

Grammatiche LR(k)

(45)

- item $LR(k) = [item\ LR(0), \beta]$ con $|\beta| \leq k$
- item iniziale = $[S^i \rightarrow . S, \$]$
- Quando $[A \rightarrow \alpha_0 X \gamma, \beta] \in S$ (stato dell'automa canonico $LR(k)$), allora pure $[X \rightarrow . \delta, w] \in S$ se $X \rightarrow \delta$ è una produzione e $w \in \text{First}_k(\gamma \beta)$
- Tabella con colonne su $T \cup \$$ (per shift e accept), più tutti i w tali che esiste un item $[A \rightarrow \alpha_0, w]$ (per le reduce)
- Affinché G sia $LR(k)$ bisogna che la tabella di parsing $LR(k)$, ottenuta a partire dall'automa canonico $LR(k)$
 - contenga al più una azione per ogni entrata della tabella
 - non esistano due w_1 e w_2 sulle colonne tali che w_1 è prefisso di w_2 e per almeno una riga le 2 corrispondenti entrate contengano un'azione

(Ovviamente la tabella viene compilata nel modo ovvio per le azioni di shift/accept, mentre le reduce sono messe solo in corrispondenza del lookahead w - ad esempio $[A \rightarrow \alpha_0, w]$)

Grammatiche SLR(k)

(45 bis)

Si parte dall'automa canonico $LR(0)$ e si riempie la tabella di parsing $SLR(k)$ - che ha colonne su $T \cup \{\$\}$ (per shift/accept) e in più sui $w \in \text{Follow}_K(A)$ per $A \rightarrow \alpha$. - secondo la legge:

2. se $[A \rightarrow \alpha.] \in S$ e $A \neq S'$, inserisci "reduce $A \rightarrow \alpha$ " in $M[S, w]$ per tutti i $w \in \text{Follow}_K(A)$

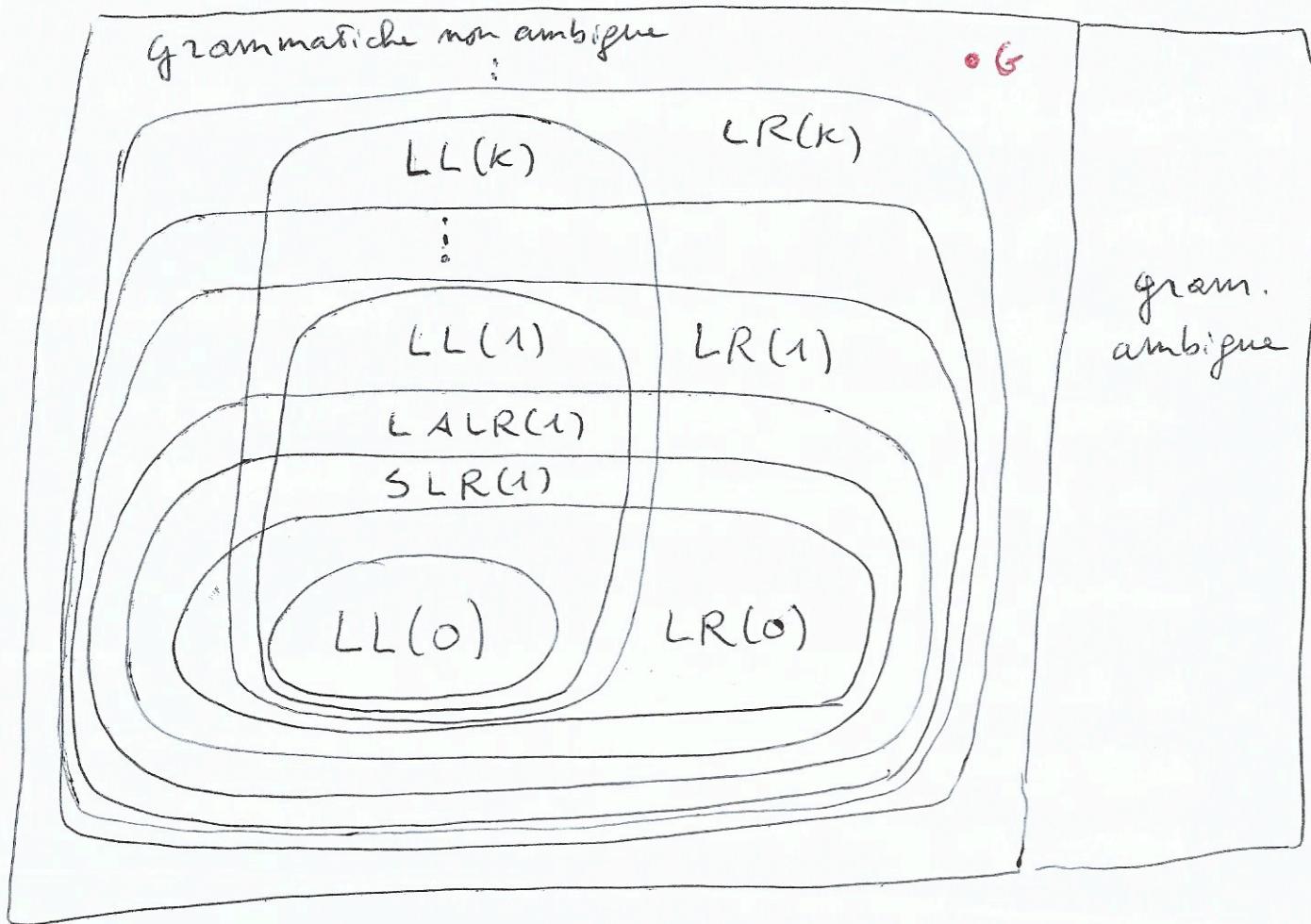
Se la tabella di parsing $SLR(k)$ così riempita non ha conflitti (e se non esistono w_1 e w_2 sulle colonne tali che w_1 sia prefisso di w_2 con una riga su cui le 2 entrate sono entrambe riempite), allora G è $SLR(k)$

Grammatiche LALR(k)

- Si parte dall'automa canonico $LR(k)$ e si fondono insieme gli stati con lo stesso nucleo. Se la tabella risultante non presenta conflitti (e in più non esistono w_1 e w_2 tali che ...), allora G è $LALR(k)$

Relazione tra grammatiche LL(k) e LR(k)

(46)



- LR
- $SLR(k) \subset LALR(k) \subset LR(k)$ per ogni $k \geq 1$
 - $SLR(1) \subset SLR(2) \subset \dots \subset SLR(k)$
 - $LALR(1) \subset LALR(2) \subset \dots \subset LALR(k)$
 - $LR(0) \subset LR(1) \subset \dots \subset LR(k)$

LL vs LR

$$\begin{cases} - LL(k) \subset LR(k) & \text{per ogni } k \geq 0 \\ - LL(k) \not\subset LR(k-1) & \text{per ogni } k \geq 1 \end{cases}$$

Proposizione

- 1) Se $G \in LL(k)$, allora G è non ambigua e $L(G)$ è deterministico
- 2) Se $G \in LR(k)$, allora G è non ambigua e $L(G)$ è deterministico

Oss: Esistono linguaggi generati da grammatiche non ambigue, ma non deterministiche

$$S \rightarrow a Sa \mid b S b \mid \epsilon] G \bullet$$

$$L(G) = \{ ww^R \mid w \in \{a,b\}^*\}$$

Ma cosa possiamo dire dei linguaggi generati da tali grammatiche?

Def Un linguaggio L è di classe X se $\exists G$ di classe X tale che $L = L(G)$.

(dove X sta per $LR(0)/SLR(1)/LL(1) \dots$)

Se classifichiamo i linguaggi, anche le grammatiche, il diagramma si semplifica molto.

Esempio

$$\begin{array}{l} S \rightarrow aAc \\ A \rightarrow bAb \mid b \end{array}$$

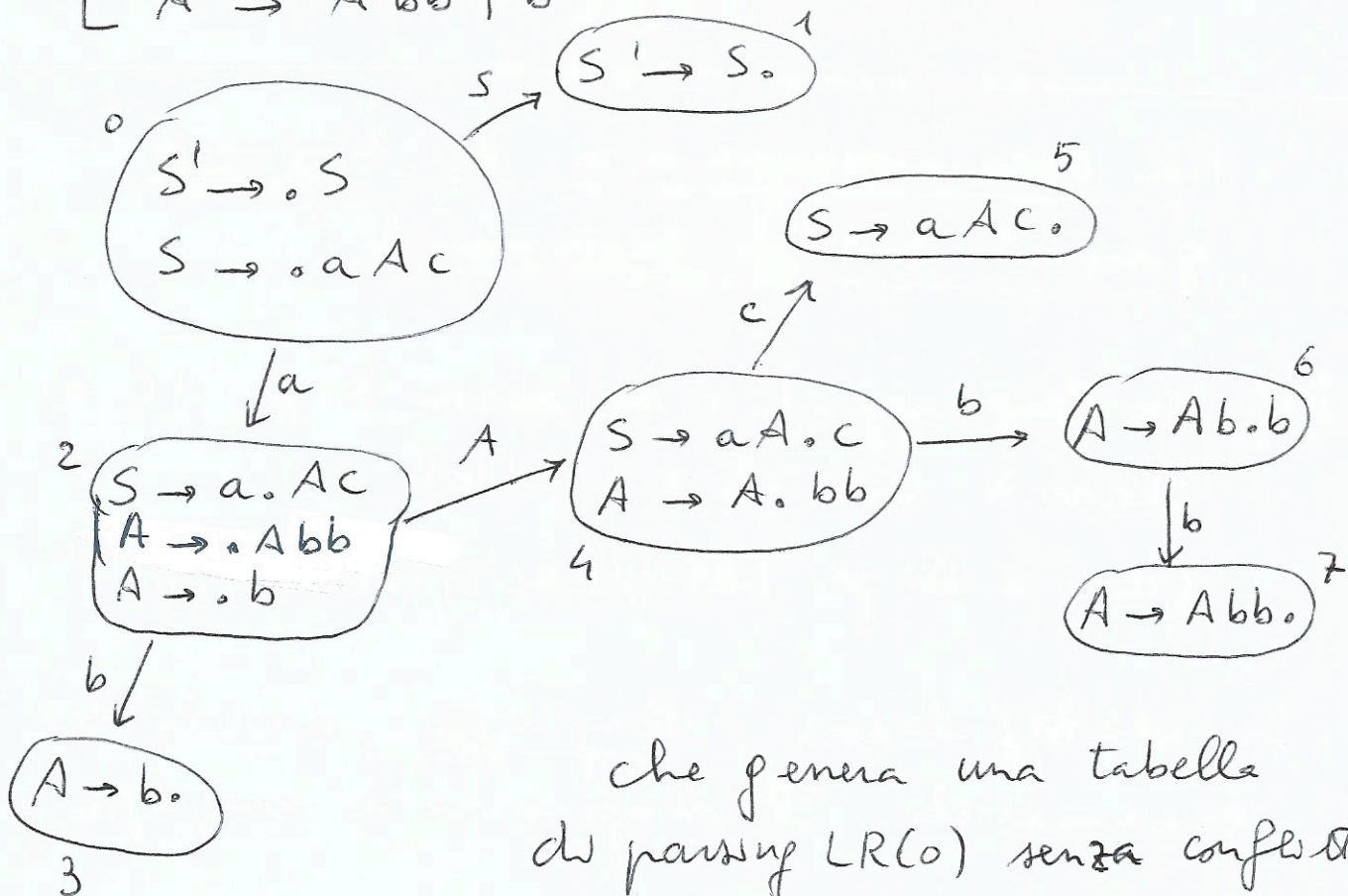
G_1 si può dimostrare non essere LR(k) per nessun k !

$$L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$$

Infatti, non si può mai sapere se in "a^{b^m}" si debba ridurre con $A \rightarrow b$, fino a quando non si incontra "c"; cioè dovrei sapere quando sono nel "messo".

Ma il lang. generato da G_1 è generabile da una gram. LR(0)!

$$G_2 \left[\begin{array}{l} S \rightarrow aAc \\ A \rightarrow A.bbb \mid b \end{array} \right]$$



che genera una tabella di parsing LR(0) senza conflitti.

$$\Rightarrow L(G_1) = L(G_2) \text{ è un lang. LR(0)!}$$

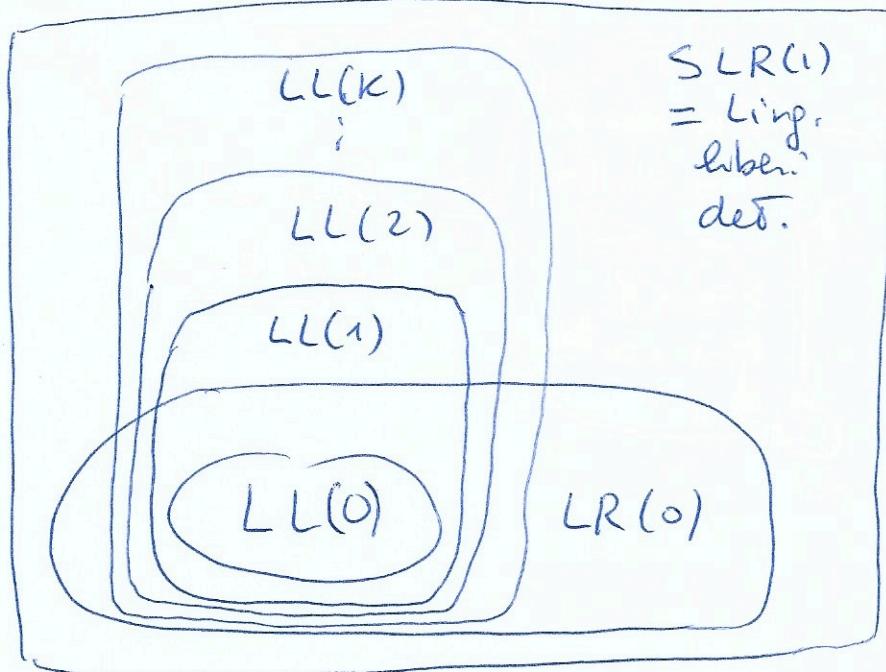
Ricordiamo che:

(49)

- Un lmp. è libero deterministico se è accettato, per stato finale, da un DPDA
- Ogni lmp. regolare è generato da una gram. di classe LL(1)
- Esistono lmp. regolari che non sono LR(0)
(ad es: $L = \{a, ab\}$ vedi pg. 23)

Tesoremi

- (1) Un lmp. è libero det. se è generato da una gram. LR(k) per qualche $k \geq 0$
- (2) Un lmp. è libero det. se è generato da una gram. SLR(1)
- (3) I lmp. generati da gram. LL(k) sono strettamente contenuti nei lmp. generati da gram. SLR(1), $\forall k \geq 0$



Oss: Se $G \in LR(k)$, esiste $G' \in SLR(1)$ equivalente, ma G' può essere molto più complessa di G .

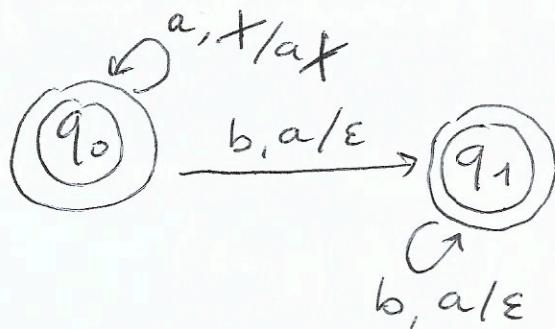
Classificazione dei linguaggi

Esempio

(50)

$$L = \{ a^i b^j \mid i \geq j \geq 0 \} = a^* \cdot \{ a^n b^n \mid n \geq 0 \}$$

è libero deterministico



DPDA che riconosce L
per stato finale

- L non è LL(k) per nessun k !

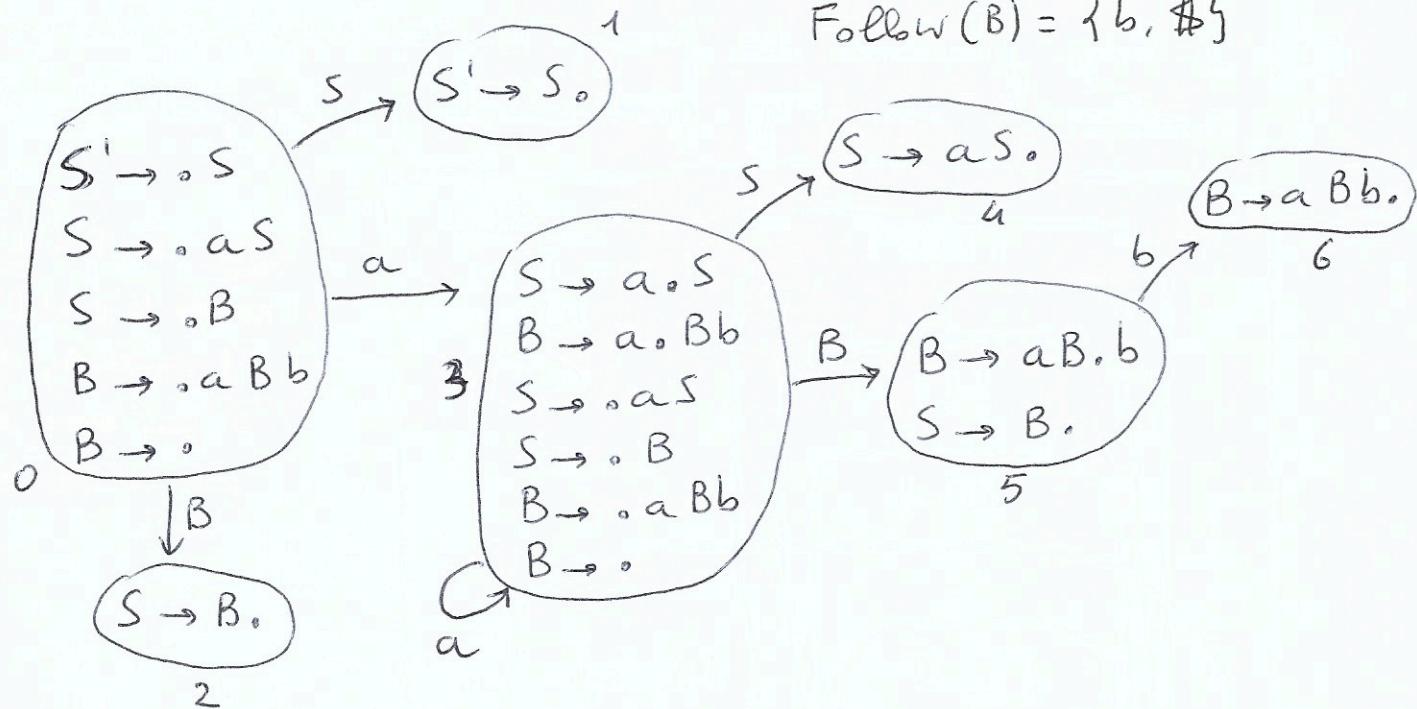
- Tuttavia L è SLR(1)

$$\begin{array}{l} S \rightarrow a^1 S \mid B^2 \\ B \rightarrow a B b \mid \epsilon \end{array} \quad G$$

$$L(G) = L$$

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

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



	a	b	\$	S	B
0	S3	R4	R4	G1	G2
1			ACC		
2			R2		
3	S3	R4	R4	G4	G5
4			R1		
5		S6	R2		
6		R3	R3		

Tabella di parsing SLR(1)
 senza conflitti
 $\Rightarrow G \in SLR(1)$

Osservazioni

- (1) $L_1 = \{a^n b^n \mid n \geq 1\}$ è LL(1) e LR(0)
 $L_2 = \{a^n c^n \mid n \geq 1\}$ è LL(1) e LR(0)

ma $L_1 \cup L_2$ non è LL(k) per nessun k,
 mentre $L_1 \cup L_2 \in LR(0)$!

\Rightarrow l'unione di ling. LL(1) può non essere LL(1)

- (2) $L_1 = \{a\}$ è LL(1) e LR(0)
 $L_2 = \{ab\}$ è LL(1) e LR(0)

ma $L_1 \cup L_2$ non è LR(0), perché non gode
 delle prefix property, mentre $L_1 \cup L_2 \in LL(1)$

\Rightarrow l'unione di ling. LR(0) può non essere LR(0)

(52)

$$(3) \quad L_1 = a^* = \{a^n \mid n \geq 0\} \text{ è } LL(1)$$

$$L_2 = \{a^n b^n \mid n \geq 0\} \text{ è } LL(1)$$

ma $L_1 \cdot L_2 = \{a^i b^j \mid i \geq j \geq 0\}$ non è $LL(k)$
per nessun k

\Rightarrow la concatenazione di due lang. $LL(1)$
può non essere $LL(1)$

Infine, per i più curiosi, ...

$$G_K \left[\begin{array}{l} S \rightarrow aSA \mid \epsilon \\ A \rightarrow a^{k-1}bS \mid c \end{array} \right]$$

$L(G_k)$ è un linguaggio $LL(k)$ per il quale non esiste una grammatica $LL(k-1)$ che lo genera.

Ad esempio

$$G_2 \left[\begin{array}{l} S \rightarrow aSA \mid \epsilon \\ A \rightarrow abS \mid c \end{array} \right] \quad G_2 \text{ non è } LL(1)$$

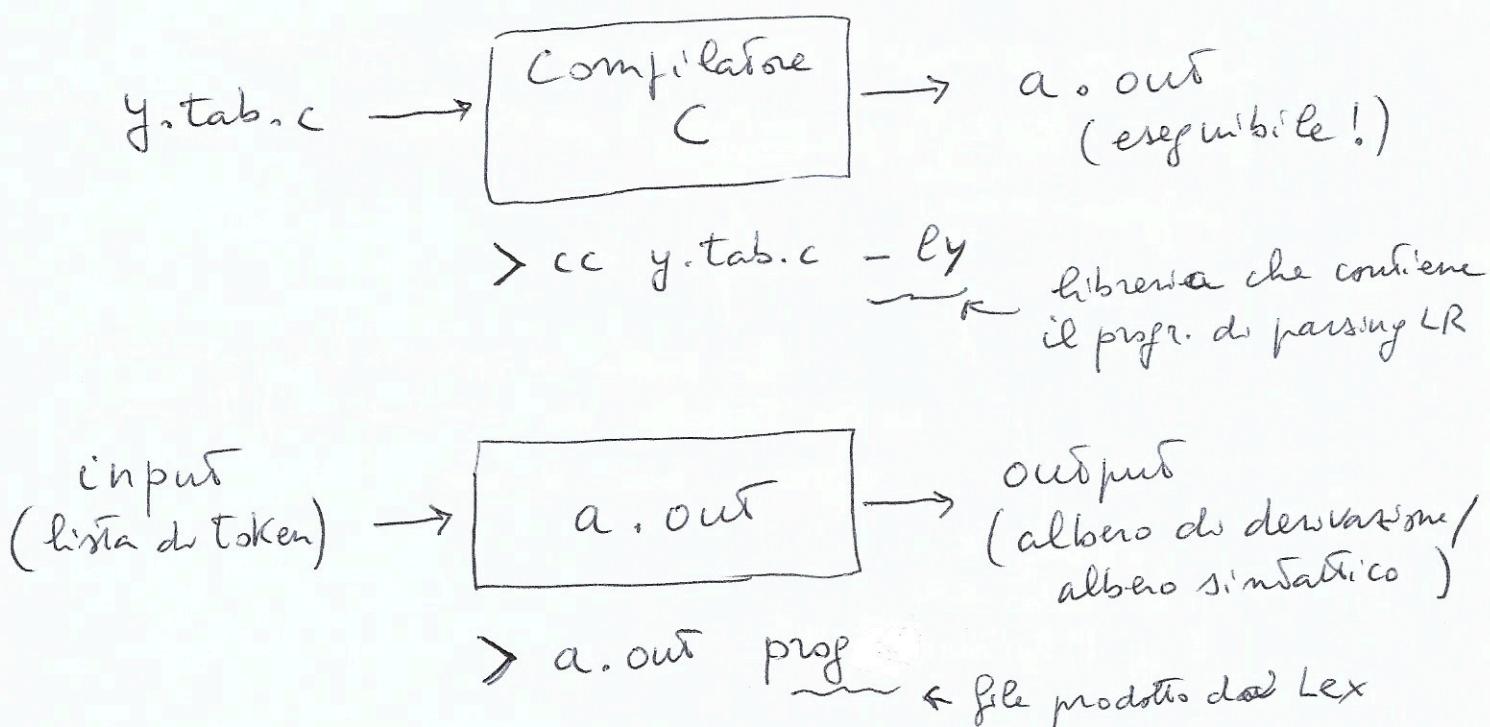
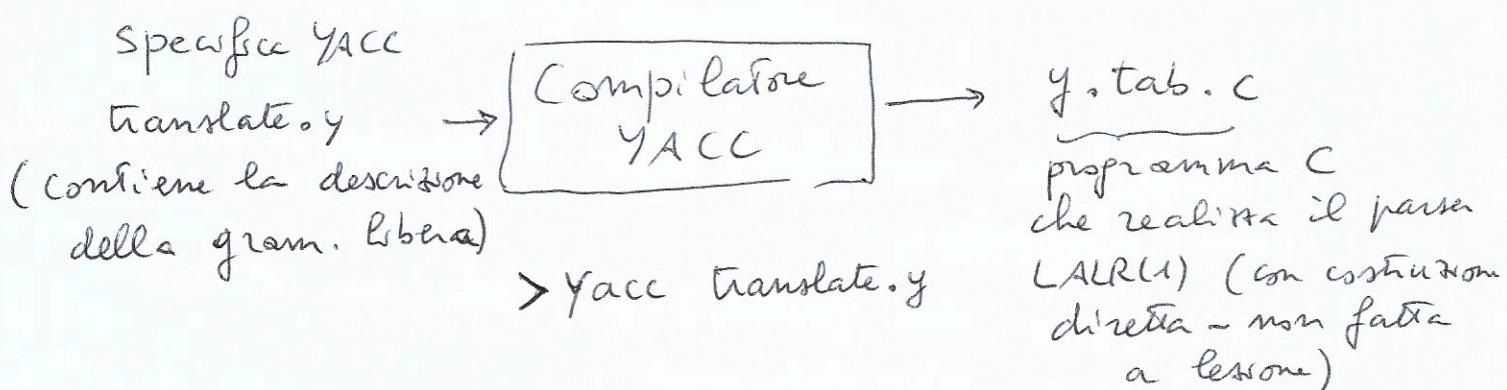
ma è $LL(2)$

e si può dimostrare che non esiste G' di classe $LL(1)$ che genera $L(G_2)$!

Generazione di Analizzatori Sintattici

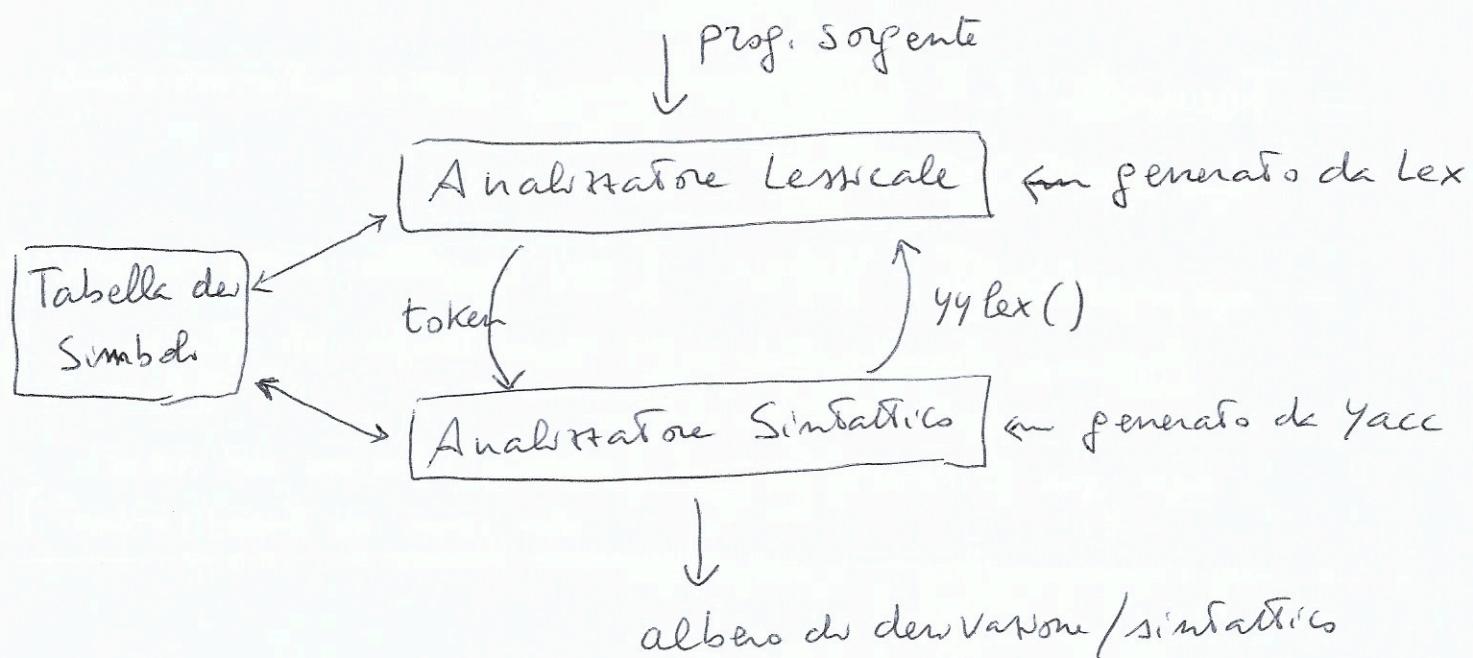
(53)

- Tecniche LR (shift-reduce) sono automatizzate da molti strumenti:
- Capostipite: YACC (Yet another Compiler-Compiler)
(prima versione del 1975 realizzata da Stephen C. Johnson)
ma anche GNU Bison (prima versione del 1985
da parti di Robert Corbett)



In realtà

(54)



- L'analizzatore sintattico usa l'analizzatore lessicale come subroutine
 - Lo invoca con `yylex()` per richiedere il token successivo ("on demand")
 - alcune variabili comuni ai due programmi (come ad esempio `yyval`) permettono di scambiare informazioni
- ↓
lex mette in `yyval` (il puntatore nella Tabella dei simboli al) l'insieme appena individuato

Struttura di un File YACC

(55)

% { prologo % }

- parte optionale che contiene definizioni di macro e altre dichiarazioni di variabili o funzioni che saranno usate nelle sezioni seguenti.
- viene copiato da YACC nel suo output (y.tab.c) in modo da precedere la definizione delle funzione "yyparse" che effettivamente farà l'analisi sintattica

definizioni

- contiene dichiarazioni di simboli usati nella descrizione della grammatica (nomi di token, condizioni con lex)
- in questa sezione è possibile dichiarare la precedenza e l'associatività di alcuni terminali/operatori

%, %.

regole (vedi prossima pagina)

%, %.

funzioni auxiliarie

- contiene le funzioni di supporto per la generazione del parser; tra queste yylex(), funzione che invoca l'analizzatore lessicale e che restituisce il nome del token (che deve essere stato definito nella sezione "definizioni" di YACC) e il suo valore (nella variabile yyval)

Una produzione della forma

nonterm \rightarrow corpo₁ | ... | corpo_k

è espressa in YACC con le regole

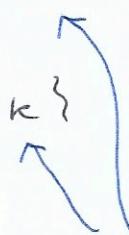
nonterm : corpo₁ {azione semantica 1}

...

| corpo_k

}

{azione semantica k}



è una stringa alfabetica
non dichiarata in precedenza
come token

Codice C eseguito
al momento in cui il
parser riduce mediante
quella produzione

- un carattere tra singoli apici, 'a', è un terminale!
- il simbolo iniziale della grammatica è il nonTerm
usato nella prima regola

Funzione auxiliare: contiene almeno

il parser generato da

- o yylex() con la sua definizione, se YACC è creato senza usare lex
- oppure #include "lex.yy.c" se il parser viene generato insieme da Lex e YACC

Sotto Unix, se la superficie lex è chiamata primo.l e la
specifica YACC second.y \Rightarrow > lex primo.l
> yacc second.y
> cc y.tab.c -ly -ll

L'azione semantica

(57)

L'azione semantica (codice C) calcola il "valore semantico" della Testa della "produzione" in funzione dei valori semantici dei simboli che compongono il corpo.

Esempio: Il "valore semantico" potrebbe essere:

- (a) l'albero di derivazione, nel caso in cui stiamo producendo un compilatore che genera alberi esplicitamente.
- (b) il codice intermedio, connesso alla produzione, se andiamo direttamente a produrre codice intermedio
- (c) la vera e propria valutazione dell'espressione, se stiamo in realtà producendo un interprete

In una azione semantica:

- $\$ \$$ si riferisce al valore semantico della Testa
- $\$ _i$ si riferisce al valore semantico dell' i -esimo simbolo nel corpo della produzione

Vediamo un esempio di file `yy.c`, che lavora
insieme con Lex, perché nelle funzioni auxiliary
c'è `#include "lex.yy.c"`

Generazione di un interprete per espressioni aritmetiche

(58)

% { /* PROLOGO */

define YYSTYPE double dichiaro il tipo della pila interna di YACC usata per i valori semantiche

include <math.h> libreria standard C con macro per le funzioni matematiche

include <stdio.h> libreria standard C con macro per gestire input/output

% }

/* DEFINIZIONI */

% token NUM

definizione del tipo di token NUM
(la cui vera definizione è presente nell'assonato file di lex.c)

% left '-' '+' }

token costituiti da un solo carattere che non necessitano di essere dichiarati token

% left '*' '/' }

- left vuol dire che sono operatori associativi a sinistra a dx

% right NEG

/* meno unario */ specifica la precedenza tra gli operatori: - unario > *, / > -, +

/* /* REGOLE E AZIONI SEMANTICHE */

input :

/* empty */

corrisponde alle produzioni

| input line

input → ε | input line
(z. s.)

;

azione
semantica
vista

line :

'\n'

← a capo = line vista

| exp '\n' {printf ("%f\n", \$1);}

;

corrisponde alle produzioni
line → \n | exp \n

stampa in formato "reale"
(notazione scientifica o virgola mobile, a seconda di cosa è più breve) seguito da "a capo"

⇒ l'input è un file con una espressione su ogni riga (59)
e il parser/interprete stampa il valore di ciascuna
espressione, una per riga.

N.B. - Operatori dichiarati sulla stessa riga hanno
uguale precedenza / priorità

- un operatore ha precedenza su tutti gli operatori
delle linee precedenti

exp : NUM { \$\$ = \$1; }

| exp '+' exp { \$\$ = \$1 + \$3; }

| exp '-' exp { \$\$ = \$1 - \$3; }

| exp '*' exp { \$\$ = \$1 * \$3; }

| exp '/' exp { \$\$ = \$1 / \$3; }

| '-' exp %prec NEG { \$\$ = -\$2; }

| '(' exp ')' ↑ { \$\$ = \$2; }

;

questo '-' è come NEG e quindi ha
la precedenza massima ("tag" %prec)

Corrisponde alle produzioni

exp → NUM | exp + exp | exp - exp | exp * exp | exp / exp |
- exp | (exp)

grammatica ambigua

%% /* Funzioni Auxiliari */

#include "lex.yy.c"

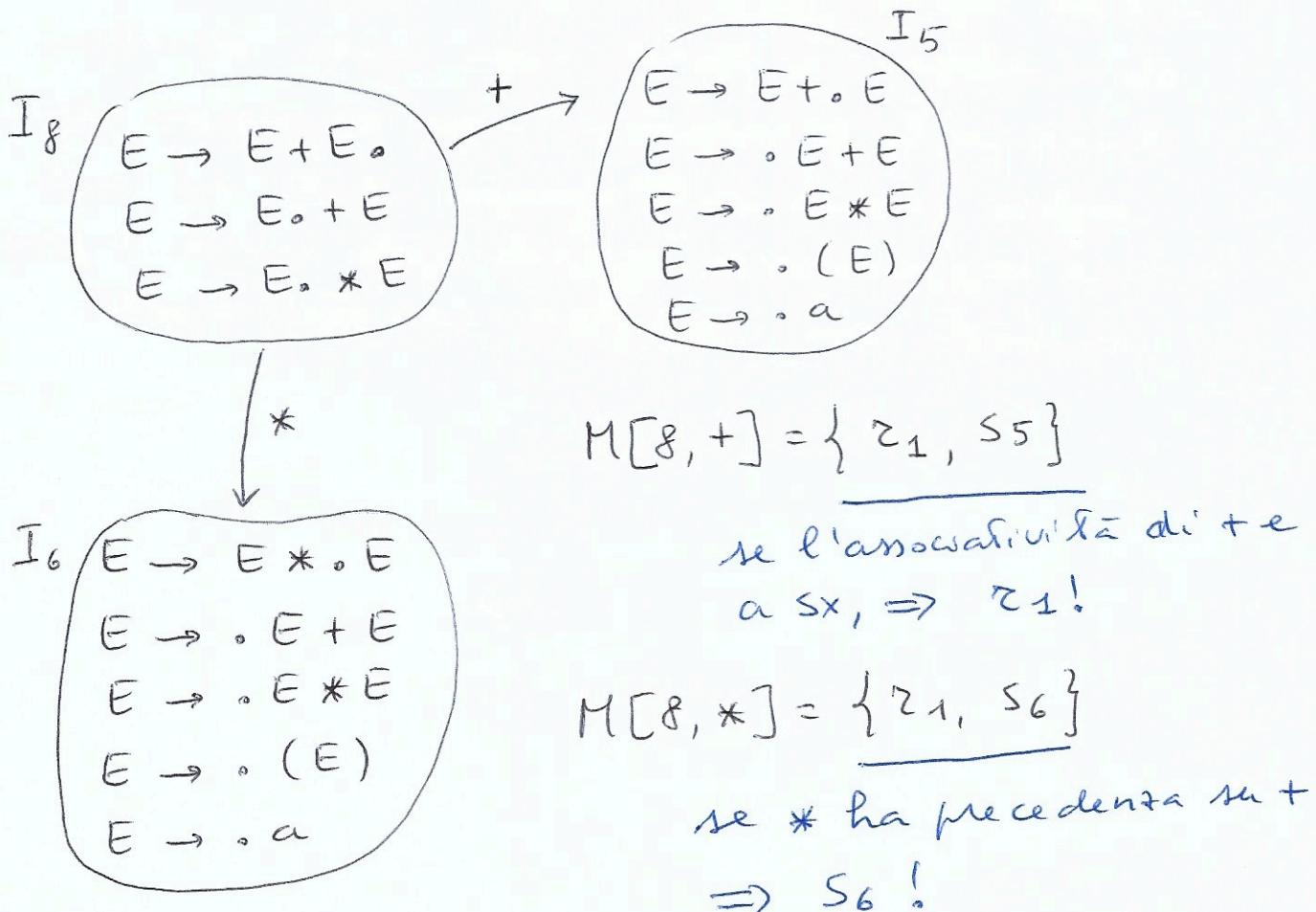
La Tabella LALR(1) che YACC genera, poiché la grammatica è - di base - ambigua, potrebbe generare conflitti. Ma usando le informazioni aggiuntive su

- assocatività
- precedenza degli operatori

è possibile risolvere tutti i conflitti !!

Esercizio Costruire l'automa LR(0) per la gram. delle espressioni (ambigua) — consultare libri capitoli 4.8.7

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$



\Rightarrow sfruttando le informazioni dichiarate su assocatività e precedenza, YACC sa risolvere i conflitti e genera un parso corretto anche per grammatiche "potenzialmente" ambigue

In generale YACC, in assenza di indicazioni, risolve

- conflitti di tipo shift/reduce a favore dello shift
- conflitti di tipo reduce/reduce a favore della produzione elencata prima

E' possibile invocare YACC con l'opzione -v

Questa opzione genera un file aggiuntivo `y.output` che contiene i Kernel degli inservi di item trovati per la grammatica, una descrizione dei conflitti generati dall'algoritmo LALR, ed anche una rappresentazione leggibile delle tabella di parsing che mostra come i conflitti sono stati risolti.