

Analisi ascendente

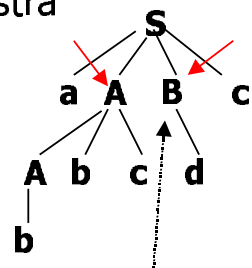
Analisi ascendente (bottom up)

- Si costruisce l'albero di analisi **dalle foglie verso la radice**.
 - Si riduce una stringa allo scopo S
 - Ad ogni passo una sottostringa che corrisponde alla **parte di sinistra** di una produzione è rimpiazzata col **simbolo di destra**
- Se ci sono più stringhe da ridurre, si sceglie quella **più a sinistra**
 - cioè si sceglie di espandere il simbolo terminale più a destra

$S \rightarrow aABe$
 $A \rightarrow Abc|b$
 $B \rightarrow d$

$abbcde \rightarrow aAbcde \rightarrow aAde \rightarrow aABe \rightarrow S$

Prima $b \rightarrow A$ o $d \rightarrow B$?
La stringa più a sinistra

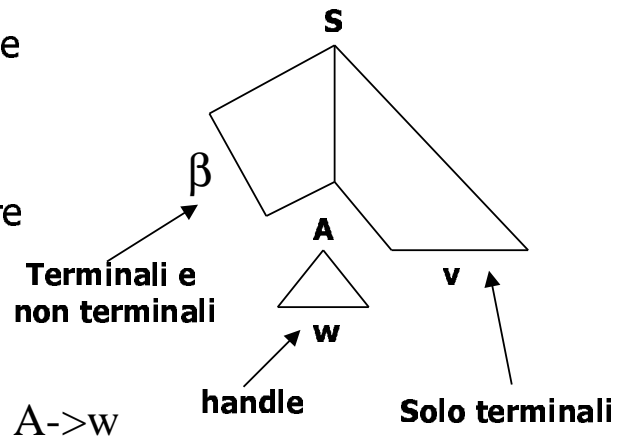


Prima $A \rightarrow Abc$ o $B \rightarrow d$?
Il simbolo più a destra

Le sottostringe "Handle" (di aggancio)

- **Handle**: sottostringa tale che
 - corrisponde al lato destro di una produzione $A \rightarrow w$
 - si può rimpiazzare nella stringa di partenza α con A ottenendo una derivazione destra
 - $S \Rightarrow^* \beta A v \rightarrow \beta w v$, dove v contiene **solo simboli terminali**

- una grammatica **ambigua** può avere più handle



Implementazione

- Si usa uno **stack** per i simboli della grammatica
 - Il parser inserisce simboli sullo stack fino a quando non trova un handle w (ad es. $A \rightarrow w$)
 - Il parser riduce w al simbolo non terminale corrispondente A
- Il parser può fare le seguenti azioni
 - **sposta (shift)**: il prossimo simbolo in ingresso è messo sullo stack
 - **riduce (reduce)**: viene riconosciuto un handle sullo stack e sostituito da un non terminale
 - **accetta**: sullo stack è rimasto S e in ingresso non c'è niente
 - **errore**: si è riconosciuto un errore di sintassi

Un esempio

Stack	Ingresso	Azione
\$	n+n*n\$	Sposta
\$n	+n*n\$	Riduci E->n
\$E	+n*n\$	Sposta
\$E+	n*n\$	Sposta
\$E+n	*n\$	Riduci E->n
\$E+E	*n\$	Sposta
\$E+E*	n\$	Sposta
\$E+E*n	\$	Riduci E->n
\$E+E*n	\$	Riduci E->E*E
\$E+E	\$	Riduci E->E+E
\$E	\$	Accetta

E -> E+E

E-> E*E

E -> (E)

E-> n

Si anche poteva scegliere di ridurre

I conflitti

- Ci sono grammatiche per le quali non si può applicare l'analisi ascendente perché noto il contenuto dello stack e il prossimo simbolo
 - è possibile sia ridurre che spostare (**conflitto shift/reduce**)
 - è possibile ridurre con più produzioni (**conflitto reduce/reduce**)
- Le grammatiche per cui non esistono conflitti si dicono **LR**
- Gli analizzatori bottom up
 - in caso di conflitto **usano un criterio** per decidere cosa fare (ad esempio, se esiste una precedenza fra gli operatori)
 - usano **grammatiche LR** (analizzatori LR)

Analizzatori LR

■ Analisi LR(K)

- L (left to right): analisi della stringa da sinistra a destra
- R (rightmost derivation): si applica una derivazione che espande il non terminale più a destra
- K: numero di simboli di previsione

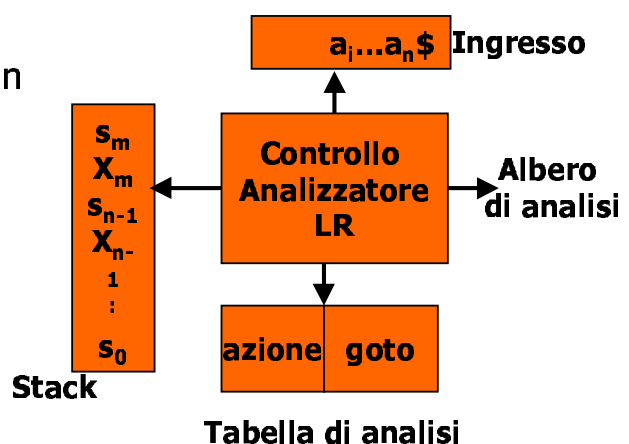
■ LR indica LR(1)

Funzionamento di un analizzatore LR

- Lo **stack** contiene una stringa del tipo $s_0X_1s_1...X_ms_m$,
 - Dove X_i è un **simbolo** della grammatica e S_i uno **stato**
 - Ogni **stato** S_i riassume l'informazione contenuta nello stack al di sotto di esso
 - Nell'implementazione i simboli X_i non sono necessari

- L'azione del parser è determinata **dallo stato in testa allo stack e dal simbolo corrente**

- La tabella di analisi è divisa in una **parte azione e una parte goto**



Funzionamento di un analizzatore LR II

- La **configurazione** dell'analizzatore è dato dal contenuto dello stack e dalla parte di ingresso ancora da leggere

$$s_0 X_1 s_1 \dots X_m s_m a_i \dots a_n$$

- La configurazione rappresenta una stringa derivata sostituendo a destra

$$X_1 \dots X_m a_i \dots a_n$$

- Controllo dell'analizzatore:**

I dati s_m stato in testa allo stack, a_i simbolo corrente si determina l'azione sulla base di $AZIONE[s_m, a_i]$

Funzionamento di un analizzatore LR III

- $AZIONE[s_m, a_i] = \text{sposta } s$**

Si spostano a_i e s sullo stack (si va allo stato s)

Stack = $s_0 X_1 s_1 \dots X_m s_m$ Ingresso = $a_i a_{i+1} \dots a_n$

Stack = $s_0 X_1 s_1 \dots X_m s_m a_i s$ Ingresso = $a_{i+1} \dots a_n$

- $AZIONE[s_m, a_i] = \text{riduci } A \rightarrow w$ ($w = X_{m-r+1} \dots X_m$)**

Si rimuove $r = |w|$ simboli dallo stack, si mette A sullo stack a_i e si va allo stato $s = GOTO(S_{m-r}, A)$

Stack = $s_0 X_1 s_1 \dots X_{m-r} s_{m-r} X_{m-r+1} s_{m-r+1} \dots X_m s_m$ Ingresso = $a_i \dots a_n$

Stack = $s_0 X_1 s_1 \dots X_{m-r} s_{m-r} A s$ Ingresso = $a_i \dots a_n$

- $AZIONE[s_m, a_i] = \text{accetta}$:** L'analisi è completata

- $AZIONE[s_m, a_i] = \text{errore}$:** Si è scoperto un errore

Un esempio

stat	n	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

- (1) $E \rightarrow E+T$ (2) $E \rightarrow T$
 (3) $T \rightarrow T * F$ (4) $T \rightarrow F$
 (5) $F \rightarrow (E)$ (6) $F \rightarrow n$

Stack	Input	Azione	Goto
0	n*n+n\$	s5	
0n5	*n+n\$	r6	3
0F3	*n+n\$	r4	2
0T2	*n+n\$	s7	
0T2*7	n+n\$	s5	
0T2*7n5	+n	r6	10
0T2*7F10	+n\$	r3	2

Un esempio II

stat	n	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

- (1) $E \rightarrow E+T$ (2) $E \rightarrow T$
 (3) $T \rightarrow T * F$ (4) $T \rightarrow F$
 (5) $F \rightarrow (E)$ (6) $F \rightarrow n$

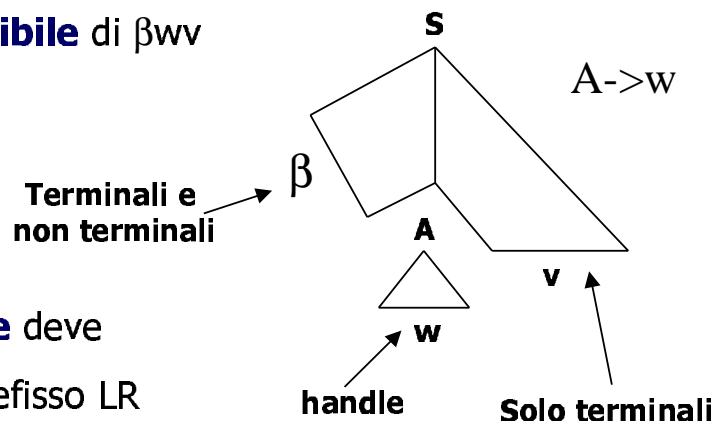
Stack	Input	Azione	Goto
0T2	+n\$	r2	1
0E1	+n\$	s5	
0E1+6	n\$	s5	
0E1+6n5	\$	r6	3
0E1+6F3	\$	r4	9
0E1+6T9	\$	r1	1
0E1	\$	acc	

Perché gli analizzatore LR funzionano?

- La stringa βw è un **prefisso riducibile** di $\beta w v$

- Si dice **prefisso LR** un qualsiasi prefisso di un prefisso riducibile

- Un analizzatore LR per **funzionare** deve
 - riconoscere la presenza di un prefisso LR
 - decidere se il prefisso è riducibile
 - decidere con quale regola ridurre



Perché gli analizzatori LR funzionano? II

- I simboli dello stack sono **prefissi LR**
 - sono prefissi riducibili quando si fa una riduzione
 - sono semplici prefissi LR quando si uno spostamento
- Si può dimostrare che i prefissi LR sono **riconoscibili da un automa a stati finiti**
- Lo stato in cima allo stack è rappresenta lo stato in cui si trova l'automa a stati finiti
- Un automa LR **utilizza lo stato in testa allo stack per sapere a che punto è nella lettura del prefisso e si aiuta con il primo simbolo in ingresso per decidere cosa fare**

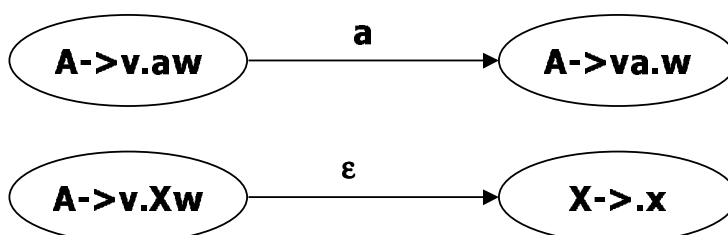
Costruzione dell'automa che riconosce i prefissi

- Un **elemento LR(0)** di una grammatica è una produzione annotata con un punto
 - $A \rightarrow \cdot xy a$
 - $A \rightarrow x \cdot ya$
 - $A \rightarrow xy \cdot a$
 - $A \rightarrow xya \cdot$
- Un elemento è individuato da una coppia di indici: numero produzione, posizione punto
- Il **punto** tiene traccia di quanto è stato già letto

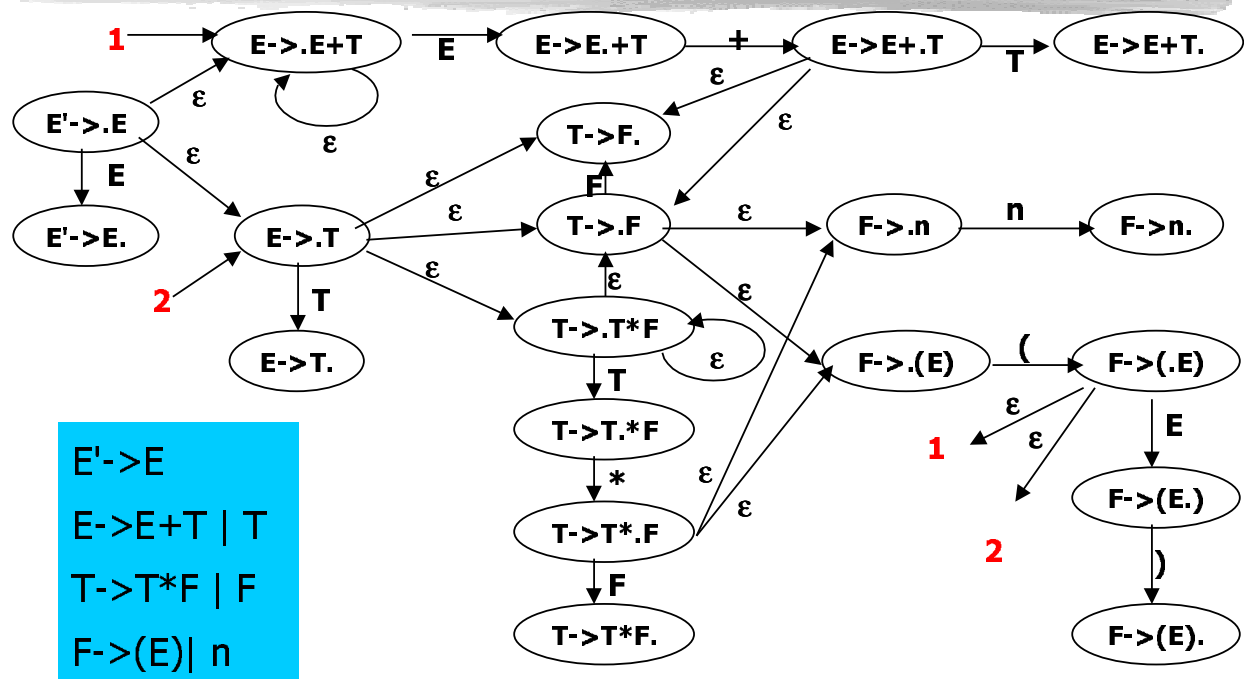
Costruzione dell'automa che riconosce i prefissi II

■ L'automa

- gli **stati** sono elementi LR(0)
- se $A \rightarrow vaw \in P$ e $a \in T$, c'è un **arco**
 $[A \rightarrow \cdot vaw] \xrightarrow{a} [A \rightarrow va \cdot w]$
- se $A \rightarrow vXw \in P$ e $X \in V$ c'è un **arco**
 $[A \rightarrow \cdot vXw] \xrightarrow{\epsilon} [X \rightarrow \cdot x]$



Costruzione dell'automata che riconosce i prefissi: un esempio

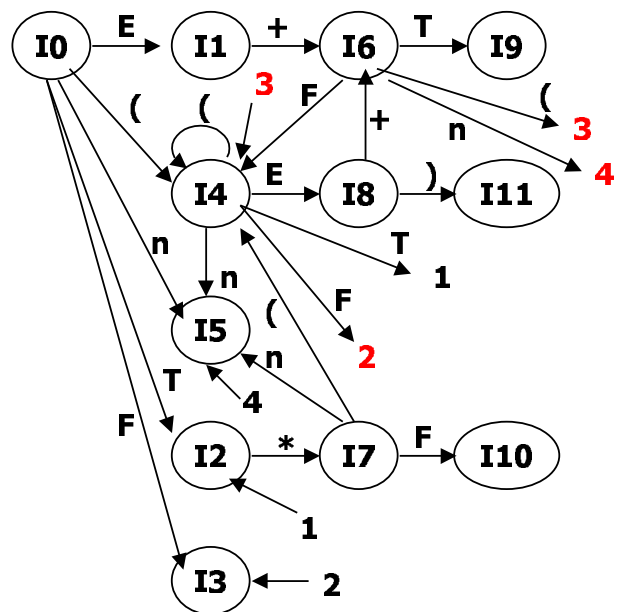


L'automata deterministico

- Gli automi costruiti con il metodo precedente sono **non deterministici**
 - occorre trasformarli in automi deterministici
 - si usa la solita procedura per passare da NFA a DFA
- Alla fine **ogni stato** I dell'automata deterministico corrisponde ad **un insieme di elementi LR(0)**
 - ad es. $I = \{F \rightarrow \cdot n, F \rightarrow \cdot (E), T \rightarrow T* \cdot F\}$

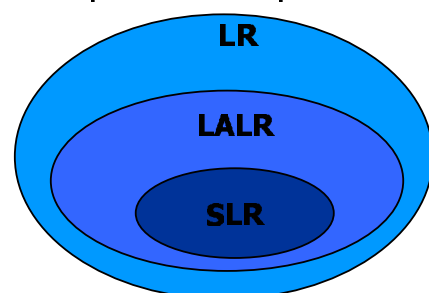
L'automata deterministico: un esempio

$I_0 = \{E' \rightarrow .E, E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .F, T \rightarrow .T*F, F \rightarrow .n, F \rightarrow .(E)\}$
 $I_1 = \{E' \rightarrow E, E \rightarrow E.+T\}$
 $I_2 = \{E \rightarrow T., E \rightarrow E.+T\}$ $I_3 = \{T \rightarrow F.\}$
 $I_4 = \{F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .F, T \rightarrow .T*F, F \rightarrow .n, F \rightarrow .(E)\}$
 $I_5 = \{F \rightarrow n.\}$
 $I_6 = \{T \rightarrow .F, T \rightarrow .T*F, F \rightarrow .n, F \rightarrow .(E)\}$
 $I_7 = \{F \rightarrow .n, F \rightarrow .(E), T \rightarrow T*.F\}$
 $I_8 = \{F \rightarrow (E.), F \rightarrow E.+T\}$
 $I_9 = \{E \rightarrow E+T., T \rightarrow T*.F\}$
 $I_{10} = \{T \rightarrow T*F.\}$ $I_{11} = \{F \rightarrow (E). \}$



Costruzione di tabelle di analisi

- Dopo l'automata con stati $\{I_1, \dots, I_n\}$ si costruiscono le **tabelle di analisi**
- Esistono vari modi di costruire le tabelle
 - Analizzatori **SLR** (simple LR), **LALR** (Lookahead LR), e **LR**
 - **tabelle più grandi** e complesse da costruire corrispondono a minori conflitti e **linguaggi più generici**
 - **tabelle più piccole** e semplici da costruire corrispondono a più conflitti e **linguaggi più ristretti**
- La maggior parte dei compilatori usa LALR
- **Noi vedremo SLR**



Costruzione di tabelle di analisi SLR

- Per un analizzatore **SLR (simple LR)**, AZIONE e GOTO saranno

- **Spostamenti**

■ se $a \in T$ e l'automa contiene la transizione $I_h \xrightarrow{a} I_k$

AZIONE[h,a]= sposta k

■ (si passa al nuovo stato dell'automa, il prefisso **non è ancora riducibile**)

- **Riduzioni**

- se $[A \rightarrow w.] \in I_h$ e $a \in \text{FOLLOW}(A)$

AZIONE[h,a]= riduci $A \rightarrow w$

■ (Si riduce, il prefisso **è riducibile**)

Costruzione di tabelle di analisi SLR II

- **Riduzioni**

- se $[S' \rightarrow S.] \in I_h$

AZIONE[h,\$]= accetta

■ ($S' \rightarrow S$ è una **nuova regola** aggiunta al linguaggio)

- **La tabella GOTO**

■ se $I_h \xrightarrow{A} I_k$

GOTO[h,A]=k

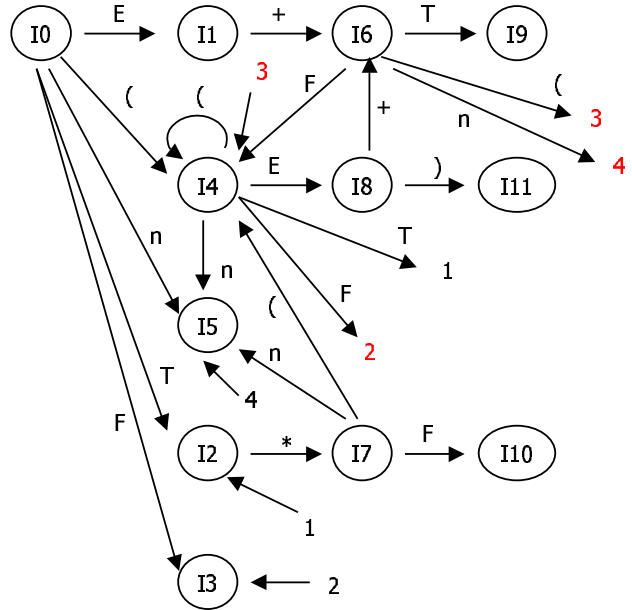
■ Lo statp iniziale è quello che contiene $[S' \rightarrow .S]$

■ Le componenti non definite producono un errore

Costruzione della tabella

AZIONE: gli spostamenti

stato	n	+	*	()	\$
0	s5			s4		
1		s6				
2			s7			
3						
4	s5			s4		
5						
6	s5			s4		
7	s5			s4		
8		s6			s11	
9			s7			
10						
11						



124

Costruzione della tabella

AZIONE: le riduzioni

state	n	+	*	()	\$
0	s5			s4		
1		s6				acc
2		r2	s7		r2	r2
3		r4	r4		r4	r4
4	s5			s4		
5		r6	r6		r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r1	s7		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

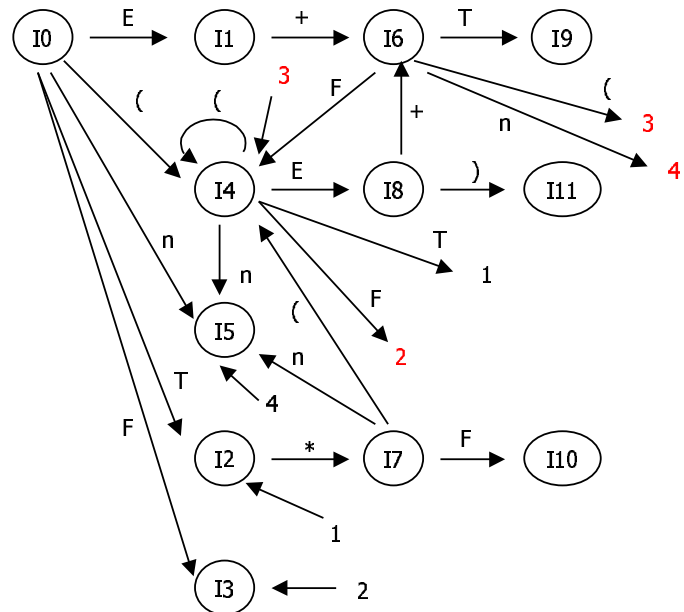
(6) $F \rightarrow n$

$$I11 = \{F \rightarrow (E), \dots\}$$

FOLLOW(T)={+,*,), \$} **FOLLOW(F)={+,*,), \$}**

Costruzione tabella GOTO: un esempio

stato	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			



Fondamenti II 2003

Franco Scarselli

126

Grammatiche LR(1)

- Una grammatica si dice **SLR(1)** se la corrispondente tabella di analisi costruita come appena mostrato **non contiene conflitti**
- Costruzione di tabelle per analizzatori LR (idea base)
 - Gli stati memorizzano **esplicitamente i non terminali b che possono seguire un handle y**
 - Gli elementi di una grammatica LR(1) sono del tipo $[A \rightarrow \cdot xya, b]$, $[A \rightarrow x \cdot ya, b]$
 - Si costruisce un automa deterministico con tali elementi
 - Con $[A \rightarrow w, b]$ **si riduce solo se il prossimo simbolo è b**

Fondamenti II 2003

Franco Scarselli

127

YACC: Yet Another Compiler Compiler

- **Yacc** è un generatore di analizzatori lessicali in linguaggio C a partire dalla grammatica LALR(1)
- L'ingresso è un file che descrive la grammatica: **simboli non terminali, simboli terminali, produzioni**

Terminali →

Non terminali →

Produzioni →

```
%token <val> NUM
%token <sptr> VAR
%type <val> exp

%%
input: /* vuota */
      | input exp '\n'
;
exp:  NUM {$$=$1;}
     VAR {$$=$1->value.var;}
     VAR '=' exp {$$=$3;$1->value.var=$3;}
     exp '+' exp {$$=$1+$3;}
     exp '*' exp {$$=$1*$3;}
     '(' exp ')' {$$=$2;}
;
```

Fondamenti II 2003 Franco Scarselli 128

I simboli

- Ad ogni simbolo è associato **un tipo e un valore semantico**
- Il valore semantico definisce **il significato di un simbolo**
 - Un intero NUM: la costante associata al numero
 - Una variabile VAR: il puntatore alla locazione della variabile
 - Un'espressione expr: il valore dell'espressione
- Il tipo definisce **il tipo C** del valore associato al valore semantico

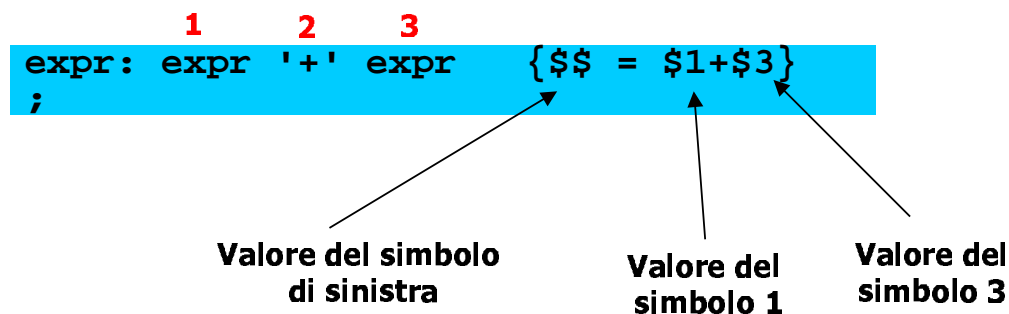
```
%union{
  double val;
  sybtbl *sptr;
}
%token <val> NUM
%token <sptr> VAR
%type <val> exp
. . . . .
```

Il tipo di espressioni e numeri

Il tipo delle variabili è un puntatore alla tabella dei simboli

Le produzioni

- Alle produzioni **sono associate azioni** (codice C) che ne definisco il significato
- Ogni volta che viene applicata la regola **si esegue il codice**
- L'azione permette di combinare i valori associati ai simboli per **calcolare il valore semantico del simbolo a sinistra** della produzione



Una calcolatrice

Definisce la
precedenza fra
gli operatori

Calcola il valore
di un'espressione

```
%union{
    double val;
    symtbl *sptr;}
%token <val> NUM
%token <sptr> VAR
%type <val> exp

%right '='
%left '+'
%left '*'

%%
input: /* vuota */
      | input exp '\n'
;

exp:  NUM { $$=$1; }
     | VAR { $$=$1->value.var; }
     | VAR '=' exp { $$=$3; $1->value.var=$3; }
     | exp '+' exp { $$=$1+$3; }
     | exp '*' exp { $$=$1*$3; }
     | '(' exp ')' { $$=$2; }
```

YACC e LEX

- Il valore semantico
 - nella calcolatrice è il valore dell'espressione
 - in compilatore è un albero sintattico
- Il valore dei simboli
 - **simboli terminali** viene calcolato da **LEX**
 - **simboli non terminali** viene calcolato da **YACC**
- YACC chiama automaticamente LEX attraverso una funzione (yyparse()) per fare l'analisi lessicale

