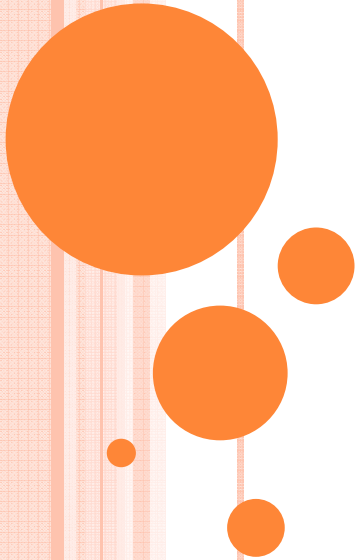
Parsing Bottom-up e Parser LR(0)



Venerdì 15 Novembre

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.

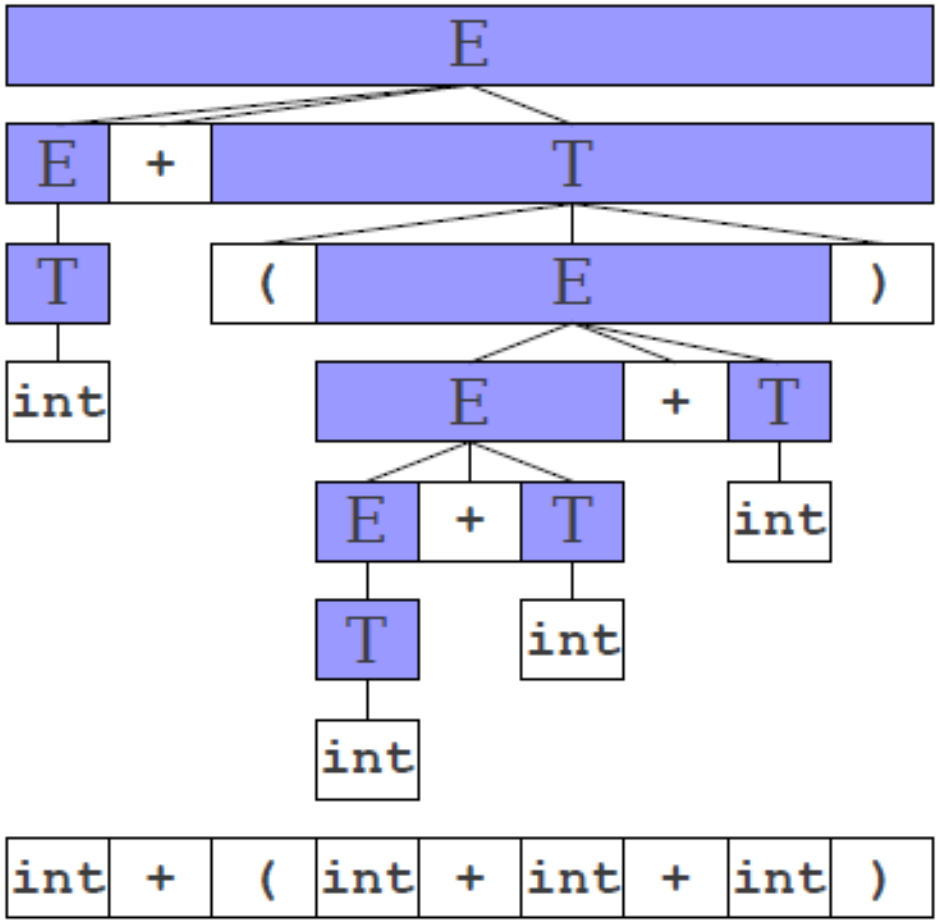
Esempio

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

$$
\begin{aligned}
& \texttt{int} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & T + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (T + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + T + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int}) \\
\Rightarrow\ & E + (E + T) \\
\Rightarrow\ & E + (E) \\
\Rightarrow\ & E + T \\
\Rightarrow\ & E
\end{aligned}
$$

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 di parsing bottom up

Per esempio: Si consideri la grammatica

S-> aABe

A-> Abc | b

B-> d

La sequenza abbcde può essere ridotta ad S nel modo seguente:

abbcde -> aAbcde -> aAde -> aABe -> 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.*

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 -> aABe
A -> Abc | b
B -> d

La frase abbcde$ viene valutata come segue:

• nello stato iniziale la pila è vuola

pila	input	azione
$	abbcde$	

• il parser esegue azioni, che oltre ad "**accept**" sono di tre ulteriori tipi:
shift: un simbolo terminale è spostato dalla stringa di input sulla pila (si indica scrivendo la parola "**shift**")
reduce: una stringa α sulla pila è ridotta al non-terminale A, secondo la regola A -> α (si indica scrivendo "**reduce** A-> α")
error: errore di sintassi

	pila	input	azione
1	$	abbcde$	**shift**
2	$a	bbcde$	**shift**
3	$ab	bcde$	**reduce** A ->b
4	$aA	bcde$	**shift**
5	$aAb	cde$	**shift**
6	$aAbc	de$	**reduce** A ->Abc
7	$aA	de$	**shift**
8	$aAd	e$	**reduce** B -> d
9	$aAB	e$	**shift**
10	$aABe	$	**reduce** S -> aABe
11	$S	$	**accept**

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?

Conflitti durante il Parsing SR

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

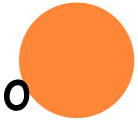
pila	**input**	
...**if** expr **then** stmt	**else** ...**$**	Shift o Reduce?

TIPICO SR 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.

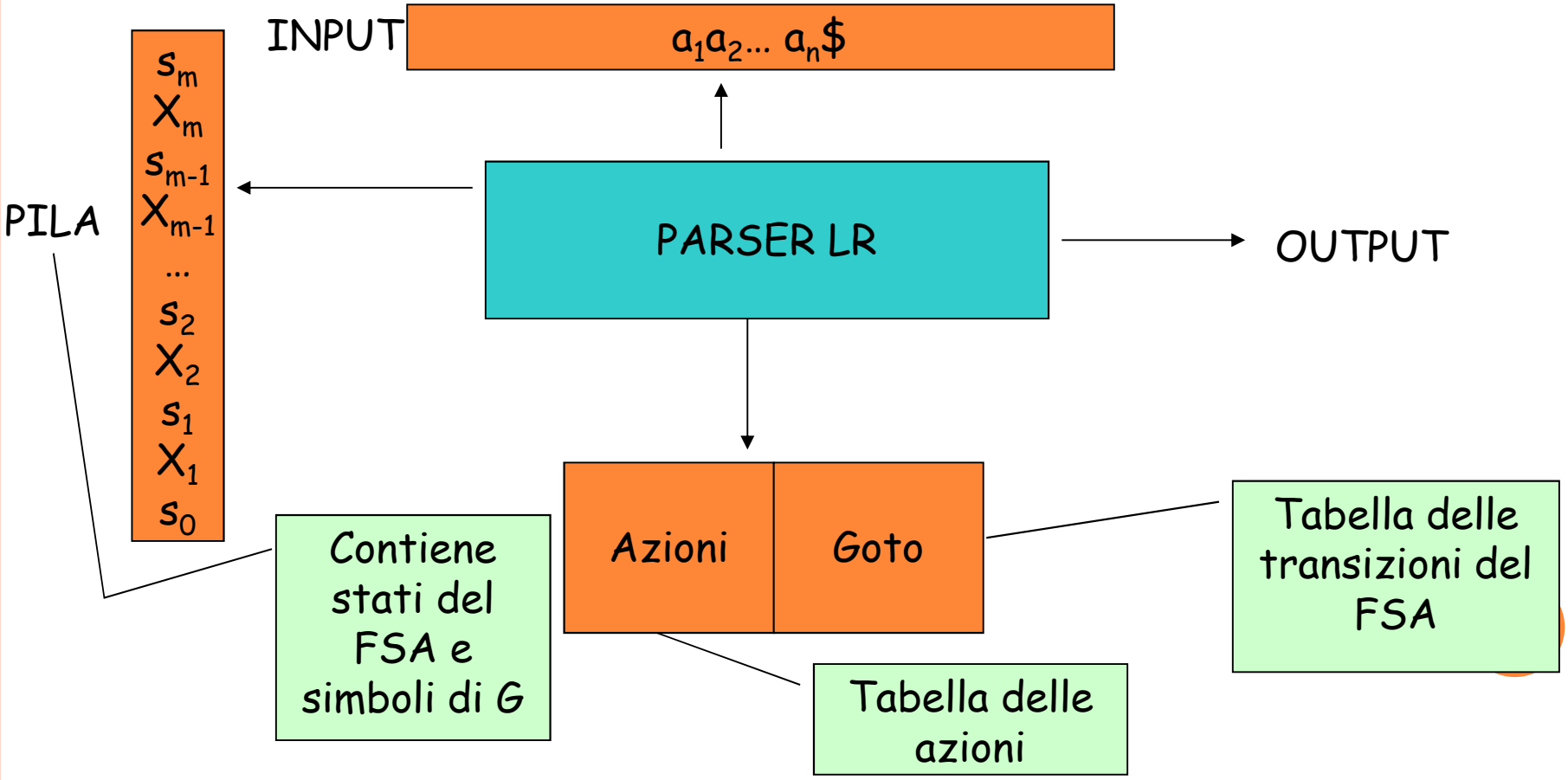
Perché usare i parser LR?

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



Configurazione e funzionamento di un parser LR

- Una configurazione di un LR parser è :
  $(\mathbf{s}_0 \ldots X_{m-1}\mathbf{s}_{m-1}X_m\mathbf{s}_m \ , a_i \ldots a_n\$)$
  che rappresenta la forma sentenziale destra
  $X_1 \ldots X_{m-1}X_m a_i \ldots a_n\$$
- L'azione del parser è determinata dallo stato che si trova in cima alla pila ed eventualmente dal simbolo (o un gruppo di simboli) dell'input.
- Lo stato successivo dipende dalla tabella goto.

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, la tecnica si

Il comportamento del parser **LR(0)** si basa sulla tabella **LR(0)** definita come segue

Caselle vuote rappre-sentano errori

Stati	**Azioni**	**Goto**	
		a b ( ) c	A B S S'
1	Shift	3 2 4 …	
2	Reduce A->b		
3	Reduce B->a		
4	Shift		4 2
…	Accept		
…	…		
	…		

Costruzione della tabella LR(0)

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-> XYZ, gli item sono:
A->.XYZ
A->X.YZ
A->XY.Z
A->XYZ.
La produzione A->ε genera l'item A->.

Un item indica quanto di una produzione è stato visto ad una certa fase del processo di parsing.

Gruppi 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' -> S

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

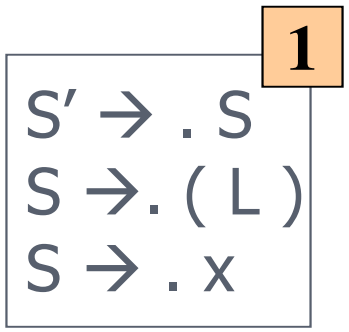
1. $I \subseteq$ CLOSURE(I);

2. Se A-> α.Bβ è in CLOSURE(I) e B->γ è una produzione, allora si aggiunge B->.γ in CLOSURE(I). Si applica questa regola fino a quando non aggiungo altri item.

```
Function Closure(I);
begin
   J:=I;
    repeat
       for each item A-> α.Bβ in J and
       each  B->γ such that B->γ is not in
       J do add B->.γ to J;
    until no more items can be added to J;
end
```

ESEMPIO: data la seguente grammatica, il primo stato è CLOSURE (S' → .S)

S' → S
S → (L)
S → x
L → S
L → L , S

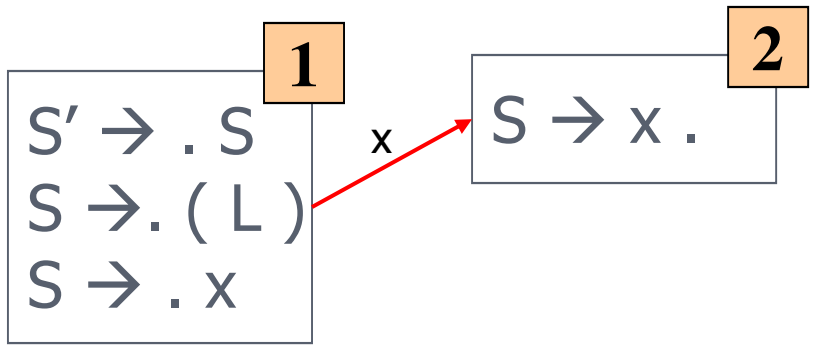


Automa LR(0)

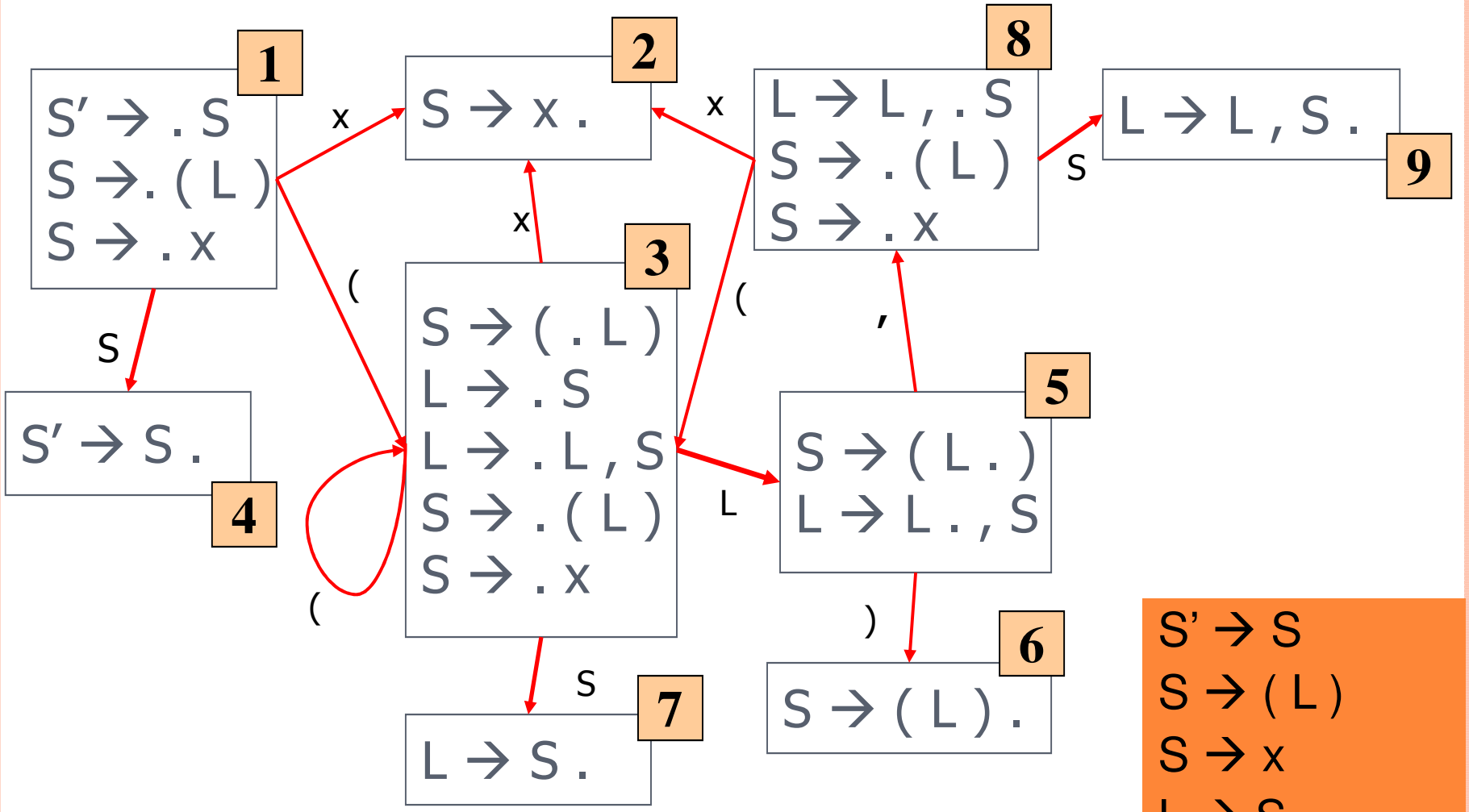
Operazione GOTO(I,X):
se I è un insieme di item e X un simbolo di G, allora GOTO(I,X) si definisce come la chiusura dell'insieme di tutti gli item A-> αX.β tale che A-> α.Xβ è un item di I.
Intuitivamente se I è l'insieme degli item validi per un prefisso riducibile γ , allora GOTO(I,X) è l'insieme degli item validi per γX.

```
Function GOTO(I,X);
begin
        J:=emptyset;
        for each item A-> α.Xβ in I do
           add A-> αX.β to J;
        Return CLOSURE(J);
end
```

Consente di costruire le transizioni dell'automa.



ESEMPIO


1
S' → . S
S → . (L)
S → . x
2
S → x .
3
S → (. L)
L → . S
L → . L , S
S → . (L)
S → . x
4
S' → S .
5
S → (L .)
L → L . , S
6
S → (L) .
7
L → S .
8
L → L , . S
S → . (L)
S → . x
9
L → L , S .
x
x
x
(
(
(
S
S
L
,
)
S' → S
S → (L)
S → x
L → S
L → L , S

Costruzione della tabella

- la tabella ha come righe il numero degli stati e come colonne i simboli della grammatica (prima i terminali e poi i non-terminali)

azioni shift: poichè dallo stato 1 esiste una transizione "(" (un simbolo terminale) nello stato 3, ciò significa eseguire una operazione di "shift 3" in corrispondenza di (1 , "(").

azioni goto: poichè dallo stato 1 esiste una transizione "S" nello stato 4 (un simbolo non-terminale), ciò significa eseguire una operazione di "goto 4" in corrispondenza di (1 , "S")

- continuando in questo modo si completa la tabella, per le azioni di shift e goto

Costruzione della tabella

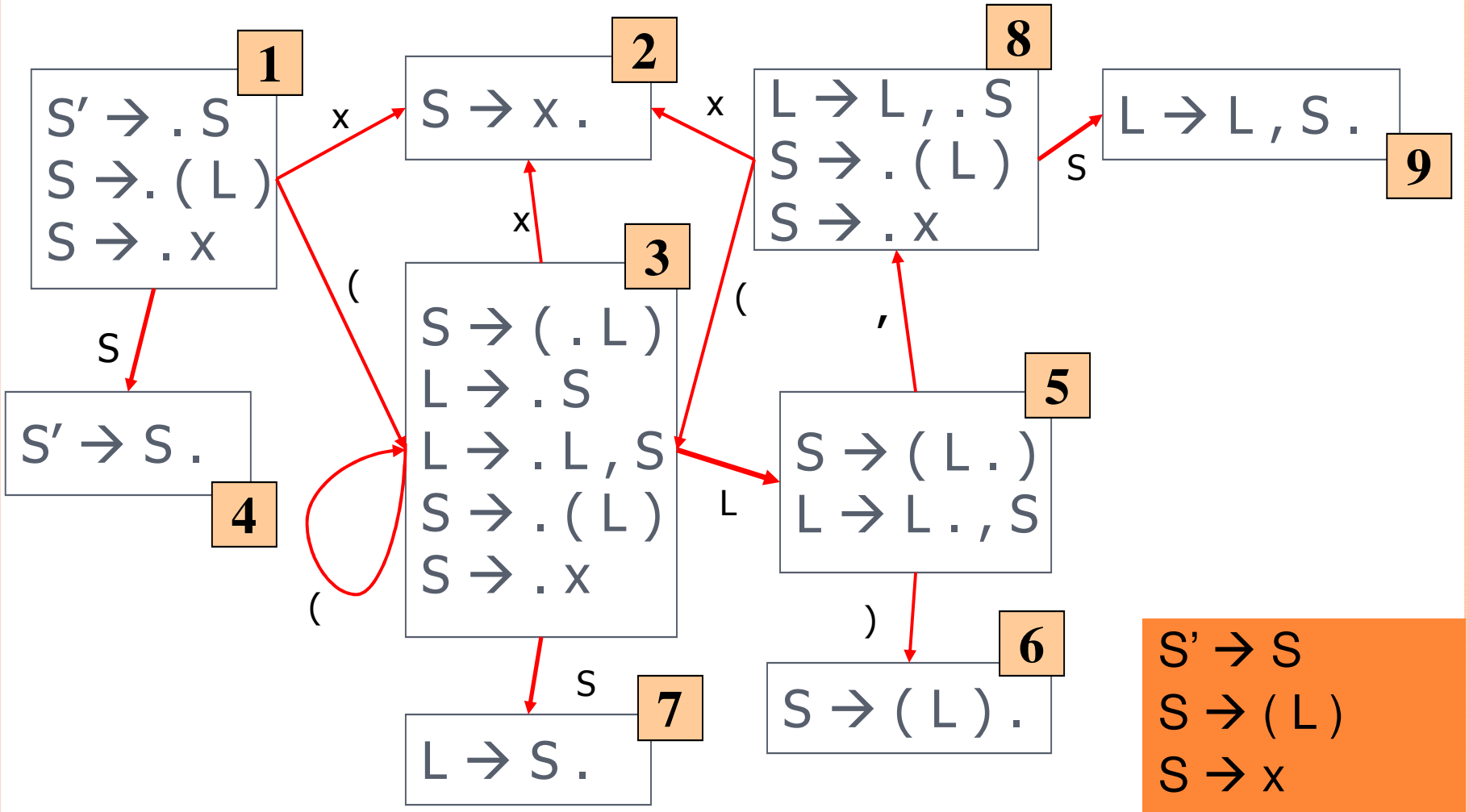
azioni reduce: ogni stato che contiene un elemento LR(0) del tipo

$$X \rightarrow \gamma.$$

deve avere una operazione di riduzione con tale regola per ogni simbolo terminale.

azione acc: l'operazione di reduce S' -> S. quando il simbolo terminale è "$" equivale ad accettazione e si sostituisce nella tabella con "acc" in corrispondenza di $.

ESEMPIO


1
S' → . S
S → . (L)
S → . x
2
S → x .
3
S → (. L)
L → . S
L → . L , S
S → . (L)
S → . x
4
S' → S .
5
S → (L .)
L → L . , S
6
S → (L) .
7
L → S .
8
L → L , . S
S → . (L)
S → . x
9
L → L , S .
x
x
x
(
(
(
S
S
L
,
)
S' → S
S → (L)
S → x
L → S
L → L , S

Costruzione della tabella

(0) S' → S
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

	(	)	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		
		action				goto	

Algoritmo di Parsing LR(0)

Sia $\mathbf{TAB_{LR0}}$ la tabella **LR(0)** e sia **u** lo stato corrente (*quello in testa alla pila*), le azioni sono:

- se $\mathbf{TAB_{LR0}}$[**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*|α| operazioni di pop)
  siano **u'***A* gli elementi in testa alla pila, e sia
  $\mathbf{TAB_{LR0}}$[**u'**, *A*] = **goto u"** , allora si inserisce sulla pila lo stato **u"**
- altrimenti, se $\mathbf{TAB_{LR0}}$[**u**, **t**] = **shift u'** si sposta sulla pila il token "**t**" in testa all'input e si inserisce nella pila lo stato **u'**
- altrimenti, se $\mathbf{TAB_{LR0}}$[**u**, **t**] = **accept** allora termina
- altrimenti si rileva un **errore**

Esempio: il riconoscimento di (x,(x)) $ per la grammatica precedente.

Esempio di parsing LR(0)

passo	*automa*	*input* `	*azione*
1	**1**	(x,(x)) $	**s3**
2	**1 (3**	x,(x)) $	**s2**
3	**1 (3 x 2**	,(x)) $	**r2** (S → x) + **g7**
4	**1 (3 S 7**	,(x)) $	**r3** (L → S) + **g5**
5	**1 (3 L 5**	,(x)) $	**s8**
6	**1 (3 L 5 , 8**	(x)) $	**s3**
7	**1 (3 L 5 , 8 (3**	x)) $	**s2**
8	**1 (3 L 5 , 8 (3 x 2**	)) $	**r2** (S → x) + **g7**
9	**1 (3 L 5 , 8 (3 S 7**	)) $	**r3** (L → S) + **g5**
10	**1 (3 L 5 , 8 (3 L 5**	)) $	**s6**
11	**1 (3 L 5 , 8 (3 L 5) 6**	) $	**r1** (S → (L)) + **g9**
12	**1 (3 L 5 , 8 S 9**	) $	**r4** (L → L,S) + **g5**
13	**1 (3 L 5**	) $	**s6**
14	**1 (3 L 5) 6**	$	**r1** (S → (L)) + **g4**
15	**1 S 4**	$	**accept**

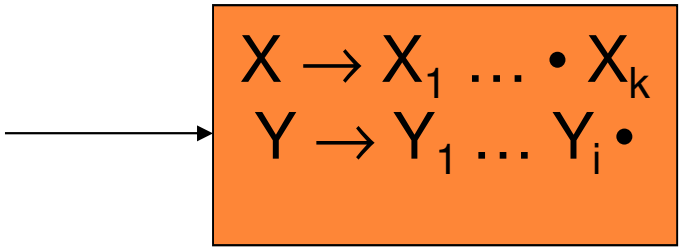
Grammatiche LR(0)

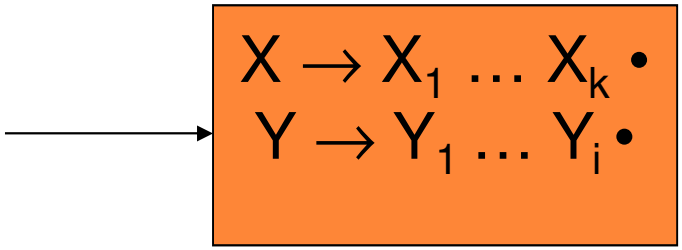
- E' una grammatica in cui ogni cella della tabella LR(0) contiene al più un solo valore.
- Equivalentemente, gli stati dell' automa sono o contenenti solo produzioni che presuppongono shift o solo produzioni che presuppongono reduce.
- Gli stati che presuppongono operazioni di reduce, inoltre, contengono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

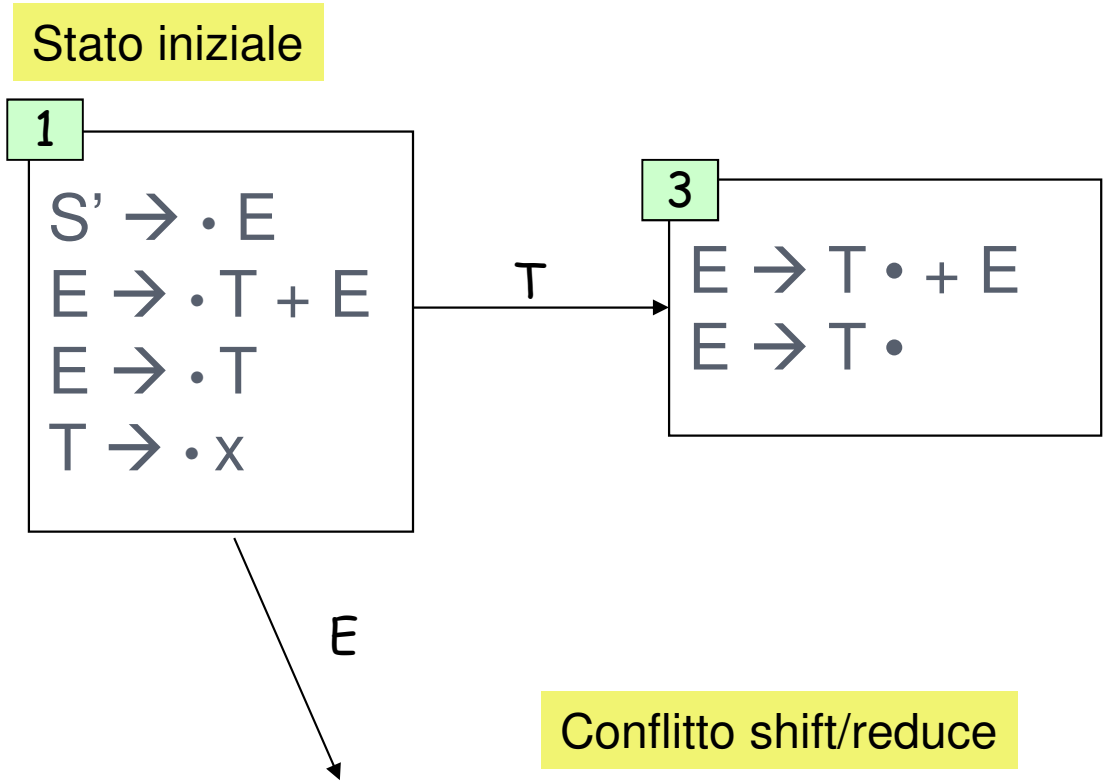




In questi casi si tratta di una grammatica non LR(0).

Esempio di grammatica non LR(0)

S' → E
E → T + E
E → T
T → x



Conflitto shift/reduce

Tabella ambigua

(0) S' → E
(1) E → T + E
(2) E → T
(3) T → x

Dallo stato 3 leggendo "+" posso o shiftare su 4 o ridurre con E → 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		

LL(k) vs. LR(k)

- **L**eft to Right parse
- **L**eftmost derivation
- *k*-token look ahead

➔ LL(k)

- 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.

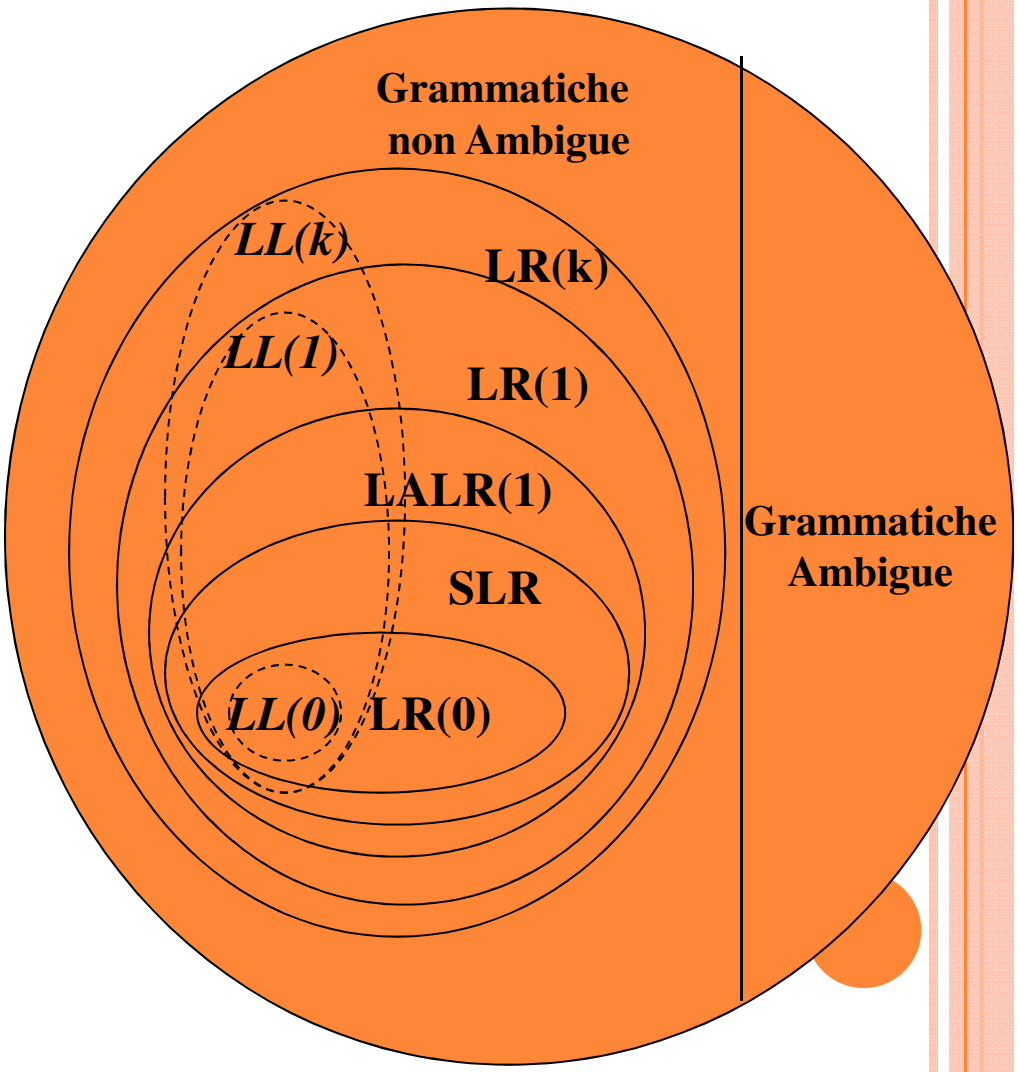
- **L**eft to Right parse
- **R**ightmost derivation
- *k*-token look ahead

➔ LR(k)

- 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'->S.

LR Parsing

- Le grammatiche LR sono più potenti delle LL.
- LR(0) ha esclusivo interesse didattico.
- Simple LR (SLR) consentono il parsing di famiglie di linguaggi interessanti.
- 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.
  - L'implementazione è poco controllabile.
  - Si cercano grammatiche LALR(1) equivalenti.



Esercizi

La grammatica

S-> A| aS

A-> aAb |ε

è LR(0)? E' LL(1)?

Il linguaggio è deterministico ma non LL(k)

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

La grammatica

S->a|ab

è LR(0)? E' LL(1)?

In un linguaggio LR(0), se una stringa appartiene al linguaggio, nessun prefisso di essa può appartenervi

Esercizi

La grammatica

S->aSb|ε

è LR(0)? E' LL(1)?

La grammatica

S->SA|A

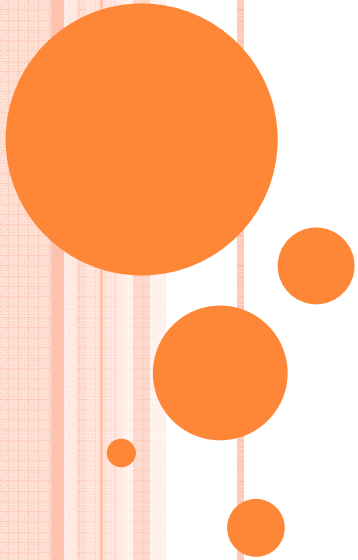
A-> aAb|ab

è LR(0)? E' LL(1)?

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 linguagio chiuso per prefissi non è LR(0) ma può essere LL(1)
  - Esistono linguaggi LR(0) ma non LL(1)



Parsing SLR

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).
      - 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)).

Esempio di grammatica non LR(0)

S' → E
E → T + E
E → T
T → x

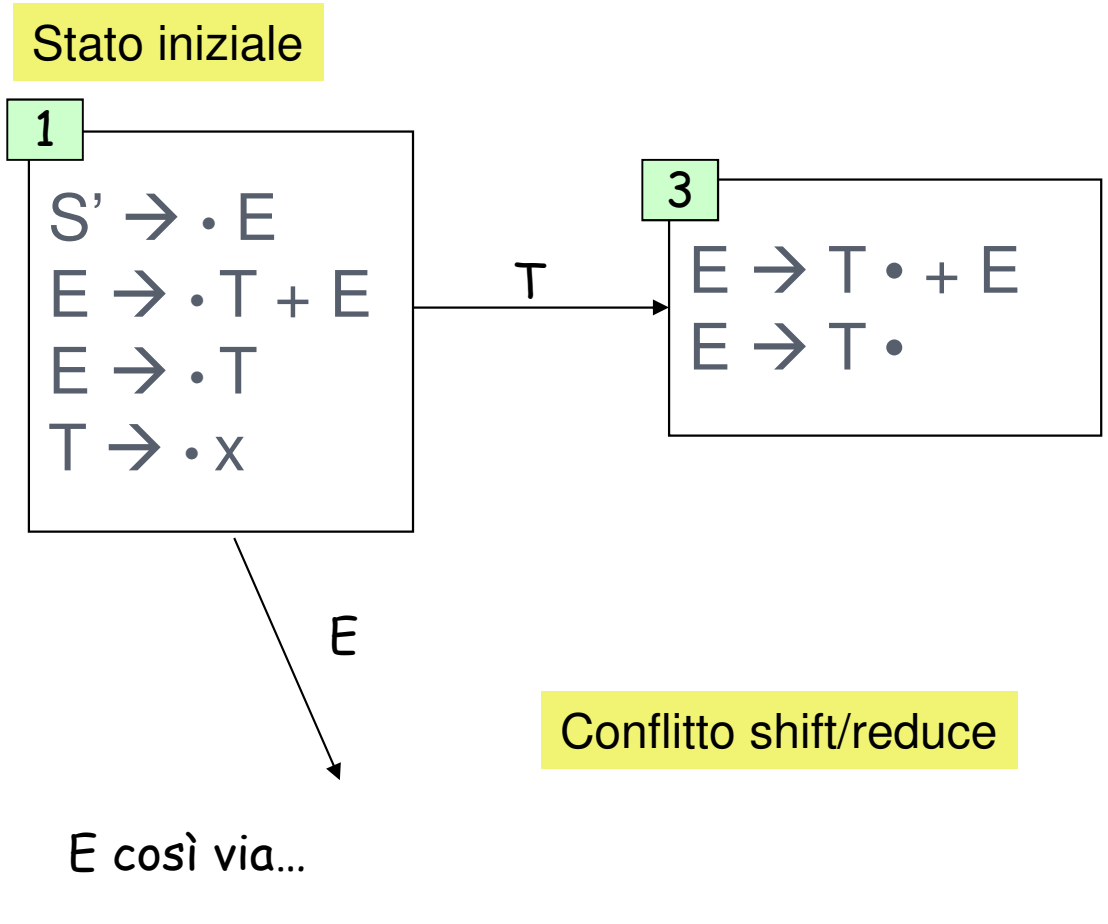


Tabella ambigua

(0) S' → E
(1) E → T + E
(2) E → T
(3) T → x

Dallo stato 3 leggendo "+" posso o shiftare su 4 o ridurre con by E → 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		

SIMPLE LR PARSER (SLR o SLR(1))

E' un modo semplice di costruire parser più potenti di **LR(0)** *utilizzando il prossimo token di input per decidere su alcune azioni e costruire la tabella.*

Sia s lo stato corrente:

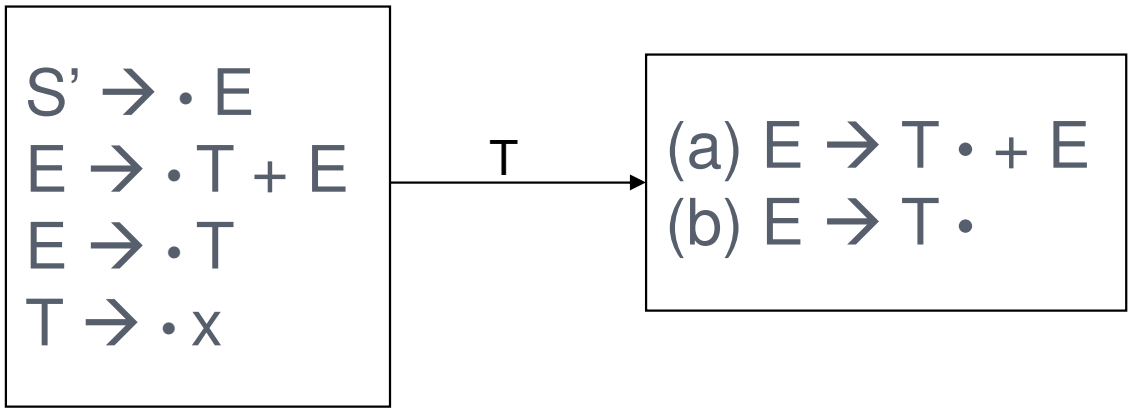
1. se lo stato s contiene un item della forma A-> α.xβ e x è l'etichetta di una transizione uscente, allora TAB_{SRS}[s,x]=shift u, dove u è lo stato che contiene
   A-> αx.β.
2. se lo stato s contiene A-> γ. allora TAB_{SRS}[s,t]=reduce A-> γ per tutti i token t contenuti in FOLLOW(A).
   Ciò vale nel caso in cui A sia diverso da S'.
3. se lo stato s contiene S'-> S. allora TAB_{SRS}[s,$]=accept.

Le grammatiche per cui le tabella di analisi prodotte dai parser **SLR(1)** non contengono ambiguità sono dette ***grammatiche*** **SLR(1)**

Molte grammatiche dei linguaggi di programmazione sono **SLR(1)**

FOLLOW(E) = { $ }

S' → E
E → T + E
E → T
T → x



Viene eliminata l'ambiguità dalla tabella poichè:

- reduce (b) sul token "$"
- shift (a) sul token "+"

Tabella SLR

(0) S' → E
(1) E → T + E
(2) E → T
(3) T → x

La riduzione avviene solo se il token successivo è un simbolo valido nella riduzione.

	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

Una grammatica è SLR se e solo se per ogni stato s:

1. Per ogni item A->α.xβ con x terminale, non c'è l'item B->γ. in s con x in Follow(B).
2. Per ogni coppia di item A->α. e B->β. Follow(A) e Follow(B) sono disgiunti

Esercizi

Costruire la tabella SLR per la grammatica
E'-> E
E-> E+n | n

Costruire la tabella SLR per la grammatica
S'-> S
S-> (S)S | ε

Costruire la tabella SLR per la grammatica
E -> T E -> E+T T -> (E)
T -> k

Costruire la tabella SLR per la grammatica
D->tL; L->i L->L,i
che schematizza la dichiarazione di una serie di identificatori. E' LR(0)?
Genera un linguaggio regolare?