Grammatiche LR(k)

- item LR(k) = [item LR(0), β] con $|\beta| \leq k$
- item iniziale = $[S' \to \circ S, \$]$
- Quando $[A \to \alpha \circ X \gamma, \beta] \in s$ (stato dell'automa canonico LR(k)), allora pure $[X \to \circ \delta, w] \in s$ se $X \to \delta$ è una produzione e $w \in First_k(\gamma\beta)$
- colonne su T^k
- se la tabella di parsing LR(k), ottenuta a partire dall'automa canonico LR(k), contiene al più una azione per ogni entrata, allora G è LR(k)

Grammatiche SLR(k)

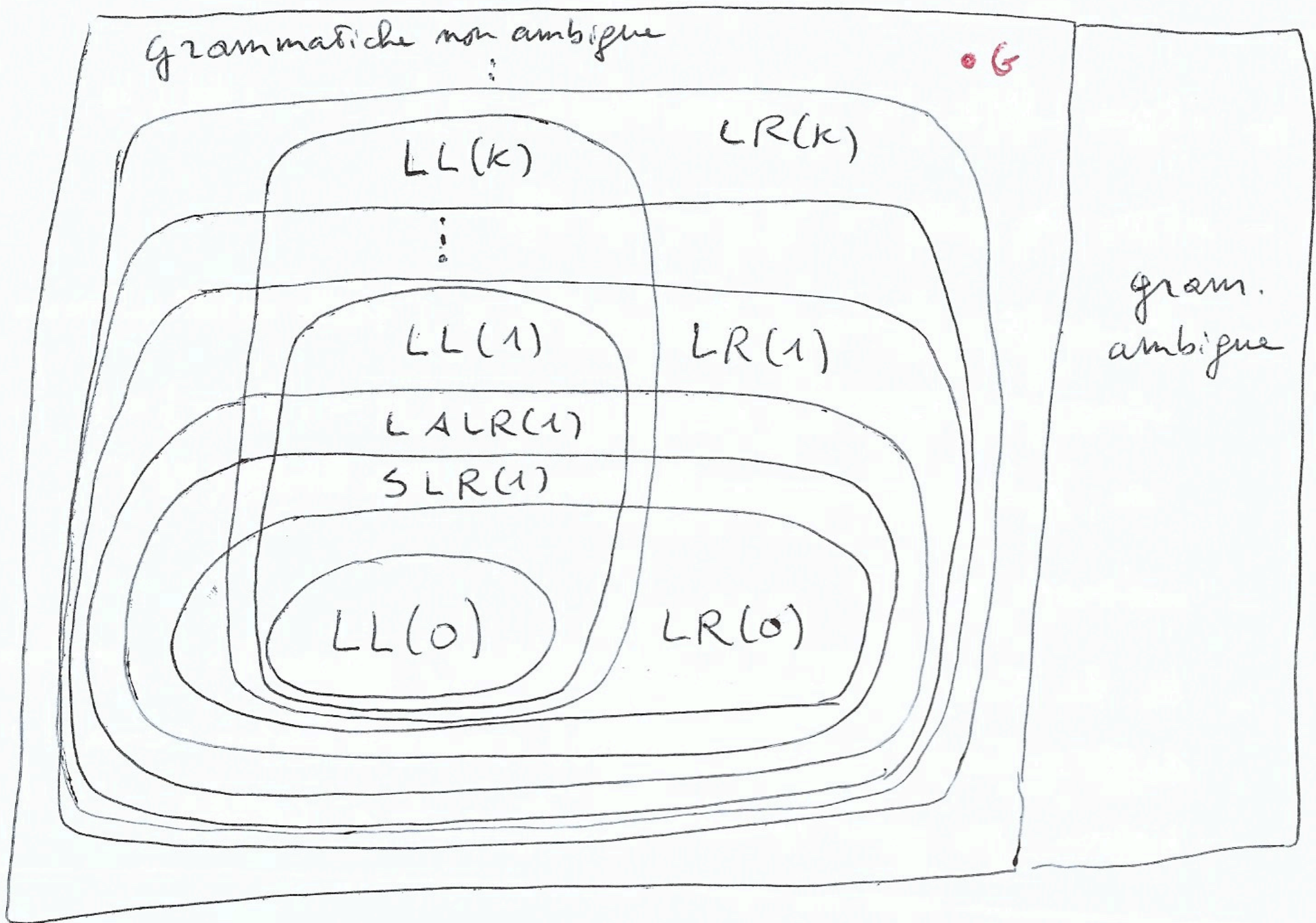
- Si parte dall'automa canonico LR(0) e si riempie la tabella di parsing SLR(k) – che ha colonne su T^k – secondo la legge:

  2. se $[A \to \alpha \circ] \in s$ e $A \neq S'$, inserisci "reduce $A \to \alpha$" in $M[s, w]$ per tutti i $w \in Follow_k(A)$

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
  $\Rightarrow$ G è LALR(k)

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



LR

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

LL vs LR

- $LL(k) \subset LR(k)$ per ogni $k \geq 0$
- $LL(k) \not\subset LR(k-1)$ per ogni $k \geq 1$

Proposizione

1) Se G è $LL(k)$, allora G è non ambigua e $L(G)$ è deterministico

2) Se G è $LR(k)$, allora G è non ambigua e $L(G)$ è deterministico

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

$$S \to aSa \mid bSb \mid \varepsilon \quad \Big] \, G$$

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

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 $LR(0)/SLR(1)/LL(1) \ldots$)

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

Esempio

$S \to a A c$
$A \to b A b \mid b$

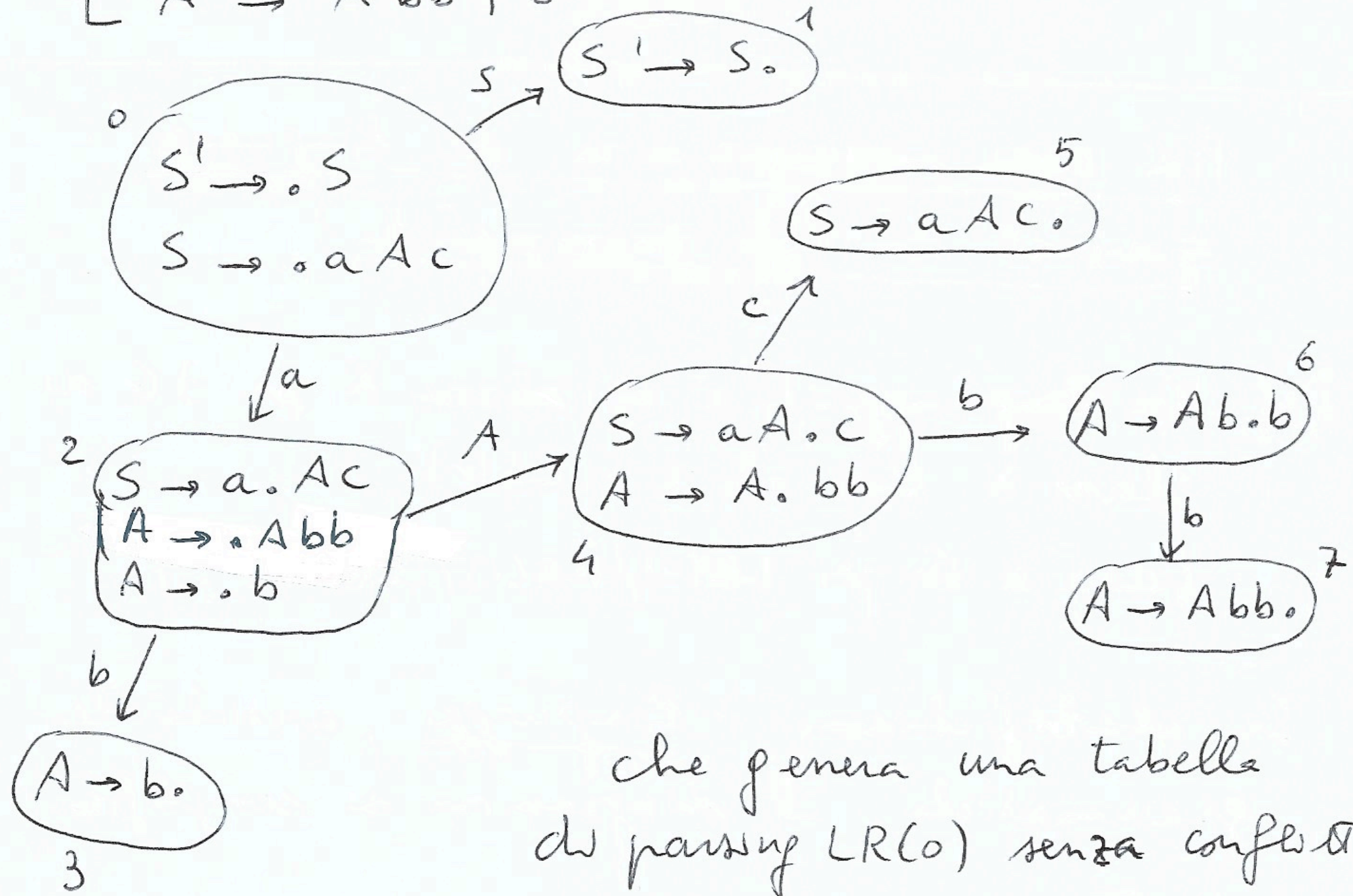
G_1 si può dimostrare non essere LR(k) per nessun k!

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

Infatti, non si può mai sapere se in "a b^m" si debba ridurre con $A \to b$, fino a quando non si incontra "c": cioè dovrei sapere quando sono nel "mezzo"!

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

G_2
$S \to a A c$
$A \to A b b \mid b$



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

$\Rightarrow L(G_1) = L(G_2)$ è un ling. LR(0)!

Ricordiamo che:

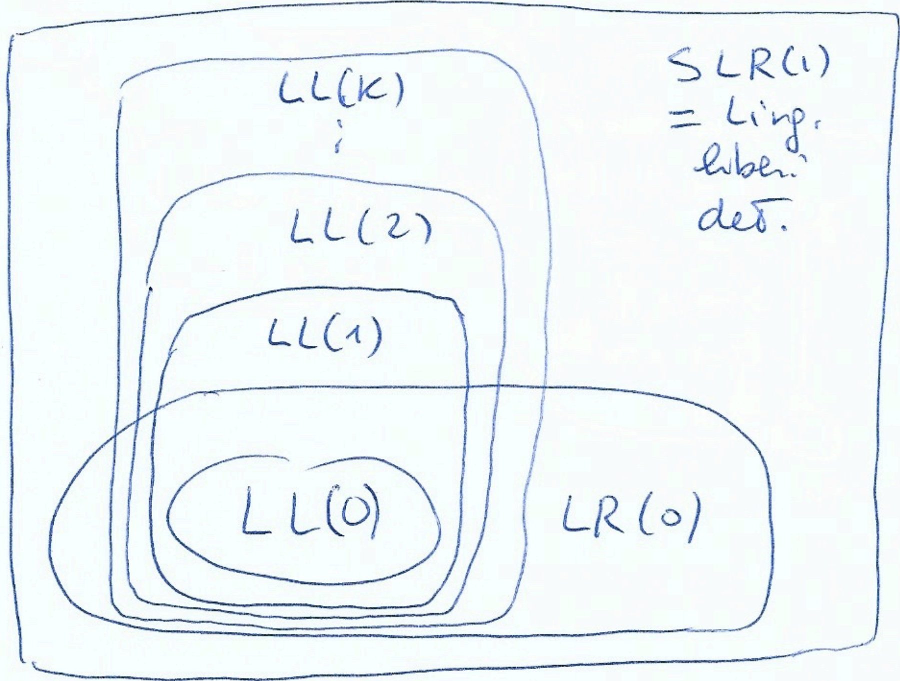
- Un ling. è libero deterministico se è accettato, per stato finale, da un DPDA
- Ogni ling. regolare è generato da una gram. di classe LL(1)
- Esistono ling. regolari che non sono LR(0) (ad es: $L = \{a, ab\}$ vedi pg. 23)

Teoremi

(1) Un ling. è libero det. sse è generato da una gram. LR(K) per qualche $K \geq 0$

(2) Un ling. è libero det. sse è generato da una gram. SLR(1)

(3) I ling. generati da gram. LL(K) sono strettamente contenuti nei ling. generati da gram. SLR(1), $\forall K \geq 0$



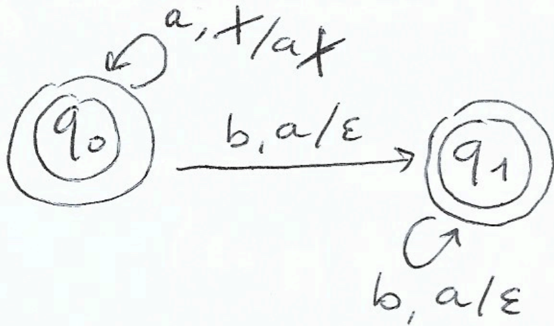
OSS: Se $G \in LR(K)$, esiste $G' \in SLR(1)$ equivalente, ma G' può essere molto più complessa di G

Classificazione dei linguaggi

Esempio

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

è libero deterministico



DPDA che riconosce L per stato finale

- L non è LL(k) per nessun k !

- Tuttavia L è SLR(1)

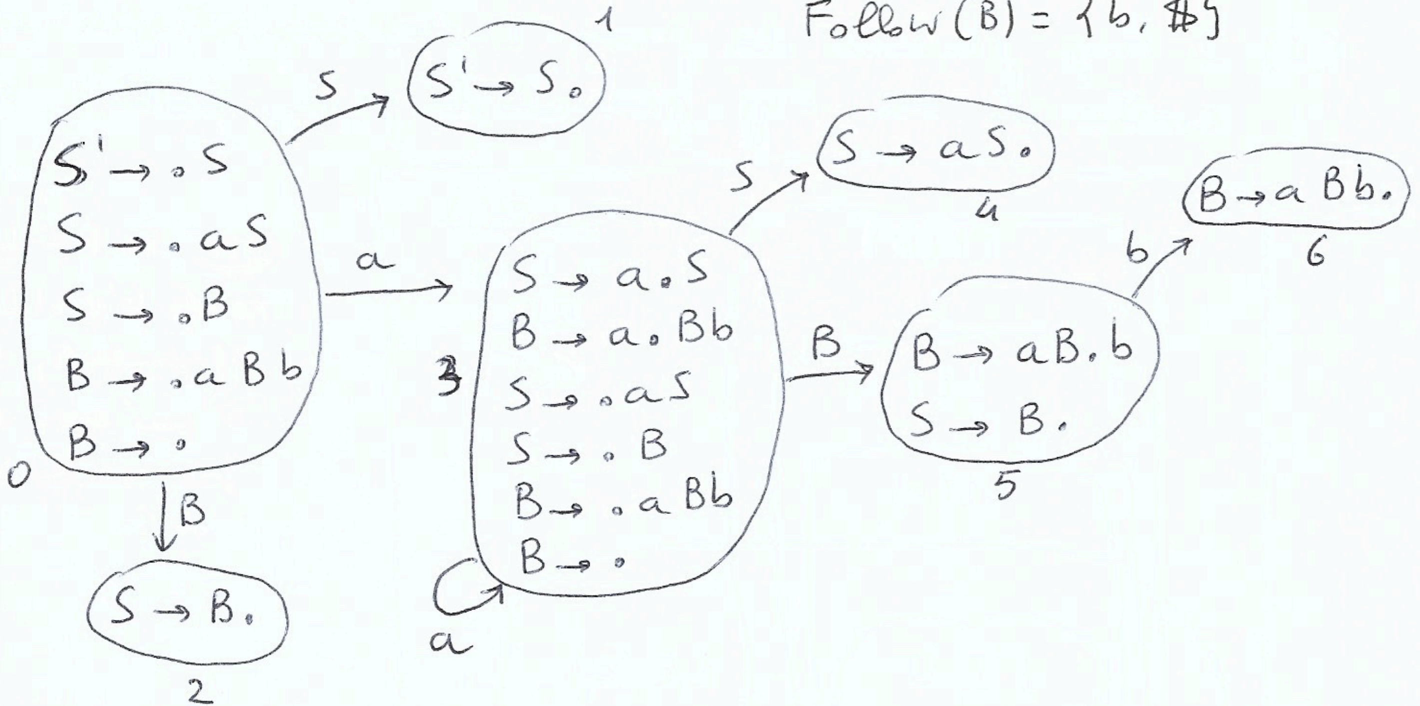
$$\left.\begin{array}{l} S \rightarrow a S \mid B \\ B \rightarrow a B b \mid \varepsilon \end{array}\right] G$$

(produzioni numerate 1: $S \rightarrow aS$, 2: $S \rightarrow B$, 3: $B \rightarrow aBb$, 4: $B \rightarrow \varepsilon$)

$L(G) = L$

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

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



	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 è 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 $L_1 \cup L_2$ è 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 della prefix property, mentre $L_1 \cup L_2$ è LL(1)

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

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

$L_2 = \{a^n b^n \mid n \geq 0\}$ 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 ling. LL(1) può non essere LL(1)

Infine, per i più curiosi, ...

$$G_K \begin{cases} S \to aSA \mid \varepsilon \\ A \to a^{k-1} bS \mid c \end{cases}$$

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

Ad esempio

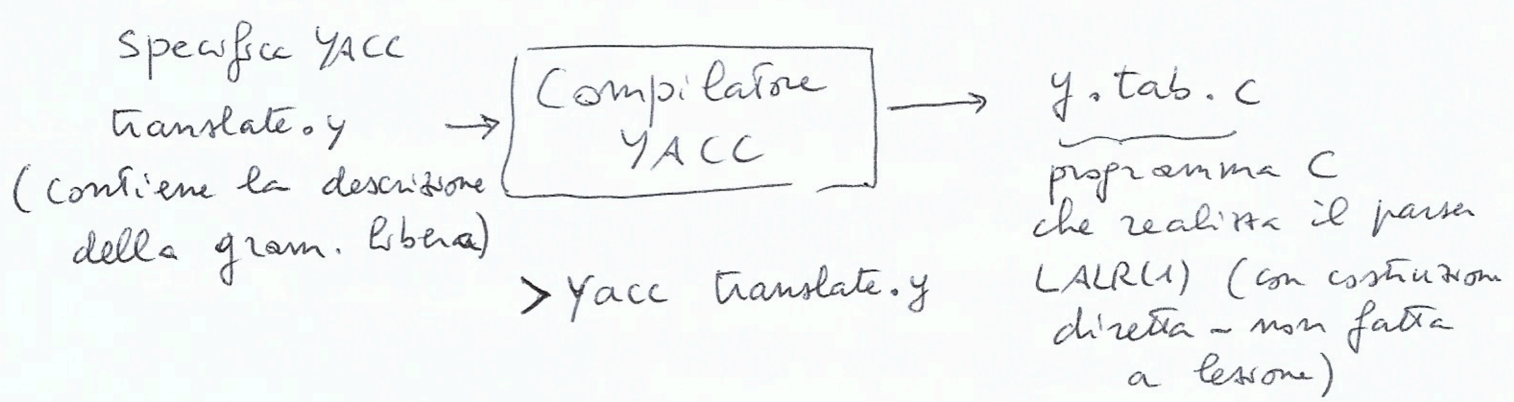
$$G_2 \begin{cases} S \to aSA \mid \varepsilon \\ A \to abS \mid c \end{cases}$$

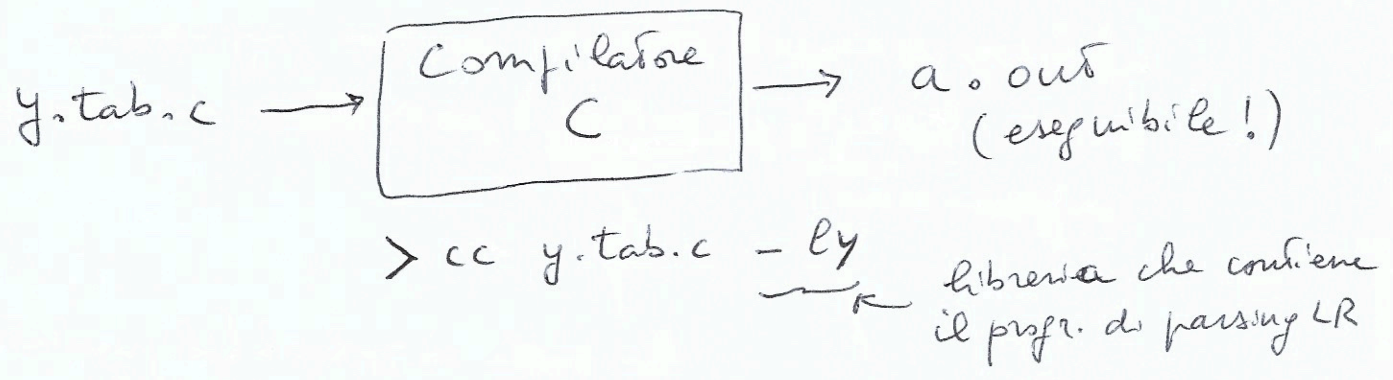
G_2 non è LL(1) ma è LL(2) e si può dimostrare che non esiste G' di classe LL(1) che genera $L(G_2)$!

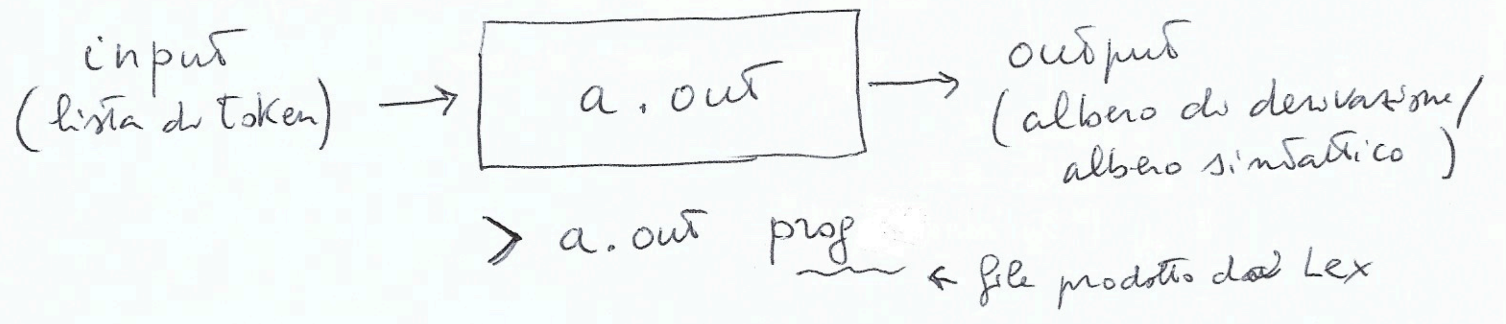
Generatori di Analizzatori Sintattici

- Tecniche LR (shift-reduce) sono automatizzate da molti strumenti
- Capostipite: YACC (Yet another Compiler-Compiler) (prima versione del 1975 realizzata da Stephen C. Johnson)

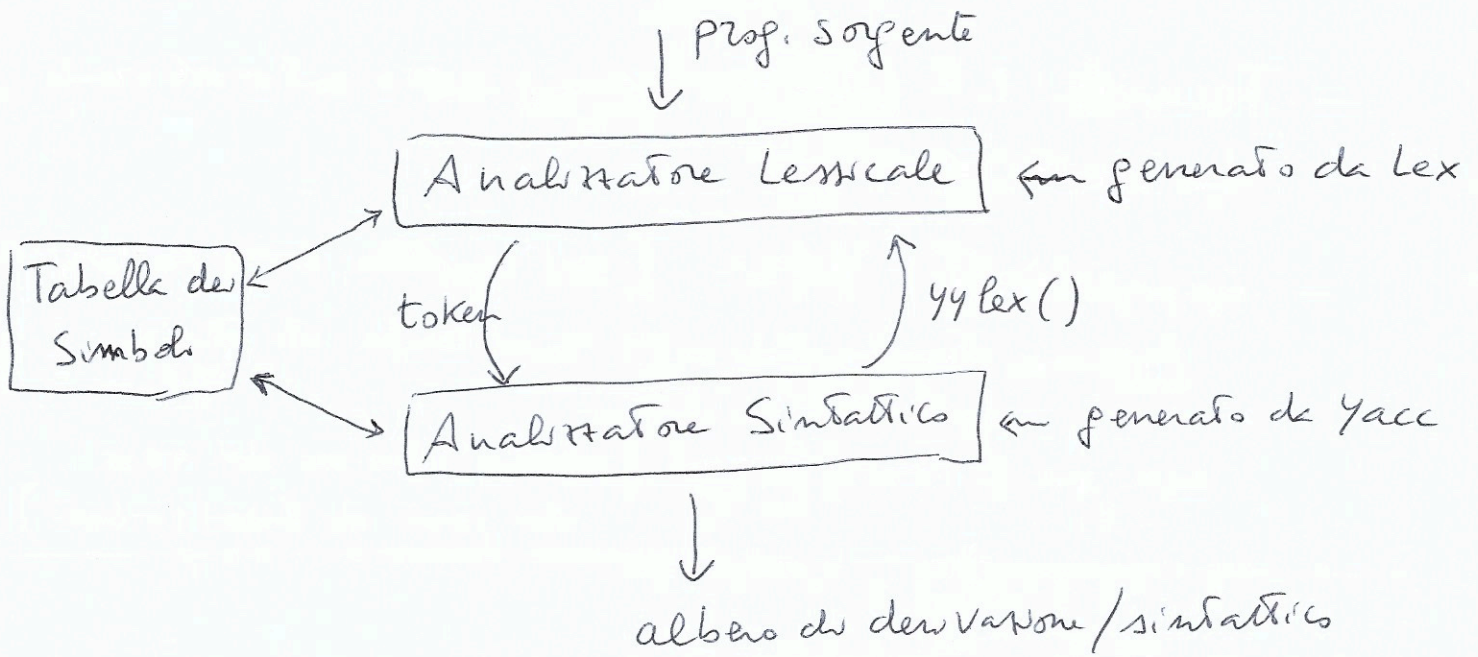
  ma anche GNU Bison (prima versione del 1985 da parte di Robert Corbett)







In realtà



albero di derivazione/sintattico

- 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 yylval) permettono di scambiare informazioni

lex mette in yylval (il puntatore nella Tabella dei simboli al) il lessema appena individuato

Struttura di un File YACC .y

%{ prologo %} – parte opzionale 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 della funzione "yyparse" che effettivamente farà l'analisi sintattica

---

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

%%

regole (vedi prossima pagina)

%%

funzioni ausiliarie
- 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 yylval)

Le regole

Una produzione della forma

$$nonterm \rightarrow corpo_1 \mid \ldots \mid corpo_k$$

è espressa in YACC con le regole

```
nonterm :  corpo1      {azione semantica 1}
           ...
         | corpok      {azione semantica k}
         ;
```

nonterm → è una stringa alfanumerica non dichiarata in precedenza come token

{azione semantica} → 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

---

Funzioni ausiliarie: contiene almeno

- o yylex() con la sua definizione, se il parser generato da 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 specifica Lex è chiamata primo.l e la specifica YACC secondo.y ⇒

```
> lex primo.l
> yacc secondo.y
> cc y.tab.c -ly -ll
```

L'azione semantica

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

Es: 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 <u>interprete</u>

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 .y, che lavora di concerto con Lex, perché nelle funzioni ausiliarie c'è #include "lex.yy.c"

Generazione di un interprete per espressioni aritmetiche

```
%{  /* PROLOGO */
 #define YYSTYPE double
 #include <math.h>
 #include <stdio.h>
%}
 /* DEFINIZIONI */
 %token NUM
 %left '-' '+'
 %left '*' '/'
 %right NEG   /* meno unario */
%%  /* REGOLE E AZIONI SEMANTICHE */
input:      /* empty */
         | input line
   ;
line :     '\n'
         | exp '\n'   {printf ("%g \n", $1);}
;
```

- `#define YYSTYPE double` ← dichiaro il tipo della pila interna di YACC usata per i valori semantici
- `#include <math.h>` ← libreria standard C con macro per le funzioni matematiche
- `#include <stdio.h>` ← libreria standard C con macro per gestire input/output
- `%token NUM` ← definizione del tipo di token NUM (la cui vera definizione è presente nell'associato file di lex .l)
- `%left '-' '+'` / `%left '*' '/'` ←
  - token costituiti da un solo carattere che non necessitano di essere dichiarati token
  - **left** vuol dire che sono operatori associativi a sx / right a dx
  - l'ordine in elenco specifica la precedenza tra gli operatori: - unario > *, / > -, +
- `input: /* empty */ | input line` — corrisponde alle produzioni input → ε | input line (ric. sx); azione semantica vuota
- `'\n'` ← a capo = line vuota
- `line: '\n' | exp '\n'` — corrisponde alle produzioni line → \n | exp \n
- `{printf ("%g \n", $1);}` — `$1` si riferisce a exp; 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 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)

grammatica ambigua

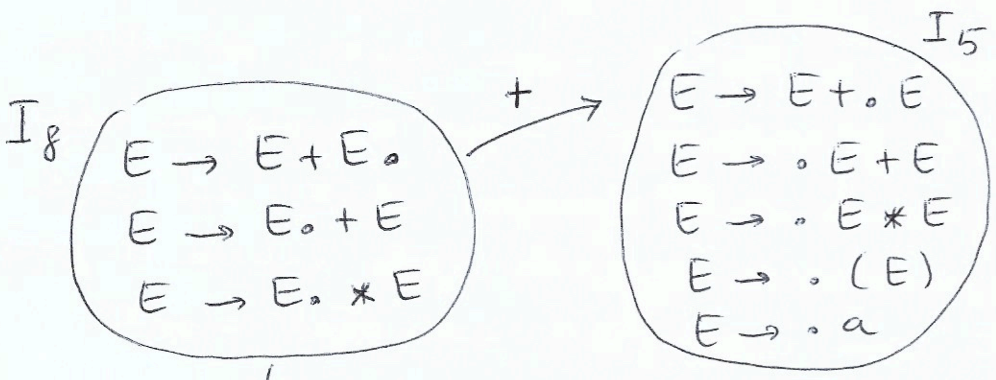
```
%% /* Funzioni Ausiliarie */
#include "lex.yy.c"
```

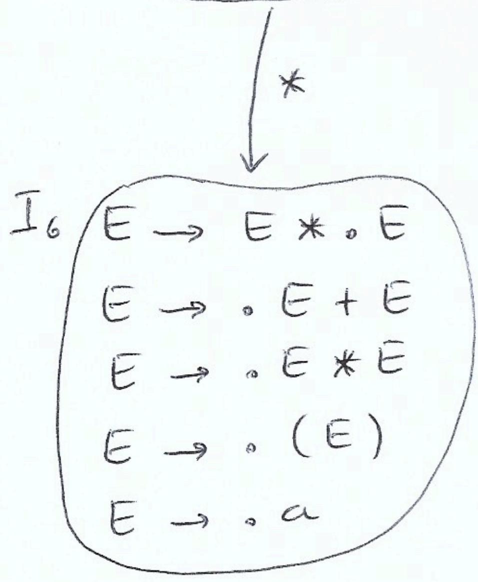
La Tabella LALR(1) che YACC genera, poiché la grammatica è – di base – ambigua, potrebbe generare conflitti. Ma usando le informazioni aggiuntive su
- associatività
- precedenze degli operatori

è possibile risolvere tutti i conflitti!!

<u>Esercizio</u> Costruire l'automa LR(0) per la gram. delle espressioni (ambigua) — consultare libro capitolo 4.8.7

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





$M[8,+] = \{r_1, s_5\}$

se l'associatività di + è a sx, ⇒ r_1!

$M[8,*] = \{r_1, s_6\}$

se * ha precedenza su +
⇒ s_6!

⇒ sfruttando le informazioni dichiarate su associatività e precedenza, YACC sa risolvere i conflitti e genera un parser 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

È possibile invocare YACC con l'opzione -v
Questa opzione genera un file aggiuntivo y.output che contiene i Kernel degli insiemi di item trovati per la grammatica, una descrizione dei conflitti generati dall'algoritmo LALR, ed anche una rappresentazione leggibile delle tabelle di parsing che mostra come i conflitti sono stati risolti.