

PARSING BOTTOM-UP E PARSER LR(0)

PARSER ASCENDENTI

Gli algoritmi di parsing bottom-up o ascendente analizzano una stringa di input cercando di ricostruire i passi di una **derivazione rightmost**.

Sono detti bottom-up perchè costruiscono l'albero sintattico relativo ad una stringa prodotta dalla grammatica dalle foglie alla radice.

Un modello generale di parsing bottom-up è il **parsing shift-reduce**, chiamato comunemente **SR-parsing**.

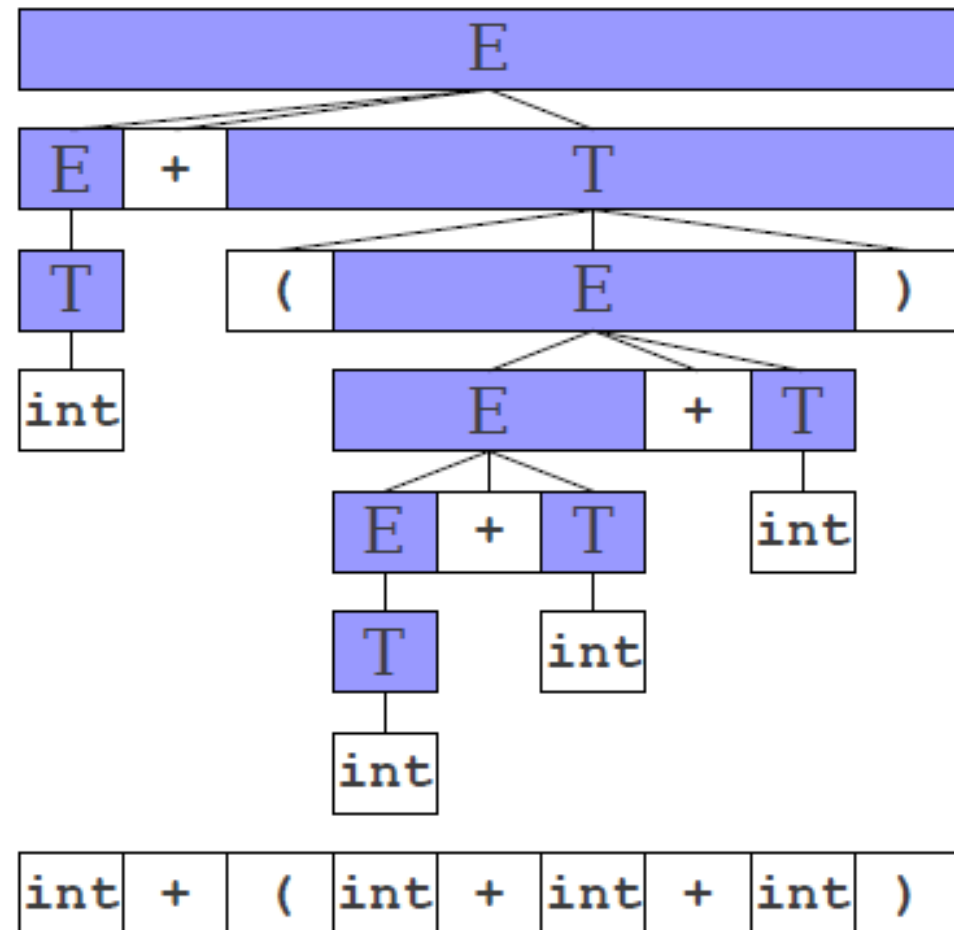
La classe più grande di grammatiche per cui è possibile usare un SR-parsing è data dalle grammatiche **LR**. Costruire un parser LR a mano è molto difficile ma tale metodo è utile nel caso di generazione automatica di parser.

ESEMPIO

$E \rightarrow T$	$\text{int} + (\text{int} + \text{int} + \text{int})$
$E \rightarrow E + T$	$\Rightarrow T + (\text{int} + \text{int} + \text{int})$
$T \rightarrow \text{int}$	$\Rightarrow E + (\text{int} + \text{int} + \text{int})$
$T \rightarrow (E)$	$\Rightarrow E + (T + \text{int} + \text{int})$
	$\Rightarrow E + (E + \text{int} + \text{int})$
	$\Rightarrow E + (E + T + \text{int})$
	$\Rightarrow E + (E + \text{int})$
	$\Rightarrow E + (E + T)$
	$\Rightarrow E + (E)$
	$\Rightarrow E + T$
	$\Rightarrow E$

CREAZIONE DEL PARSE TREE

`int + (int + int + int)`
⇒ `T + (int + int + int)`
⇒ `E + (int + int + int)`
⇒ `E + (T + int + int)`
⇒ `E + (E + int + int)`
⇒ `E + (E + T + int)`
⇒ `E + (E + int)`
⇒ `E + (E + T)`
⇒ `E + (E)`
⇒ `E + T`
⇒ `E`



PARSING BOTTOM UP

Obiettivo:

Costruire un parse tree di una stringa in input partendo dalle foglie fino ad arrivare alla radice. Questo processo può essere pensato come una **riduzione** di una stringa **all'assioma della grammatica**.

Metodo (intuitivo):

ad ogni passo di riduzione una particolare sottostringa che ha un match con la parte destra di una qualche regola di produzione viene sostituita con la parte sinistra di quella produzione. Se la sottostringa è scelta in modo corretto, allora viene così prodotta una derivazione rightmost in ordine inverso.

ESEMPIO

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

La sequenza **abbcede** può essere ridotta ad S nel modo seguente (indichiamo in rosso la sottostringa corrispondente alla parte destra di una derivazione e sottolineiamo in non terminale appena determinato da una produzione inversa):

abbcede \rightarrow a**Abc**de \rightarrow a**A**de \rightarrow a**AB**e \rightarrow S

Handle: è una sottostringa di una forma sentenziale destra che coincide con la parte destra di una produzione e la cui riduzione rappresenta un passo della inversa della derivazione rightmost (i simboli che abbiamo indicato in rosso)

Un handle può essere seguito solo da simboli terminali.

PARSER SHIFT-REDUCE (SR)

Il parser usa una **pila**, che contiene inizialmente il simbolo "\$" (fondo della Pila), ed un **input**, la cui fine è marcata dal simbolo "\$" (è l'eof generato dallo scanner).

Esempio:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Pila	input	azione
\$	abbcde\$	

- Nello stato iniziale la pila è vuota
- Il parser esegue azioni, di quattro tipi:

Shift: un simbolo terminale è spostato dalla stringa di input sulla pila (si indica scrivendo la parola **shift**)

Reduce: una stringa α in cima alla pila è ridotta al (sostituita con il) non-terminale A , secondo la regola $A \rightarrow \alpha$ (si indica scrivendo **reduce $A \rightarrow \alpha$**) che corrisponde al un passo di derivazione al contrario.

Error: errore di sintassi

Accept: la pila ha in cima il solo simbolo S (contiene solo $\$S$) e la lettura dell'input è terminata (input col solo simbolo $\$$)

ESEMPIO

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

Parsing della stringa `abbcdce$`

Prefissi ammissibili: insieme dei prefissi delle forme sentenziali destre che possono apparire sullo stack di uno SR parser. Ovvero, sono i prefissi delle forme sentenziali destre di derivazioni rightmost, che non contengono sottostringhe interne che sono handle (tali sottostringhe possono apparire solo come suffisso).

Come riconoscere un handle sullo stack?
Come scegliere la produzione o l'azione opportuna?

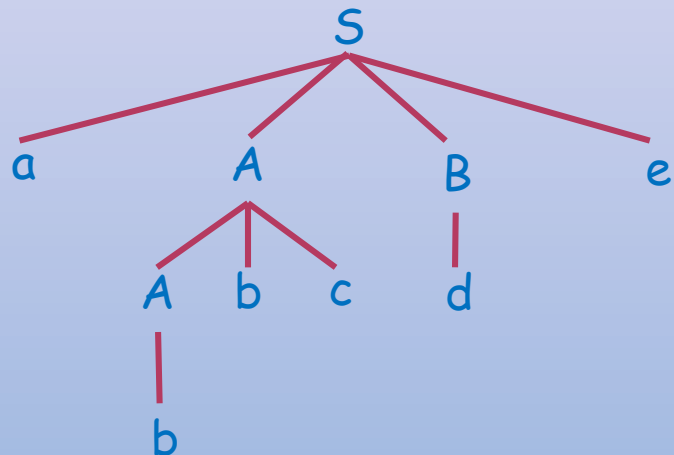
	PILA	INPUT	AZIONE
1	\$	abbcdce\$	shift
2	\$a	bbcdce\$	shift
3	\$a <u>b</u>	bbcdce\$	reduce $A \rightarrow b$
4	\$aA	bbcdce\$	shift
5	\$aAb	bbcdce\$	shift
6	\$a <u>Abc</u>	bbcdce\$	reduce $A \rightarrow Abc$
7	\$aA	bbcdce\$	shift
8	\$aA <u>d</u>	bbcdce\$	reduce $B \rightarrow d$
9	\$aAB	bbcdce\$	shift
10	\$aABe	bbcdce\$	reduce $S \rightarrow aABe$
11	\$S	bbcdce\$	accept

ESEMPIO

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

Ricostruzione dell'albero di derivazione

Ripercorrendo dal basso verso l'alto tutte le regole reduce applicate, si può facilmente ricostruire l'albero di derivazione (da destra verso sinistra)



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

	PILA	INPUT	AZIONE
1	\$	abbcde\$	shift
2	\$a	bbcd\$	shift
3	\$a <u>b</u>	bcde\$	reduce $A \rightarrow b$
4	\$aA	bcde\$	shift
5	\$aAb	cde\$	shift
6	\$a <u>Abc</u>	de\$	reduce $A \rightarrow Abc$
7	\$aA	de\$	shift
8	\$aA <u>d</u>	e\$	reduce $B \rightarrow d$
9	\$aAB	e\$	shift
10	\$aABe	\$	reduce $S \rightarrow aABe$
11	\$S	\$	accept

GRAMMATICHE LR

Se il parser di una grammatica può essere implementato in maniera deterministica con un algoritmo shift-reduce, allora la grammatica si dice **LR**.

Il parsing shift-reduce effettua un'operazione di **reduce** quando il top dello stack corrisponde al lato destro di una regola della grammatica

Comunque se la grammatica non è LR ci possono essere istanze per cui questa non è l'operazione corretta (sarebbe opportuno eseguire uno shift) oppure ci possono essere istanze per cui non è chiaro quale operazione reduce applicare.

CONFLITTI SHIFT-REDUCE

$S \rightarrow SaB$

$S \rightarrow a$

$B \rightarrow ab$

Se si vuole effettuare il parsing della stringa aaab

	PILA	INPUT	AZIONE
1	\$	aaab\$	shift
2	\$a	aab\$	Shift o reduce $S \rightarrow a$?

	PILA	INPUT	AZIONE
1	\$	aaab\$	shift
2	\$a	aab\$	<u>reduce $S \rightarrow a$</u>
3	\$S	aab\$	shift
4	\$Sa	ab\$	reduce $S \rightarrow a$
5	\$SS	ab\$	shift
6	\$SSa	b\$	shift
7	\$SSab	\$	reduce $B \rightarrow ab$
8	\$SSB	\$	errore

	PILA	INPUT	AZIONE
1	\$	aaab\$	shift
2	\$a	aab\$	<u>reduce $S \rightarrow a$</u>
3	\$S	aab\$	shift
4	\$Sa	ab\$	<u>shift</u>
5	\$Saa	b\$	shift
6	\$Saab	\$	reduce $B \rightarrow ab$
7	\$SaB	\$	reduce $S \rightarrow SaB$
8	\$S	\$	accept

CONFLITTI REDUCE-REDUCE

$S \rightarrow SA$

$S \rightarrow a$

$A \rightarrow a$

Fare il parse della stringa aa

	PILA	INPUT	AZIONE
1	\$	aa\$	shift
2	\$a	a\$	reduce $A \rightarrow a$ o reduce $S \rightarrow a$?

	PILA	INPUT	AZIONE
1	\$	aa\$	shift
2	\$a	a\$	<u>reduce $A \rightarrow a$</u>
3	\$A	a\$	shift
4	\$Aa	\$	reduce $S \rightarrow a$
5	\$AS	\$	errore

	PILA	INPUT	AZIONE
1	\$	aa\$	shift
2	\$a	a\$	<u>reduce $S \rightarrow a$</u>
3	\$S	a\$	shift
4	\$Sa	\$	reduce $A \rightarrow a$
5	\$SA	\$	reduce $S \rightarrow SA$
6	\$S	\$	accept

GRAMMATICHE LR(K)

Nei due casi precedenti il conflitto può essere risolto guardando un simbolo dell'input in avanti

Un algoritmo LR che guarda avanti k simboli dell'input per risolvere i conflitti si chiama LR(k)

In generale quando si implementa un linguaggio di programmazione bottom-up, definiamo il linguaggio con grammatiche LR(1)

Le grammatiche ambigue non possono essere LR(k) per nessun k

ESEMPIO

Esistono grammatiche CF per le quali il parsing SR non può essere usato. In questi casi ogni SR-parser può raggiungere una configurazione in cui sfruttando l'intero stack e il simbolo da leggere non si riesce a decidere se eseguire uno shift o un reduce, o non si sa quale riduzione applicare.

Esempio: una grammatica ambigua non può essere LR

Stmt → if expr then stmt

| If expr then stmt else stmt

| Other

Pila	input
...If expr then stmt	else ...\$

Shift
O
Reduce?

PARSING LR(K)

Il termine LR(k) ha il seguente significato:

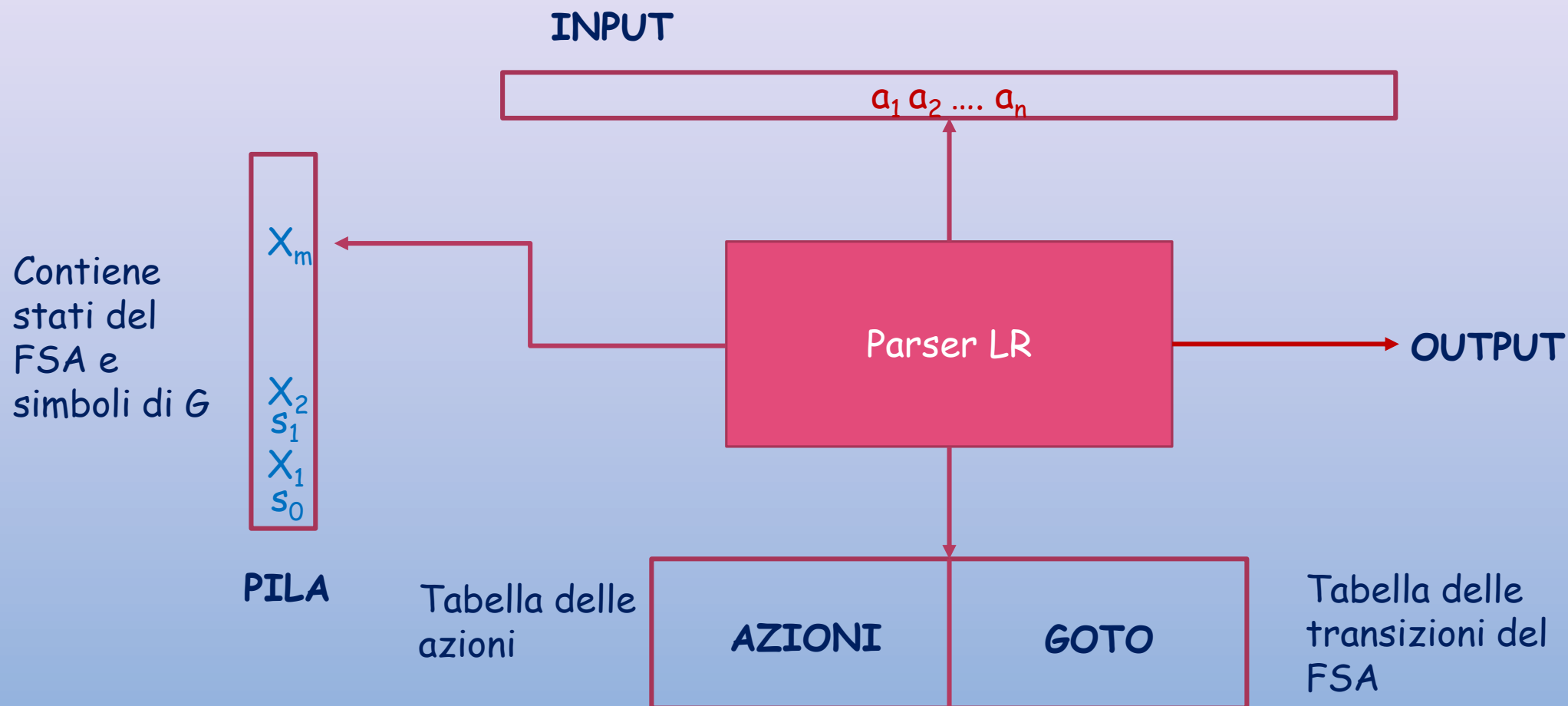
1. La **L** significa che l'input è analizzato da **sinistra verso destra**
2. La **R** significa che il parser produce una **derivazione rightmost** per la stringa di input
3. Il numero **k** significa che l'algoritmo utilizza **k simboli dell'input per decidere l'azione del parser.**

PERCHÈ USARE I PARSER LR?

- Possono essere costruiti per riconoscere virtualmente tutti i costrutti di linguaggi di programmazione definiti da Grammatiche CF.
- E' il più generale parsing SR che non richiede Backtracking.
- Riconosce velocemente gli errori sintattici in una scansione da sinistra a destra dell'input
- La classe delle grammatiche **LR contiene propriamente le Grammatiche LL**. Infatti un parser LR(k) deve riconoscere l'occorrenza di una parte destra di una produzione in una forma sentenziale destra con k simboli di prospezione. Un parser LL(k) deve scegliere una produzione in base ai primi k simboli della stringa da derivare.
- Scrivere un parser LR a mano è difficile, ma esistono generatori automatici.

SCHEMA DI UN PARSER LR

Si costruisce il FSPDA che riconosce l'insieme dei prefissi ammissibili. Sia $s_0, s_1, s_2, \dots, s_p$ il suo insieme degli stati (ciascuno di essi rappresenta lo stato della pila).



PARSER LR(0)

I **parser LR(0)** analizzano la stringa di input considerando solamente il simbolo in testa alla pila.

La classe delle grammatiche corrispondenti non è interessante, ma la tecnica sì. Il comportamento del parser **LR(0)** si basa sulla tabella **LR(0)**, come quella a fianco

ACTION						GOTO	
	()	x	,	\$	S	L
1	S3		S2			G4	
2	R2	R2	R2	R2	R2		
3	S3		S2			G7	G5
4					acc		
5		S6		S8			
6	R1	R1	R1	R1	R1		
7	R3	R3	R3	R3	R3		
8	S3		S2			G9	
9	R4	R4	R4	R4	R4		

PER LE GRAMMATICHE LR

La tecnica può essere estesa a tutte le grammatiche LR.

Una grammatica è LR quando un parser shift-reduce è in grado di riconoscere le Handle quando appaiono in cima allo stack (l'informazione è contenuta nella tabella).

Un parser LR non deve guardare tutto lo stack: lo stato che in ogni momento si trova in testa ad esso contiene tutta l'informazione di cui il parser ha bisogno

Fatto importante: se è possibile riconoscere una handle guardando solo i simboli della grammatica che sono sullo stack allora esiste un automa finito che, leggendo i simboli dello stack, determina se una certa handle è presente in testa

La funzione GOTO di un parser LR è essenzialmente questo automa finito

Lo stato in testa allo stack è esattamente lo stato in cui l'automa si troverebbe se lo si facesse partire dallo stato iniziale dopo aver letto la stringa composta dai simboli dello stack che si trovano dal fondo alla testa

COSTRUZIONE DELLA TABELLA LR

Un LR(0)-item di una grammatica G è una produzione di G insieme con un punto in una posizione della parte destra. Ad esempio, data la produzione $A \rightarrow XYZ$, gli item sono:

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

La produzione $A \rightarrow \epsilon$ genera l'item $A \rightarrow .$

Un item indica quale parte di una produzione è stata vista ad una certa fase del processo di parsing.

Insiemi di item costituiscono gli stati dell'automa che riconosce i prefissi ammissibili.

Tale costruzione è alla base di tutti i parser LR.

AUTOMA LR(0)

1° passo: Si dota la grammatica della produzione $S' \rightarrow S$ (grammatica aumentata)

Operazione CLOSURE:

se I è un insieme di item, allora $CLOSURE(I)$ si costruisce come segue:

1. $I \subseteq CLOSURE(I)$;
2. Se $A \rightarrow \alpha.B\beta$ è in $CLOSURE(I)$ con B simbolo non terminale e $B \rightarrow \gamma$ è una produzione, allora si aggiunge $B \rightarrow \cdot\gamma$ in $CLOSURE(I)$. Si applica questa regola fino a quando non si possono aggiungere altri item.

```
Function Closure(I) ;  
begin  
  J:=I;  
  repeat  
    for each item  $A \rightarrow \alpha.B\beta$  in J and  
    each  $B \rightarrow \gamma$  such that  $B \rightarrow \gamma$  is not in  
    J do add  $B \rightarrow \cdot\gamma$  to J;  
  until no more items can be added to J;  
end
```

A partire da $Closure(S')$, si costruiscono vari insiemi $closure(I)$ che costituiranno gli stati dell'automa

CLOSURE (I)

Intuitivamente, il fatto che $A \rightarrow \alpha . B \beta$ appartenga a $CLOSURE(I)$, significa che a un certo punto del processo di parsing ci si aspetta di riconoscere in ingresso una sottostringa derivabile da $B \beta$.

Tale sottostringa avrà un prefisso derivabile da B applicando una produzione di B .
Aggiungiamo quindi tutti gli item relativi a B del tipo $B \rightarrow . \gamma$

ESEMPIO:

data la seguente grammatica, il primo stato dell'automa è **CLOSURE** ($S' \rightarrow \cdot S$)

$S' \rightarrow S$

$S \rightarrow (L)$

$S \rightarrow x$

$L \rightarrow S$

$L \rightarrow L, S$

$S' \rightarrow \cdot S$
 $S \rightarrow \cdot (L)$
 $S \rightarrow \cdot x$

CLOSURE ($S' \rightarrow \cdot S$)

AUTOMA LR(0)

Operazione **GOTO(I, X)** :

Se I è un insieme di item e X un simbolo (terminale o non terminale) di G , allora **GOTO(I, X)** si definisce come la chiusura dell'insieme di tutti gli item $A \rightarrow \alpha X \beta$ tale che $A \rightarrow \alpha \cdot X \beta$ è un item di I .

Intuitivamente se I è l'insieme degli item validi per un prefisso riducibile γ , allora **GOTO(I, X)** è l'insieme degli item validi per γX .

L'operazione **GOTO** permette di costruire le transizioni dell'automa

```
Function GOTO(I, X) ;  
begin  
  J:=emptyset;  
  for each item  $A \rightarrow \alpha \cdot X \beta$  in I do  
    add  $A \rightarrow \alpha X \beta$  to J;  
  Return CLOSURE(J) ;  
end
```


ESERCIZIO

Costruire l'automa di parsing per la grammatica

$S' \rightarrow S$

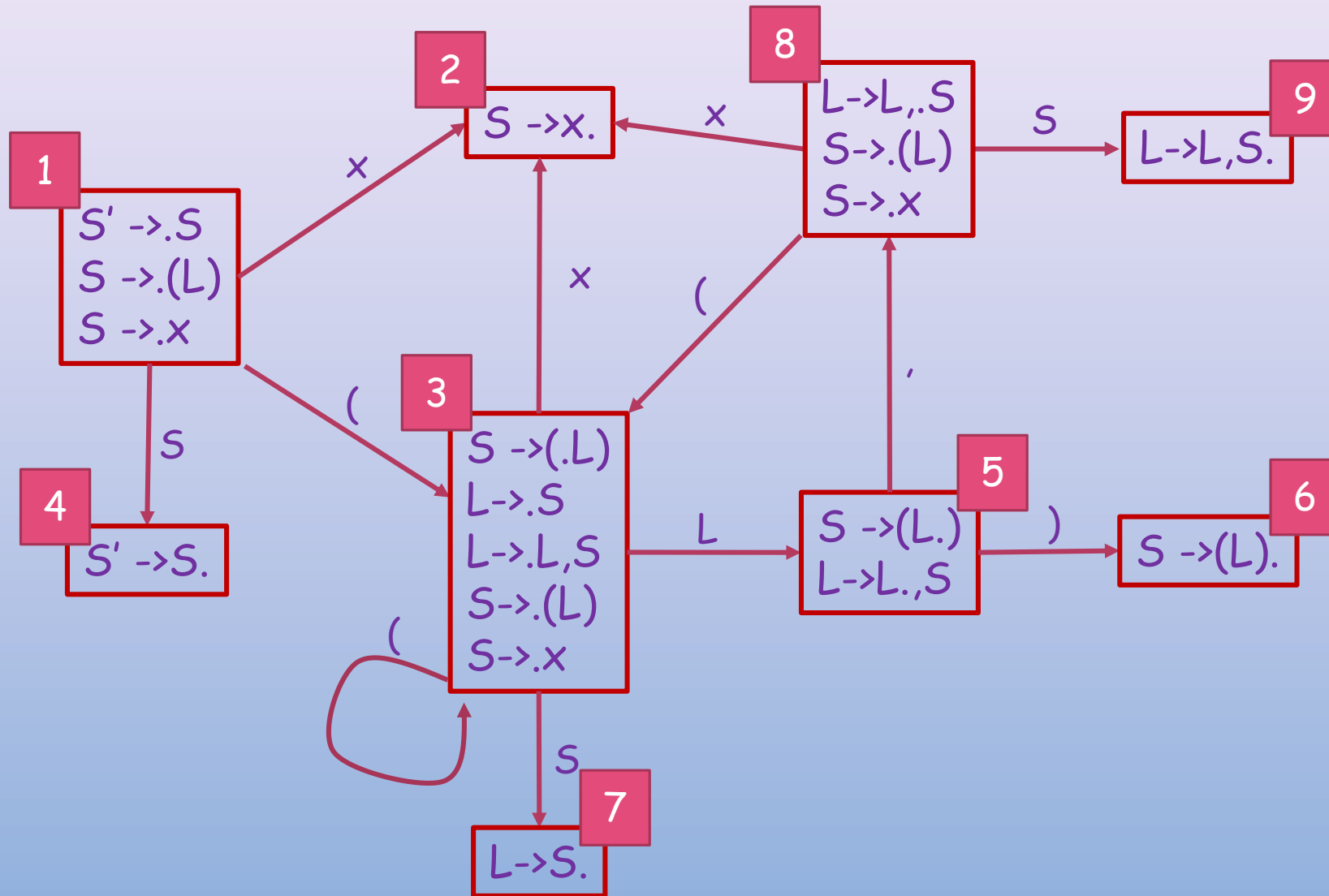
$S \rightarrow (L)$

$S \rightarrow x$

$L \rightarrow S$

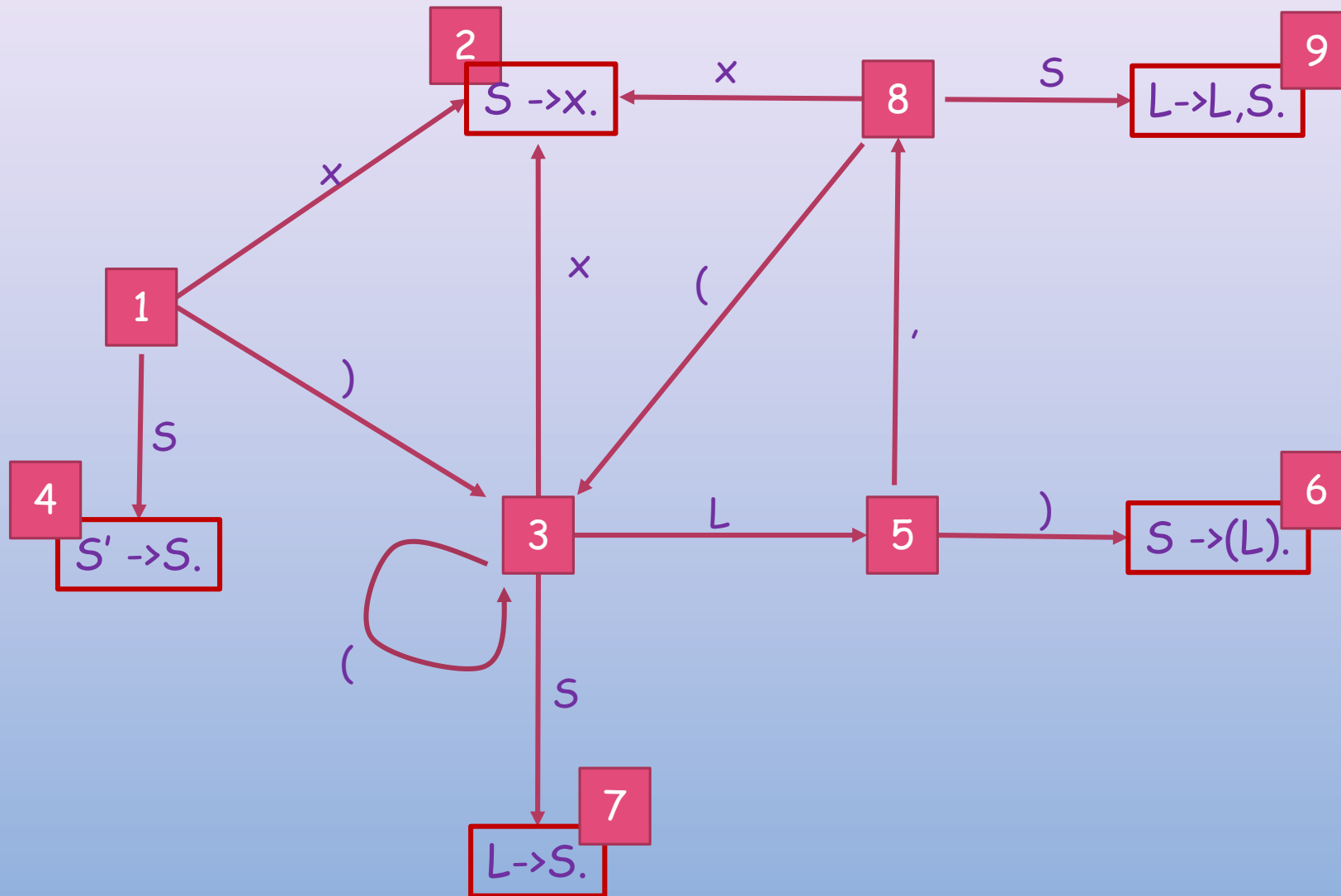
$L \rightarrow L, S$

ESEMPIO



- (0) $S' \rightarrow S$
- (1) $S \rightarrow (L)$
- (2) $S \rightarrow x$
- (3) $L \rightarrow S$
- (4) $L \rightarrow L, S$

ESEMPIO



- (0) $S' \rightarrow S$
- (1) $S \rightarrow (L)$
- (2) $S \rightarrow x$
- (3) $L \rightarrow S$
- (4) $L \rightarrow L, S$

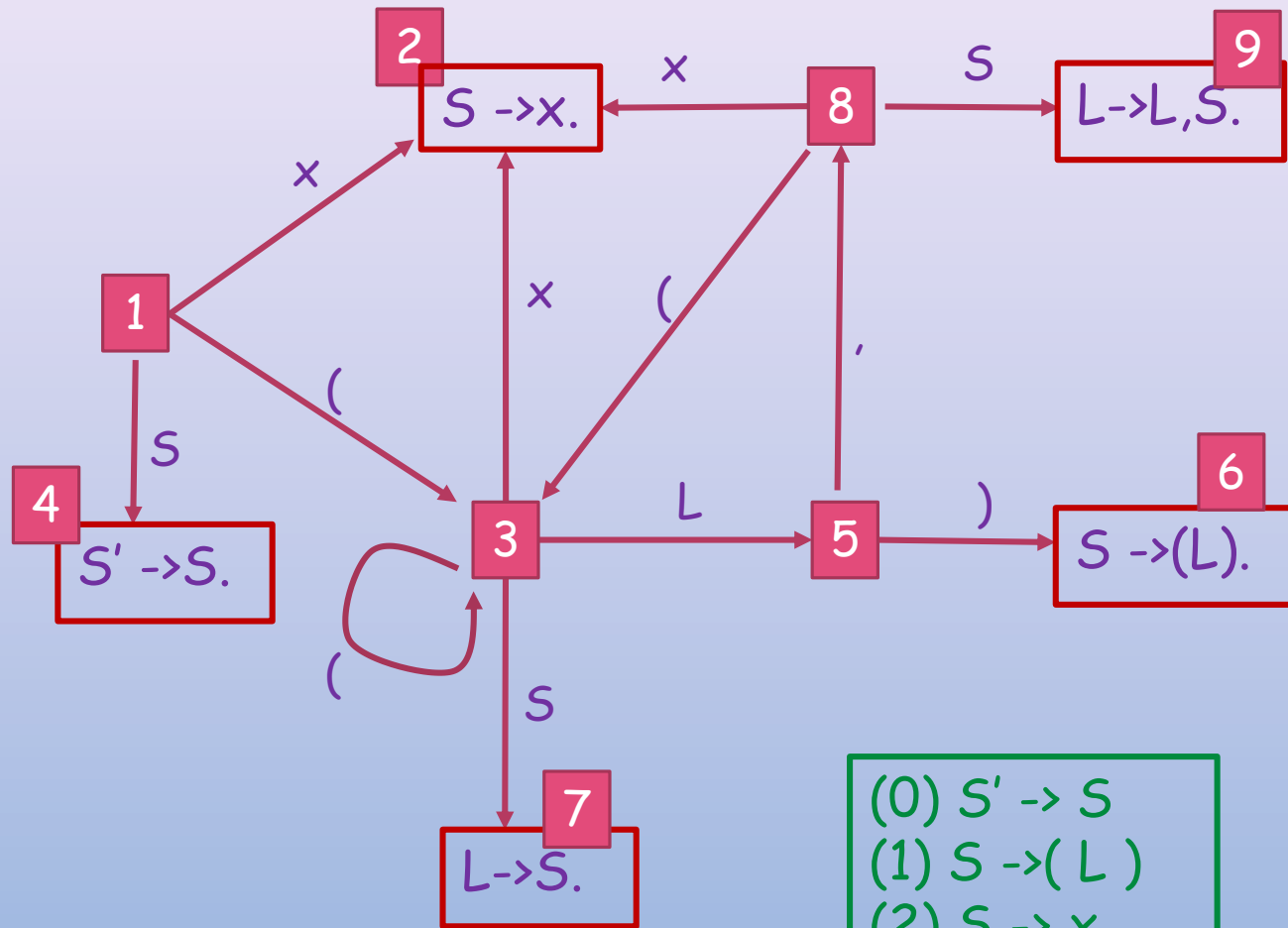
Il contenuto degli stati che non contengono items del tipo $x \rightarrow y.$ può essere dimenticato

ALGORITMO PER LA COSTRUZIONE DELLE TABELLE DI PARSING ACTION E GOTO

Indichiamo con i numeri $1, 2, \dots, n$ gli stati dell'automa. Per comodità enumeriamo anche le regole di produzione. Sia M la tabella che ha come **righe il numero degli stati** e come **colonne i simboli della grammatica** (prima i terminali - per la parte ACTION e poi i non-terminali-per la parte GOTO)

- Per ogni **arco** (i, j) dell'automa etichettato con un simbolo **terminale** a , si inserisce l'azione **SHIFT** j alla posizione $M[i, a]$
- Per ogni **arco** (i, j) dell'automa etichettato con un **non terminale** x , si inserisce l'azione **GOTO** j alla posizione $M[i, x]$
- Per ogni **stato** i contenente l'item $x \rightarrow y$. (tale che $x \rightarrow y$ sia la produzione k -esima della grammatica), si inserisce in tabella l'azione **REDUCE** k in tutta la riga i
- Per ogni **stato** i contenente l'item $S' \rightarrow S$. si inserisce in tabella l'azione **ACCEPT** alla posizione $M[i, \$]$

ESEMPIO COSTRUZIONE TABELLA DI PARSING



(0) $S' \rightarrow S$
 (1) $S \rightarrow (L)$
 (2) $S \rightarrow x$
 (3) $L \rightarrow S$
 (4) $L \rightarrow L, S$

	ACTION				
	()	x	,	\$
1	S3		S2		
2	R2	R2	R2	R2	R2
3	S3		S2		
4					acc
5		S6		S8	
6	R1	R1	R1	R1	R1
7	R3	R3	R3	R3	R3
8	S3		S2		
9	R4	R4	R4	R4	R4

GOTO		S	L
1	G4		
2			
3	G7	G5	
4			
5			
6			
7			
8	G9		
9			

ALGORITMO DI PARSING LR(0)

Sia **M** la tabella LR(0) e sia **u** lo stato corrente (quello in testa alla pila):

- Se **M[u, t] = SHIFT u'**, si inserisce il simbolo **t** e lo stato **u'** nella pila e si sposta la testina dell'input.
- Se **M[u, t] = REDUCE k** (**t** è un qualunque terminale), dove **k** rappresenta la produzione **A → α.**, allora si rimuove la stringa **α** dalla pila, assieme a tutti gli stati corrispondenti (fino allo stato immediatamente prima di **α**) e si inserisce il non-terminale **A** ($2 * |\alpha|$ operazioni di pop e una di push). Siano **u' A** gli elementi (stato-non terminale) in testa alla pila, e sia **M[u', A] = GOTO u''**, allora si inserisce sulla pila lo stato **u''**.
- Se **M[u, t] = accept** allora termina
- Altrimenti si rileva un **errore**

PSEUDOCODICE PER PARSING LR(0)

```
ip punta al primo simbolo di w$;
while true do begin
    s := top(stack); a:= simbolo puntato da ip;
    if action[s,a] = shift s' then begin
        push(a); push(s');
        ip avanza di un simbolo
    end
    else if action[s,a] = reduce A-> $\beta$  then begin
        pop 2*| $\beta$ | simboli dallo stack;
        s' := top(stack);
        push(A); push(goto[s',A]);
        segnala in output la produzione A-> $\beta$ 
    end
    else if action[s,A] = accetta then return
        else errore()
end
```

<i>Passo</i>	<i>automa</i>	<i>input</i>	<i>azione</i>
1	1	(x,(x))\$	S3
2	1(3	x,(x))\$	S2
3	1(3x2	,(x))\$	R2 (S->x)+G7
4	1(3S7	,(x))\$	R3(L->S)+G5
5	1(3L5	,(x))\$	S8
6	1(3L5,8	(x))\$	S3
7	1(3L5,8 (3	x))\$	S2
8	1(3L5,8(3x2))\$	R2(S->x)+G7
9	1(3L5,8(3S7))\$	R3(L->S)+G5
10	1(3L5,8(3L5)) \$	S6
11	1(3L5,8(3L5)6)\$	R1(S->(L))+G9
12	1(3L5,8S9)\$	R4(L->L,S)+G5
13	1(3L5)\$	S6
14	1(3L5)6	\$	R1(S->(L))+G4
15	1 S 4	\$	accept

Esempio di Parsing
LR(0): il
riconoscimento di
(x,(x)) \$ per la
grammatica
precedente.

- (0) $S' \rightarrow S$
(1) $S \rightarrow (L)$
(2) $S \rightarrow x$
(3) $L \rightarrow S$
(4) $L \rightarrow L, S$

ACTION						GOTO	
	()	x	,	\$	S	L
1	S3		S2			G4	
2	R2	R2	R2	R2	R2		
3	S3		S2			G7	G5
4					acc		
5		S6		S8			
6	R1	R1	R1	R1	R1		
7	R3	R3	R3	R3	R3		
8	S3		S2			G9	
9	R4	R4	R4	R4	R4		

ESERCIZIO

$S \rightarrow BA$

$S \rightarrow aSAB$

$A \rightarrow B$

$A \rightarrow aA$

$B \rightarrow b$

Determinare l'automa LR, la tabella di parsing, verificare che la grammatica è LR(0) e verificare che abbbab appartiene al linguaggio generato dalla grammatica

GRAMMATICHE LR(0)

Grammatiche LR(0)

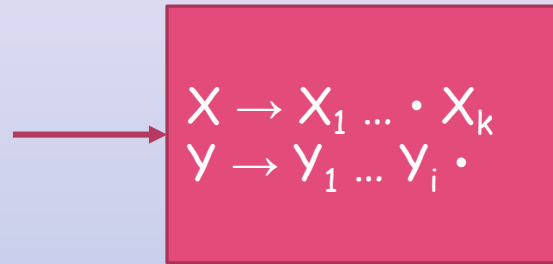
- E' una grammatica in cui ogni cella della tabella LR(0) contiene al più un solo valore.
- Equivalentemente, gli stati dell' automa o contengono solo produzioni che presuppongono shift o solo produzioni che presuppongono reduce.
- Gli stati che presuppongono operazioni di reduce, inoltre, contengono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

Proprietà dell'automata LR(0):

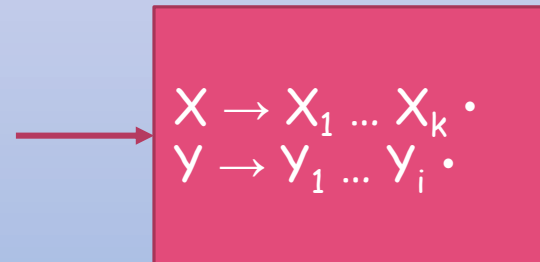
1. tutte le frecce entranti in uno stato hanno la stessa etichetta;
2. Uno stato di reduce non ha successori;
3. Uno stato di shift ha almeno un successore.

CONFLITTI SHIFT-REDUCE O REDUCE-REDUCE

Shift-Reduce


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

Reduce-Reduce


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

Se l'automa contiene stati di questo tipo, la grammatica non può essere LR(0).

ESEMPIO DI GRAMMATICA NON LR(0)

$S' \rightarrow E$
 $E \rightarrow T + E$
 $E \rightarrow T$
 $T \rightarrow x$

$S' \rightarrow \cdot E$
 $E \rightarrow \cdot T + E$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot x$

T

$E \rightarrow T \cdot + E$
 $E \rightarrow T \cdot$

Conflitto
shift-reduce

TABELLA AMBIGUA

(0) $S' \rightarrow E$

(1) $E \rightarrow T + E$

(2) $E \rightarrow T$

(3) $T \rightarrow x$

Dallo stato 3 leggendo "+" posso o shiftare su 4 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		

ESERCIZI

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

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow \text{id}$

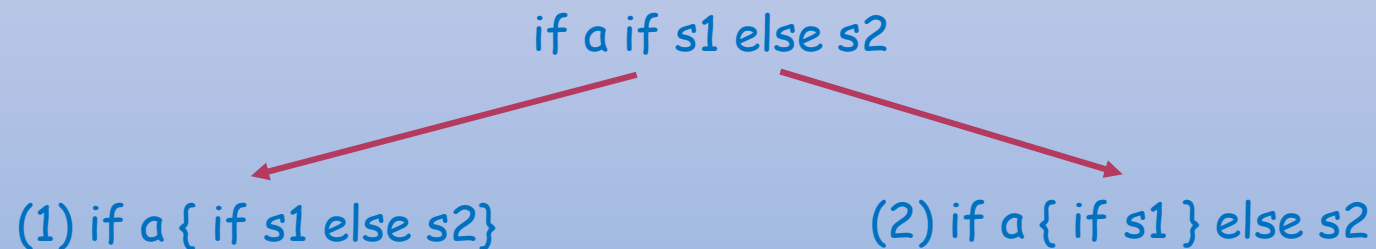
$R \rightarrow L$

CONFLITTI SHIFT REDUCE

Molti SR parser risolvono automaticamente i conflitti shift/reduce privilegiando lo **shift al reduce** (cioè 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)

Stmt \rightarrow Call-stmt | Assign-stmt

Call-stmt \rightarrow identifier

Assign-stmt \rightarrow var := esp

Var \rightarrow identifier

Esp \rightarrow var | number



$S' \rightarrow S$

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

$V \rightarrow id$

$E \rightarrow V \mid n$

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

id

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

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** for Java
- Necessitano di modificare la grammatica

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**, **CUP** for Java, **sablecc**
- 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 suo prefisso
può appartenervi

ESERCIZIO

La grammatica

$S \rightarrow aSb \mid \epsilon$

È LR(0)? È LL(1)?

Non è LR(0)! (Perché?)
È LL(1)!

La grammatica

$S \rightarrow SA \mid A$

$A \rightarrow aAb \mid ab$

È LR(0)? È LL(1)?

È 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)