

ALTRI PARSER SR

# GRAMMATICHE LR(0)

## Grammatiche LR(0)


- E' una grammatica in cui ogni cella della tabella LR(0) contiene al più un solo valore.
- Equivalentemente, gli stati dell' automa o contengono solo produzioni che presuppongono shift o solo produzioni che presuppongono reduce.
- Gli stati che presuppongono operazioni di reduce, inoltre, contengono una sola produzione.

Proprietà dell'**automa LR(0)**:


1. tutte le frecce entranti in uno stato hanno la stessa etichetta;
2. Uno stato di reduce non ha successori;
3. Uno stato di shift ha almeno un successore.

# CONFLITTI SHIFT-REDUCE O REDUCE-REDUCE

Shift-reduce


$$\begin{array}{l} X \rightarrow X_1 \dots \cdot X_k \\ Y \rightarrow Y_1 \dots Y_i \cdot \end{array}$$

Reduce-Reduce


$$\begin{array}{l} X \rightarrow X_1 \dots X_k \cdot \\ Y \rightarrow Y_1 \dots Y_i \cdot \end{array}$$

In questi casi non può essere una grammatica LR(0).

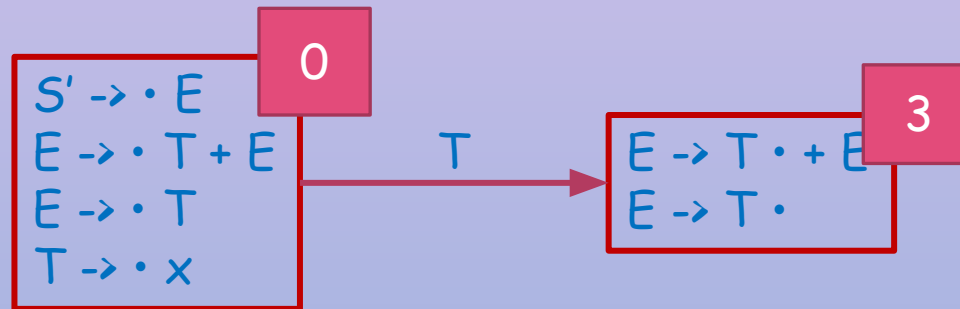
# ESEMPIO DI GRAMMATICA NON LR(0)

$S' \rightarrow E$

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow x$



Conflitto  
shift-reduce

# TABELLA AMBIGUA

(0)  $S' \rightarrow E$

(1)  $E \rightarrow T + E$

(2)  $E \rightarrow T$

(3)  $T \rightarrow x$

Dallo stato 1 leggendo "+" posso o shiftare sullo stato Closure( $E \rightarrow T + E$ ) o ridurre con  $E \rightarrow T$ .

Ovvero, il modello diventa non deterministico

Abbiamo bisogno di strumenti più potenti

	X	+	\$	E	T
1	S5			G2	G3
2			acc		
3	R2	S4, R2	R2		
4	S5			G6	G3
5	R3	R3	R3		
6	R1	R1	R1		

# ESERCIZIO

Data la grammatica seguente, stabilire se essa è LR(0).

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

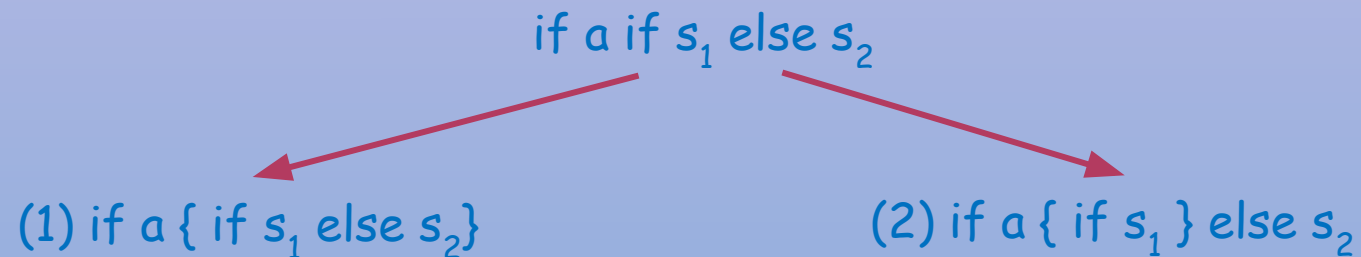
$R \rightarrow L$

# CONFLITTI SHIFT REDUCE

Molti SR parser risolvono automaticamente i conflitti shift/reduce privilegiando lo **shift al reduce** (ciò incorpora la regola dell'annidamento più vicino nel problema dell'else pendente). Per esempio una versione semplificata della grammatica è:

$S \rightarrow L \mid \text{other}$

$L \rightarrow \text{if } S \mid \text{if } S \text{ else } S$  (ambiguità in corrispondenza del token else)



Privilegiare lo shift, dà l'interpretazione 1, privilegiare il reduce dà la 2

# ESEMPIO DI CONFLITTO REDUCE/REDUCE PER PARSE LR(0)

La seguente grammatica modella statement che possono rappresentare chiamate a procedure senza parametri o assegnazione di espressioni a variabili

Stmt  $\rightarrow$  call-stmt | assign-stmt

Call-stmt  $\rightarrow$  identifier

Assign-stmt  $\rightarrow$  var := esp

Var  $\rightarrow$  identifier

Esp  $\rightarrow$  var | number



$S \rightarrow id \mid V := E$

$V \rightarrow id$

$E \rightarrow V \mid n$

$S' \rightarrow \cdot S$   
 $S \rightarrow \cdot id$   
 $S \rightarrow \cdot V := E$   
 $V \rightarrow \cdot id$

id

$S \rightarrow id \cdot$   
 $V \rightarrow id \cdot$

Conflitto  
reduce-reduce



# LL(K) VS. LR(K)

## LL(K)

Left to Right parse  
Leftmost derivation  
k-token look ahead

- Predice quale produzione usare dopo aver visto k tokens dalla stringa da derivare.
- Usati sia nei compilatori scritti a mano (discendenti ricorsivi) sia costruiti con strumenti automatici.
- Recentemente sono stati ripresi. **ANTLR** e **javacc** per Java
- Necessitano di modificare la grammatica (eliminare ricorsioni sinistre immediate e regole con testa uguale e regole di coda con prefissi in comune)

## LR(K)

Left to Right parse  
Rightmost derivation  
k-token look ahead

- Riesce a riconoscere le occorrenze del lato destro di una produzione avendo visto i primi k simboli di ciò che deriva da tale lato destro
- Usati tipicamente in modo automatico.
- I più usati per il parsing reale **YACC**, **BISON**, per C, **CUP**, **sablecc** per Java,
- Necessitano solo di aggiungere la produzione  $S' \rightarrow S$ .

# ESERCIZI

La grammatica

$S \rightarrow A \mid aS$

$A \rightarrow aAb \mid \varepsilon$

È LR(0)? È LL(1)?

Il linguaggio è deterministico  
ma non LL(k)

La presenza di  
regole vuote viola la  
condizione LR(0)

La grammatica

$S \rightarrow a \mid ab$

È LR(0)? È LL(1)?

In un linguaggio LR(0), se una  
stringa appartiene al  
linguaggio, nessun prefisso di  
essa può appartenervi (si  
determina un conflitto  
shift-reduce)

# ESERCIZIO

La grammatica

$S \rightarrow aSb \mid \varepsilon$

È LR(0)? E' LL(1)?

Non è LR(0)! (Perché?)  
E' LL(1)!

La grammatica

$S \rightarrow BA \mid A$

$A \rightarrow aAb \mid ab$

È LR(0)? E' LL(1)?

E' LR(0)! (Perché?)  
Non è LL(1)! (Perché?)

Che conclusioni trarre sulle relazioni tra grammatiche LR(0) e LL(1)?

# CONFRONTO TRA GRAMMATICHE E LINGUAGGI LR(0) E LL(1)

- Le classi di grammatiche LR(0) e LL(1) non sono incluse una nell'altra
  - ✗ Una grammatica con regole vuote non è LR(0) ma può essere LL(1)
  - ✗ Una grammatica con ricorsioni sinistre non è LL(1) ma può essere LR(0)
- Le famiglie dei linguaggi LR(0) e LL(1) sono distinte e incomparabili.
  - ✗ Un linguaggio chiuso per prefissi non è LR(0) ma può essere LL(1)
  - ✗ Esistono linguaggi LR(0) ma non LL(1)

ALTRI PARSER SR

# PARSER LR(0), SLR, LR(1), LALR(1): COSA HANNO IN COMUNE?

- Usano azioni di **shift e reduce**;
- Sono **macchine guidate da una tabella**:
  - sono **raffinamenti di LR(0)**
    - Calcolano un FSA usando la costruzione basata sugli item
    - **SLR**: usa gli **stessi item di LR(0)** e usa anche le informazioni dell'insieme **follow**
    - **LR(1)/LALR(1)**: un item contiene anche informazioni date dai simboli **lookahead**.
      - LALR(1) è una semplificazione di LR(1) per ridurre il numero degli stati
- Consentono di definire classi di grammatiche

Se il parser LR(0) (o SLR, LR(1), LALR(1)) calcolato dalla grammatica non ha conflitti shift/reduce o reduce/reduce, allora  $G$  è per definizione una grammatica LR(0) (o SLR, LR(1), LALR(1)).

# PANORAMICA SU LR PARSING

Le grammatiche **LR** sono più potenti delle **LL**.

**LR(0)** ha esclusivo interesse didattico.

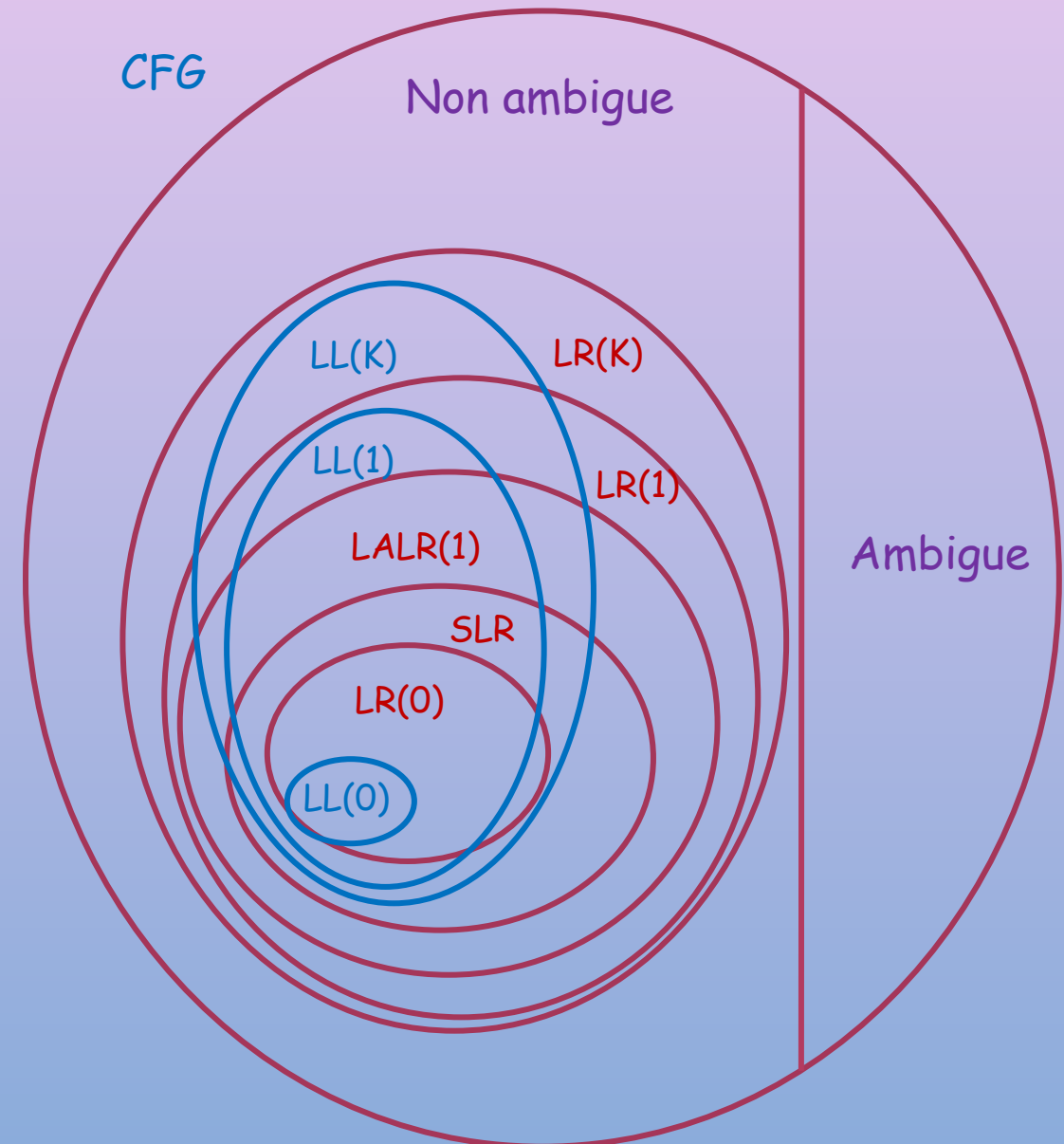
Esiste una classe di parser chiamato **Simple LR** (o **SLR**) che consentono il parsing di una famiglia un po' più vasta di Linguaggi.

La maggior parte dei linguaggi di programmazione ammettono una grammatica **LALR(1)**.

Molti generatori di parser usano questa classe.

**LR(1)** fornisce un parsing molto potente, ma l'implementazione è poco controllabile.

Si cercano grammatiche **LALR(1)** equivalenti.



PARSER SLR(1)



# SLR(1)

I parser SLR che considerano un solo simbolo di Lookahead si chiamano **SLR(1)**. Tuttavia per semplicità sono detti semplicemente **SLR**.

**SLR** sta per **SIMPLE LR**.

Il metodo è molto simile a quello utilizzato per le grammatiche LR(0), ma nella costruzione della tabella di parsing (**tabelle SLR**) si tiene conto degli elementi FOLLOW.

Un **parser SLR** sarà un parser che utilizza tabelle SLR.

Una **grammatica** si dice **SLR** se ogni casella della tabella SLR contiene al più una regola.

# METODO SLR

Data una grammatica  $G$ , si costruisce la grammatica  $G'$  aumentata della regola  $s' \rightarrow s$ , dove  $s'$  è un nuovo simbolo non terminale non appartenente alla grammatica.

Occorre costruire gli **item** della forma usata nel parsing LR(0) e considerare sottoinsiemi dell'insieme di tutti gli item

Occorre inoltre conoscere l'insieme **FOLLOW(A)** per ogni simbolo **non terminale A**.

La costruzione degli elementi **ACTION** e **GOTO** della tabella sono costruiti mediante il seguente algoritmo

# ALGORITMO TABELLA SLR

**INPUT:** una grammatica aumentata  $G'$

**OUTPUT:** le azioni action e goto della tabella di parsing SLR

1. Siano  $I_1, I_2, \dots, I_n$  gli insiemi di item LR(0) di  $G'$
2. Si costruisce lo stato  $i$  a partire da  $I_i$ : le azioni di parsing conseguenti sono:
  - A. Se  $[A \rightarrow \alpha . a \beta]$  appartiene a  $I_i$ , si assegna ad  $\text{ACTION}[i, a]$  il valore **SHIFT**  $j$  dove  $j$  è lo stato associato a  $I_j = \text{CLOSURE}(A \rightarrow \alpha a . \beta)$
  - B. Se  $[A \rightarrow \alpha .]$  appartiene a  $I_i$  allora si assegna ad  $\text{ACTION}[i, a]$  il valore **REDUCE**  $A \rightarrow \alpha$ .  
per ogni terminale  $a$  appartenente a  $\text{FOLLOW}(A)$ .  $A$  non può essere  $S'$
  - C. Se  $[S' \rightarrow S .]$  appartiene a  $I_i$ , allora si assegna ad  $\text{ACTION}[i, \$]$  il valore **ACCEPT**
3. Le transizioni **GOTO**: se  $\text{GOTO}(I_i, A) = I_j$ , allora  $\text{GOTO}[i, A] = j$
4. A tutti gli elementi non definiti si assegna il valore **ERROR**
5. Lo **stato iniziale** del parser è quello che contiene l'item  $S' \rightarrow S$ .

# OSSERVAZIONI SU SLR

In pratica l'unica differenza con la tabella LR0 è che l'azione **Reduce  $A \rightarrow \alpha$**  non viene associata a tutti i simboli terminali, ma è condizionata dal simbolo letto nell'input. Se il simbolo letto dalla testina di input è in  $\text{Follow}(A)$  significa che un'azione reduce può portare al riconoscimento

E' chiaro che tutte le grammatiche LR(0) sono SLR (non hanno conflitti shift-reduce a prescindere dall'insieme follow), ma non viceversa (esempio seguente).

SLR migliora le euristiche shift/reduce LR(0)

- Si riduce la probabilità che ci siano degli stati con conflitti shift/reduce (l'azione reduce non è in corrispondenza di tutti i simboli)
- **se ci sono conflitti, la grammatica non è SLR**

# ESEMPIO DI PARSING SLR

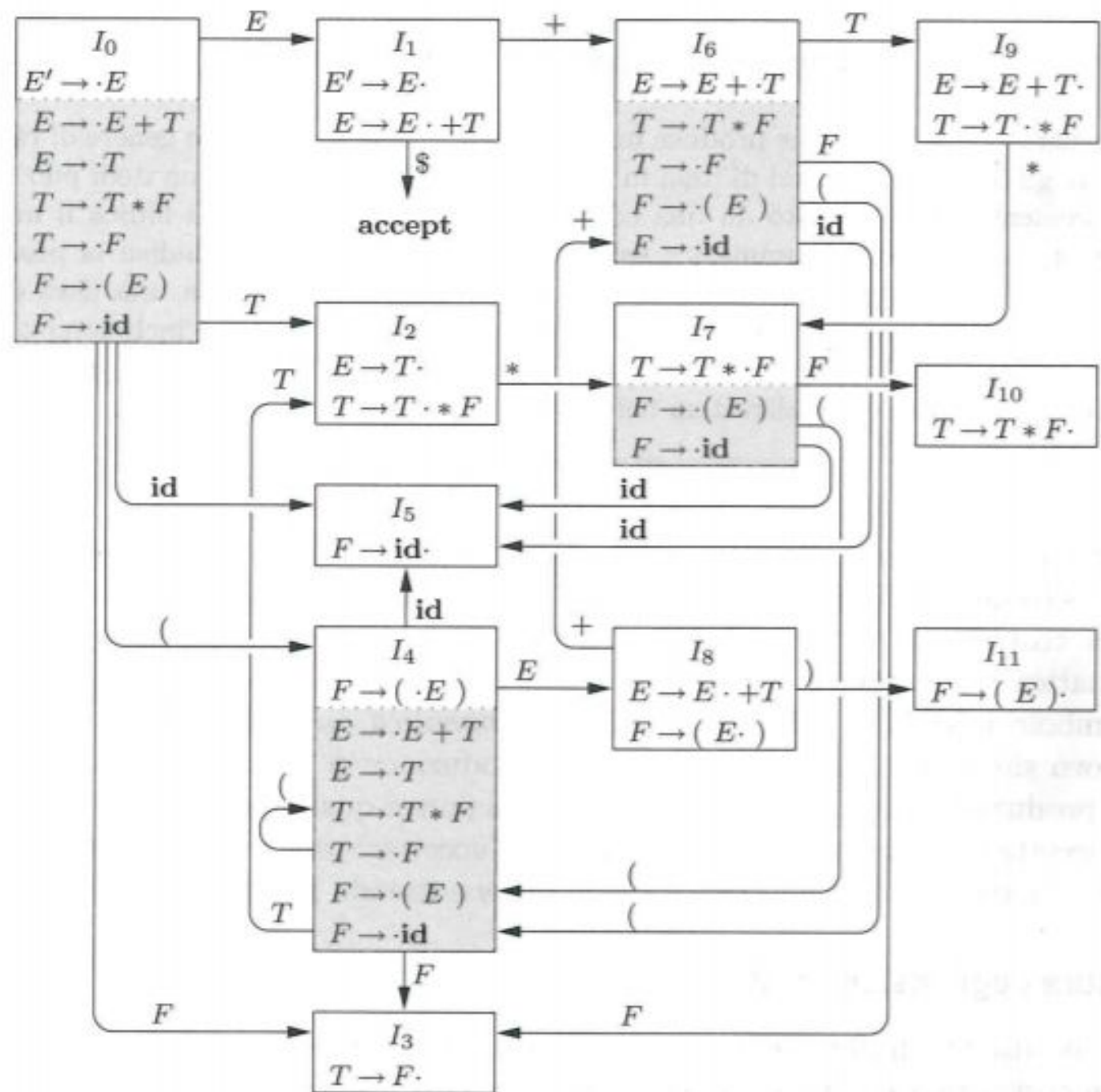
$G$ :

$$\begin{aligned} E &\rightarrow E+T \\ E &\rightarrow T \\ T &\rightarrow T*F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow id \end{aligned}$$


$G'$ :

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E+T \\ E &\rightarrow T \\ T &\rightarrow T*F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow id \end{aligned}$$

$FOLLOW(E') = \{\$ \}$   
 $FOLLOW(E) = \{\$, +, )\}$   
 $FOLLOW(T) = \{\$, +, ), *\}$   
 $FOLLOW(F) = \{\$, +, ) *\}$





	+	*	(	)	id	\$	E	T	F
$I_0$			Shift 4		Shift 5		Goto 1	Goto 2	Goto 3
$I_1$	Shift 6					ACCEPT			
$I_2$	Red $E \rightarrow T$	Shift 7	Red $E \rightarrow T$			Red $E \rightarrow T$			
$I_3$	Red $T \rightarrow F$	Red $T \rightarrow F$		Red $T \rightarrow F$		Red $T \rightarrow F$			
$I_4$			Shift 4		Shift 5		Goto 8	Goto 2	Goto 3
$I_5$	Red $F \rightarrow id$	Red $F \rightarrow id$		Red $F \rightarrow id$		Red $F \rightarrow id$			
$I_6$			Shift 4		Shift 5			Goto 9	Goto 3
$I_7$			Shift 4		Shift 5				Goto 10
$I_8$	Shift 6			Shift 11					
$I_9$	Red $E \rightarrow E+T$	Shift 7		Red $E \rightarrow E+T$		Red $E \rightarrow E+T$			
$I_{10}$	Red $T \rightarrow T * F$	Red $T \rightarrow T * F$		Red $T \rightarrow T * F$		Red $T \rightarrow T * F$			
$I_{11}$	Red $F \rightarrow (E)$	Red $F \rightarrow (E)$		Red $F \rightarrow (E)$		Red $F \rightarrow (E)$			



In alcuni casi la scelta fra shift e reduce dipende dal simbolo successivo dell'input. E' per questo che occorre considerare la funzione FOLLOW

# GRAMMATICHE NON SLR

Molte grammatiche non sono SLR. Sicuramente tutte le grammatiche ambigue non sono SLR.

Possiamo fare il parsing di un maggior numero di grammatiche introducendo **istruzioni che implementano dichiarazioni di precedenza**:

**ESEMPIO:** Consideriamo la grammatica

$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$

Questa grammatica non è SLR poiché il DFA per questa grammatica contiene uno stato con un conflitto shift/reduce:

$E \rightarrow E * E.$      $E \rightarrow E. + E$

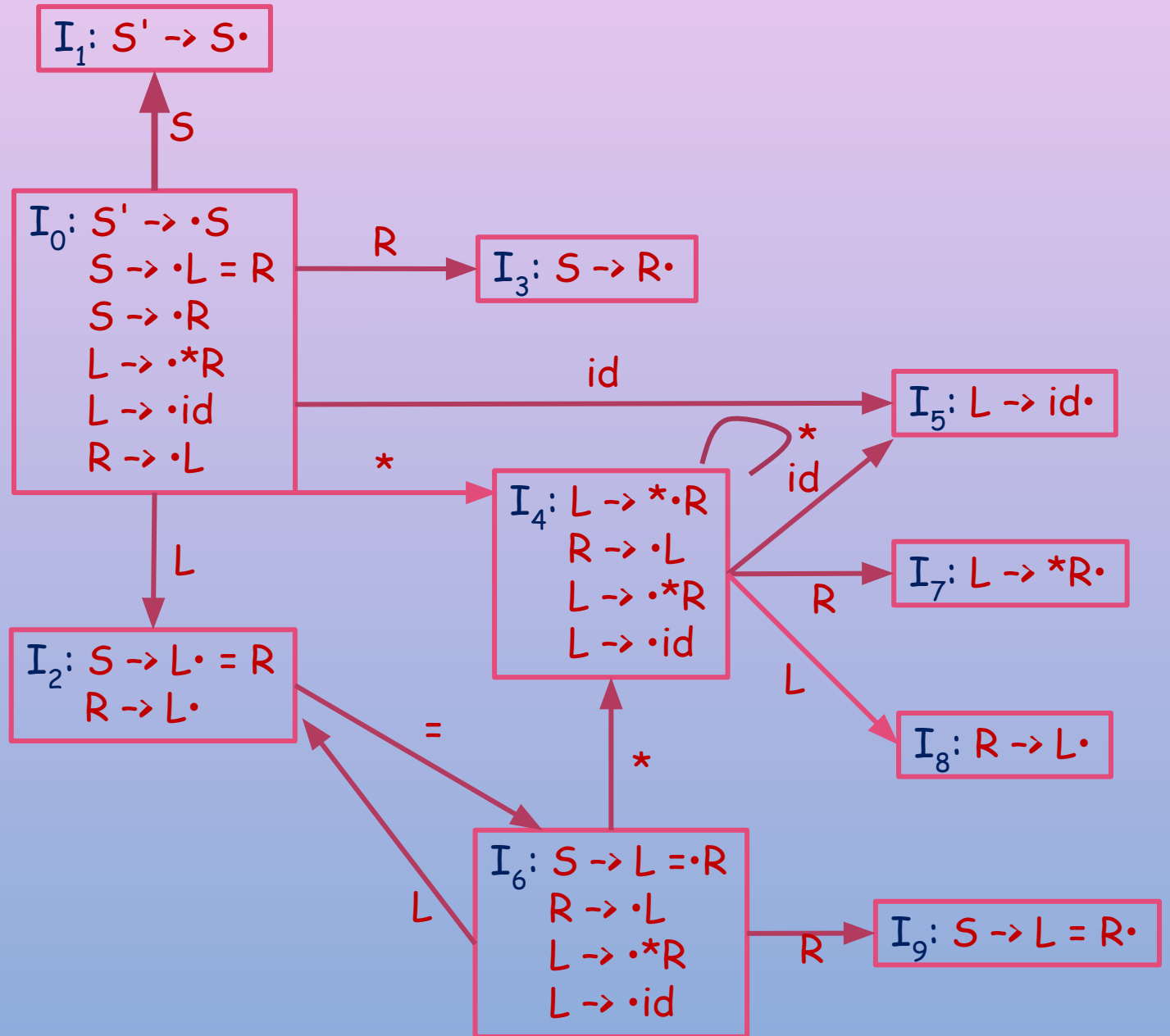
Dichiarare che  $*$  ha precedenza più alta rispetto a  $+$  risolve questo conflitto in favore della riduzione

Tutte le grammatiche SLR sono non ambigue. Tuttavia esistono grammatiche non ambigue che non sono SLR

$S' \rightarrow S$   
 $S \rightarrow L = R \mid R$   
 $R \rightarrow L$   
 $L \rightarrow *R \mid id$

$FOLLOW(S') = \{\$ \}$   
 $FOLLOW(S) = \{\$ \}$   
 $FOLLOW(R) = \{\$, =\}$   
 $FOLLOW(L) = \{\$, =, \}$

Lo stato  $I_2$  ha un conflitto **shift ( $I_6$ )-reduce ( $R \rightarrow L$ )** alla lettura del simbolo =





# ALTRI METODI BASATI SU LR

Il metodo SLR non copre un grande spettro di grammatiche. Vediamo se esistono metodi più generali

## **Metodo LR(1)**

Usa gli item LR(1), molti di più degli item LR(0)

## **Metodo LALR (lookahead LR)**

Usa gli item LR(0) a cui aggiunge simboli di input. Può essere usato su più grammatiche rispetto al metodo SLR.

Questo è il metodo più usato coprendo un insieme di grammatiche abbastanza ricco senza incorrere nelle inefficienze del metodo LR canonico