0)$, inserisci shift t in $M[s, x]$
- se $A \xrightarrow{\alpha} s$, inserisci reduce $A \xrightarrow{\alpha}$ in $M[s, x]$ per tutti gli $x \in TV \{ \$ \}$
- se $S' \xrightarrow{} S_0 \in S$, inserisci Accept in $M[s, \$]$
- se $A \in NT$ e $s \xrightarrow{A} t$ nell'automa $LR(0)$, inserisci GOTO t in $M[s, A]$

Def Una grammatica è di classe $LR(0)$ se ogni casella nelle tabelle di parsing $LR(0)$ contiene al più un elemento!

$$\begin{array}{l} 1 \ S' \rightarrow S \\ 2 \ S \rightarrow (S) \\ 3 \ S \rightarrow () \end{array} \quad G$$

(Esempio da pagina 8)



	()	\$	S	
0	S2			g1
1			acc	
2	S2	S5		g3
3		S4		
4	r2	r2	r2	
5	r3	r3	r3	

Non ci sono conflitti
 $\Rightarrow G \in LR(0)$

(0, ϵ , (()) \$)

(02, (, ()) \$)

(022, ((,)) \$)

(0225, ((),) \$) reduce $S \rightarrow ()$
 $\underbrace{\qquad}_{\text{goto 3}} \xrightarrow{S}$

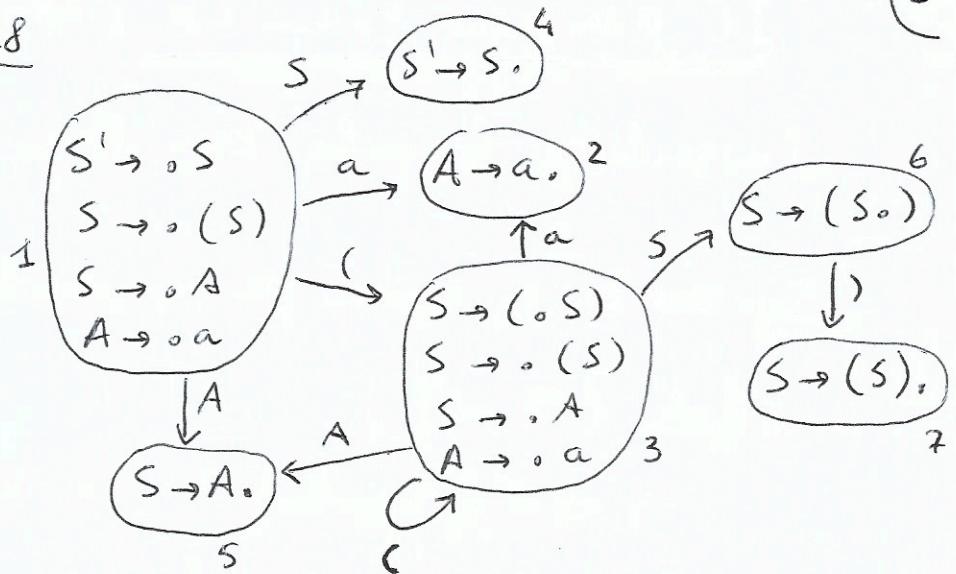
(023, (S, ,) \$)

(0234, (S), \$) reduce $S \rightarrow (S)$
 $\underbrace{\qquad}_{\text{goto 1}} \xrightarrow{S}$

(01, S, \$) accept

Esempio di fg 18

- o $S' \rightarrow S$
- 1 $S \rightarrow (S)$
- 2 $S \rightarrow A$
- 3 $A \rightarrow a$



	a	()	\$	S	A
1	S2	S3			g4	g5
2	r3	r3	r3	r3		
3	S2	S3			g6	g5
4				acc		
5	r2	r2	r2	r2		
6				S7		
7	r1	r1	r1	r1	r1	

Non ci sono conflitti
 $\Rightarrow G$ è della classe LR(0)

- $(1, \epsilon, ((a))\#)$ $(14, S, \$)$
Accept
 $(13, (, (a))\#)$
 $(133, ((, a))\#)$
 $(1332, ((a,))\#)$ reduce $A \rightarrow a$
 $\xrightarrow{\text{goto 5}} A$
 $(1335, ((A,))\#)$ reduce $S \rightarrow A$
 $\xrightarrow{\text{goto 6}} S$
 $(1336, ((S,))\#)$
 $(13367, ((S),))\#)$ reduce $S \rightarrow (S)$
 $\xrightarrow{\text{goto 6}} S$
 $(136, (S,))\#)$
 $(1367, (S), \$)$ reduce $S \rightarrow (S)$

Il generico Parser LR (con solo lo stack degli stati) (22)

- Inizializza la pila con \$ so; % cima della pila è α
so stato iniziale dell'aut.
- Inizializza i_c con il primo carattere in input;
- while (true) {
 - $s = \text{top}(\text{pila});$ % top non rimuove la Testa
 - case $M[s, i_c]$ of
 - shift $t :$ { push t sulla pila;
avanza i_c sull'input; }
 - accept : { output ('accept'); break; }
 - reduce $A \rightarrow \alpha$: { pop $|A|$ stati dalla pila;
 $s_1 = \text{Top}(\text{pila});$ % s_1 contiene $B \rightarrow \gamma.A\delta$
sia $s_2 = M[s_1, A]$ % $M[s_1, A]$ è goto s_2
push s_2 sulla pila;
output la produzione $A \rightarrow \alpha$;
}
 - else error(); % casella bianca