PARSER LR(1)

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.

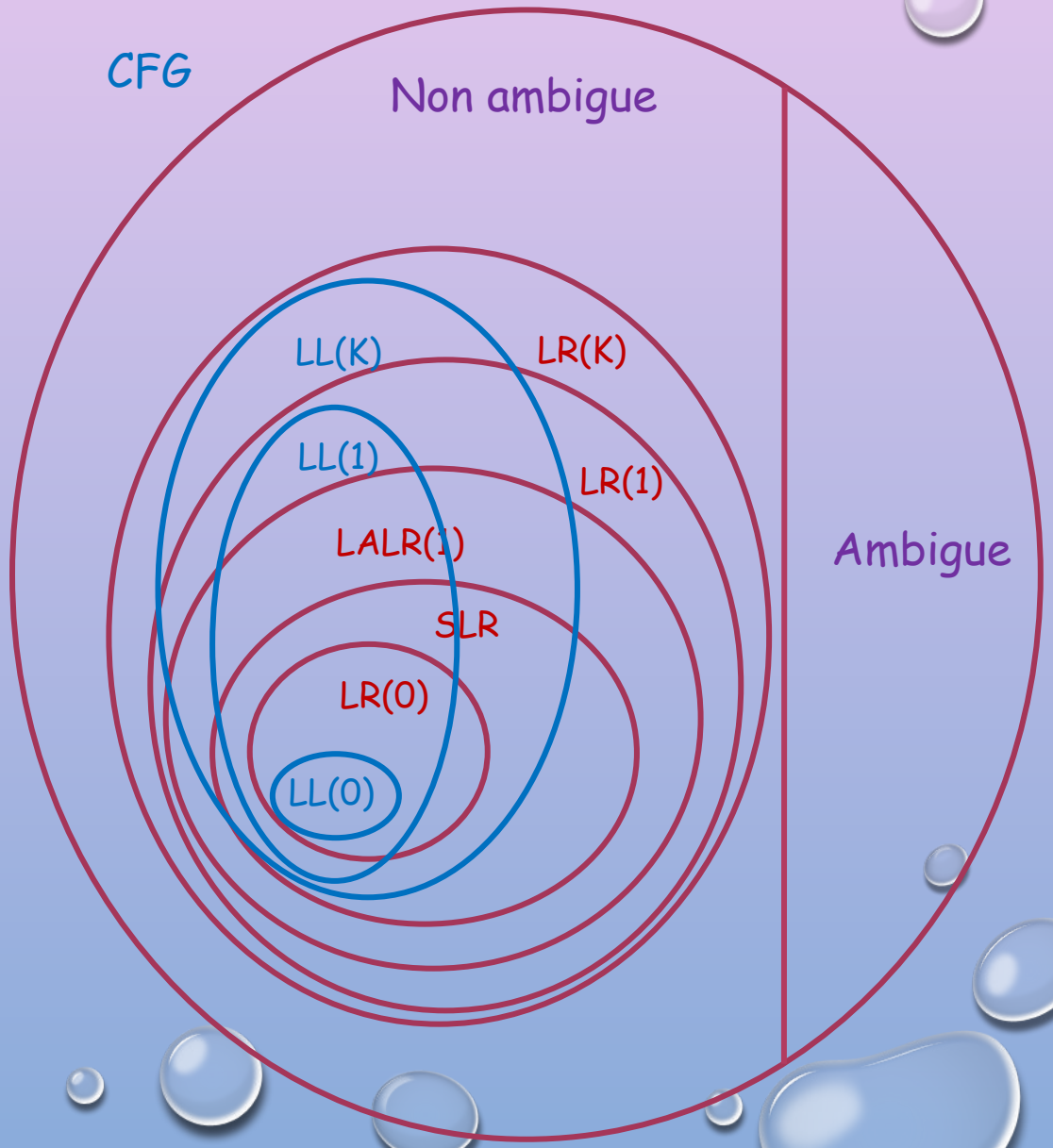
Esiste una classe di parser chiamato Simple LR (o SLR) che consentono il parsing di una famiglia un po' più vasta di Linguaggi.

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.

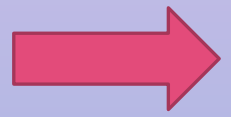
Si cercano grammatiche LALR(1) equivalenti.



IL METODO LR

Nel metodo SLR, se siamo in uno stato che contiene l'item **A->α.** e il prossimo simbolo di input **a** è in **FOLLOW(A)**, dobbiamo fare una riduzione tramite **A->α**

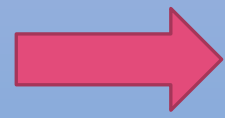
Se la grammatica non è SLR in generale questo può essere sbagliato: se sulla pila dei simboli c'è **βα**, **βA** potrebbe non essere seguito da **a** in nessuna forma sentenziale destra.



Per evitare questi errori, aggiungiamo agli item un simbolo terminale: un item nel metodo LR sarà del tipo **[A->α.β,a]** (un solo simbolo terminale se la grammatica è LR(1), k simboli terminali se è LR(k))

In realtà il simbolo terminale è usato solo negli item del tipo **A->α.**, ossia quelli che indurrebbero un reduce.

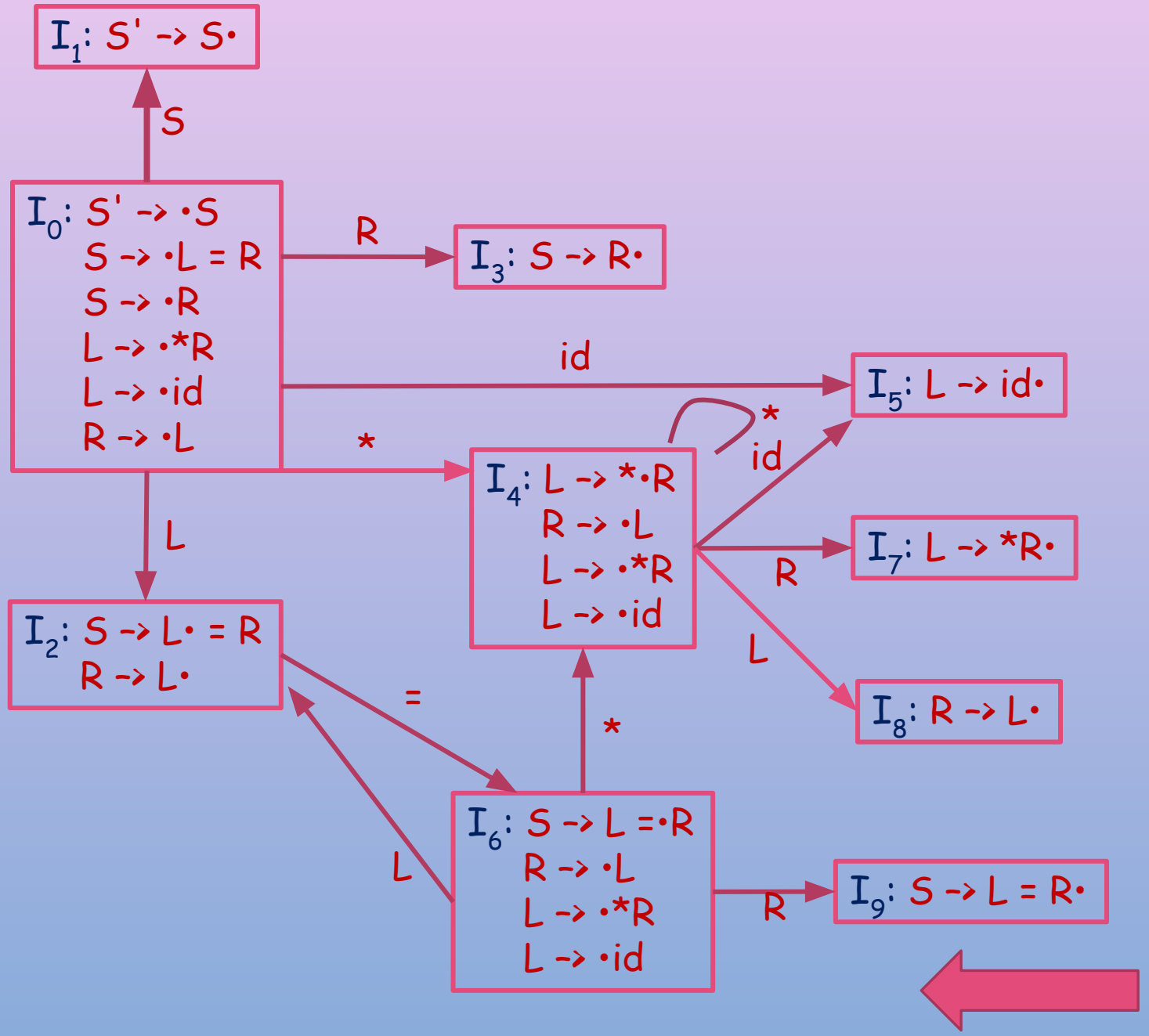
Viene fatta una riduzione tramite **A->α** solo se il prossimo simbolo di input è **a**



S' -> S
S -> L = R | R
R -> L
L -> *R | id

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

Lo stato I_2 permette un reduce R->L oltre che un'azione di shift. Ma non esiste nessuna forma sentenziale destra della grammatica che inizia con =R..
Quindi in questo caso è corretto fare lo shift e non la riduzione



PARSER LR(1)

Consente di ovviare a molte ambiguità dei parser **LR(0)** al prezzo di una crescita sostanziale della complessità dell'algoritmo. Il metodo LR(1) è poco usato in pratica proprio perché poco efficiente:

- Si preferisce il più semplice **LALR(1)**
- L'automa riconoscitore per i parser LR(1) è simile agli automi **LR(0)**: cambiano gli **item** e le operazioni **closure**, **goto** e **reduce**
- **Fu il primo ad essere introdotto [Knuth 1965]**

Idea:

Nei parser **LR(1)** il lookahead è utilizzato **durante** la costruzione dell'automa (quindi si prendono in considerazione i simboli che veramente possano seguire un certo handle)

COSTRUZIONE DELLA TABELLA LR(1)

La tabella di Parsing LR(1) si costruisce in maniera analoga alla tabella di parsing LR(0), con delle piccole differenze:

- **Gli item LR(1)** hanno la forma **(A->α.β,t)** dove **A->α.β** è un item **LR(0)** e **t** è un token (il simbolo terminale di lookahead) oppure **t=$** (ciò indica il fatto che la sequenza α si trova in cima alla pila e che alla testa dell'input c'è la stringa derivabile da **βt**).
- Si compiono azioni di **Reduce X->γ** solo in corrispondenza di quei simboli t per cui esiste lo stato di accettazione contiene l'item **[X->γ.,t]**

LE OPERAZIONI CLOSURE E GOTO

```
Function Closure(I);
begin
J:=I;
repeat
    for each item [A->α.Xβ,z] in J
    for each rule X->γ
    for each x ∈ FIRST(βz)
        add [X->.γ,x] to J;
until no more items can be added to J;
Return J;
end
```

```
Function Goto(I,X);
begin
J:= insieme degli item [A->αX.β,a]
tali che [A->α.Xβ,a] è in I;
return CLOSURE(J);
end
```

Esempio:

```
S' -> S
S-> CC
C-> cC|d
```

Closure ([S'->.S, $])

[S'->.S, $],
[S->.CC,$]
[C->.cC, {c,d}],
[C->.d, {c,d}] }

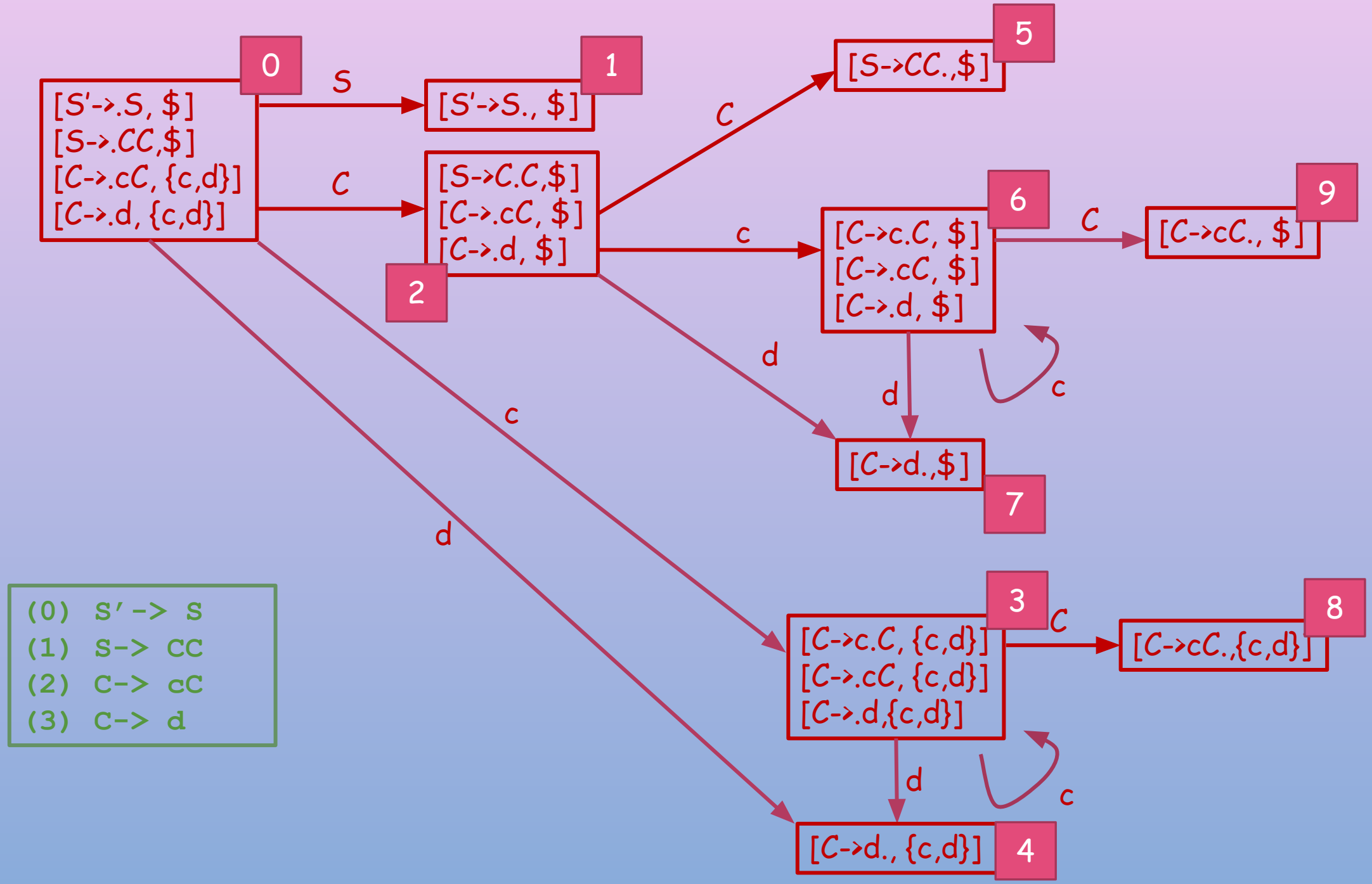
First($)=$

First(C$)={c,d}

[S'->.S, $]
[S->.CC,$]
[C->.cC, {c,d}]
[C->.d, {cd}]

S → [S'->S., $]

C → [S->C.C,$]
[C->.cC, $]
[C->.d, $]


0
[S'->.S, $]
[S->.CC,$]
[C->.cC, {c,d}]
[C->.d, {c,d}]
S
1
[S'->S., $]
C
2
[S->C.C,$]
[C->.cC, $]
[C->.d, $]
5
[S->CC.,$]
c
6
[C->c.C, $]
[C->.cC, $]
[C->.d, $]
C
9
[C->cC., $]
d
c
d
[C->d.,$]
7
c
d
3
[C->c.C, {c,d}]
[C->.cC, {c,d}]
[C->.d,{c,d}]
C
8
[C->cC.,{c,d}]
d
c
[C->d., {c,d}]
4
(0) S' -> S
(1) S-> CC
(2) C-> cC
(3) C-> d

COSTRUZIONE DELLA TABELLA LR(1)

La tabella M è strutturalmente simile a quella **LR(0)**

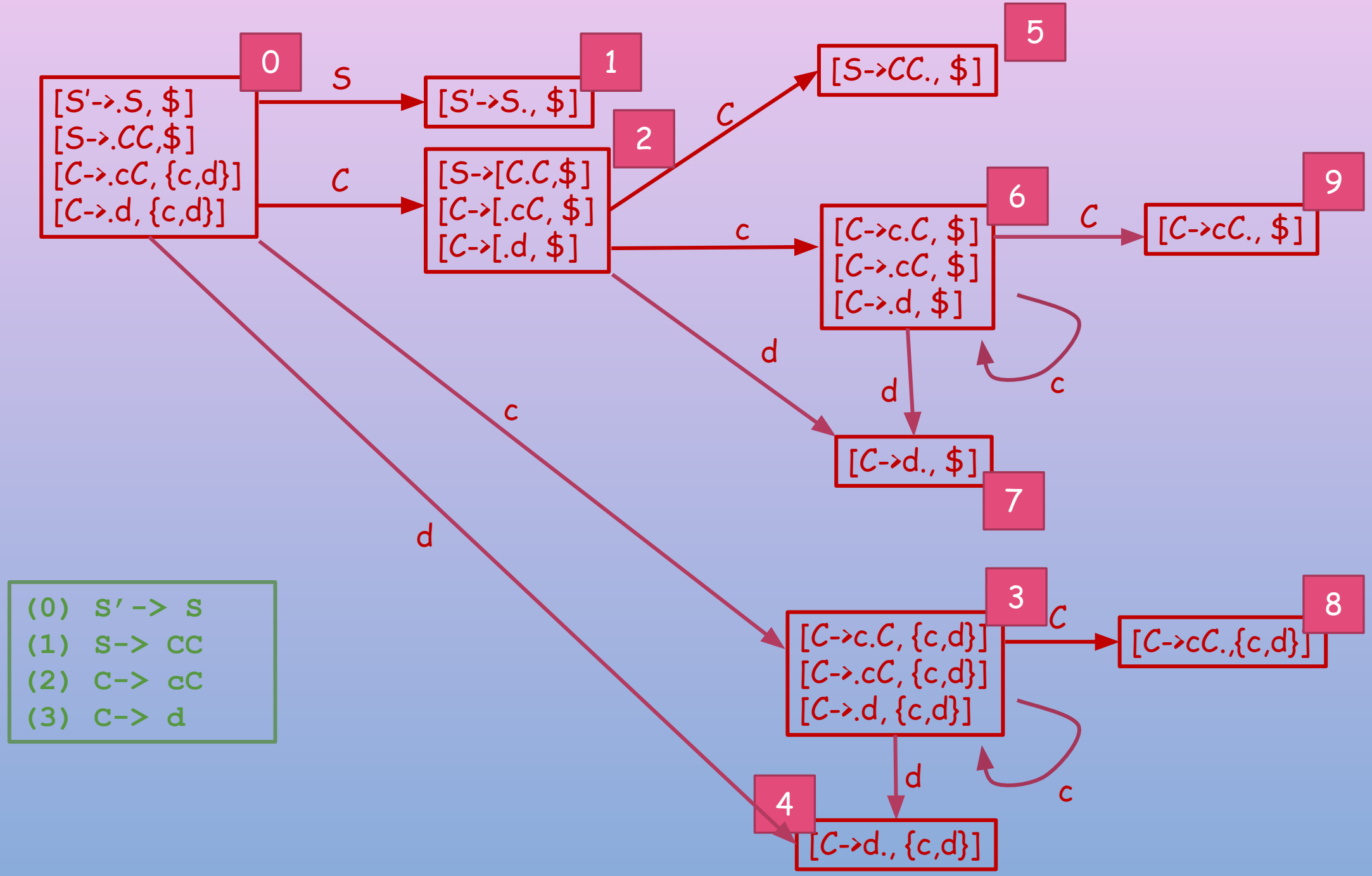
Azioni shift: se dallo stato **q** esiste una transizione **t** (**simbolo terminale**) nello stato **q'**, inserire **shift q'** in **M(q,t)**

Azioni goto: se dallo stato **q** esiste una transizione **S** (**simbolo non-terminale**) in **q'**, inserire **goto q'** in **M(q,S)**

Azioni reduce: se lo stato **q** contiene un item **LR(1)** del tipo **[X->γ.,t], con t simbolo terminale e X diverso da S'** inserire **reduce X->γ** in **M(q,t)**

Azione accept: se lo stato **q** contiene l'item **[S'->S.,$]**, allora inserire **accept** in **M(q,$)**

Osservazione: il parser **LR(1)** ridurrà **soltanto** quando il simbolo in testa all'input sarà **t**. Lo stato iniziale è quello costruito da **[S'->.S,$]**.


0
[S'->.S, $]
[S->.CC,$]
[C->.cC, {c,d}]
[C->.d, {c,d}]
S
C
c
d
1
[S'->S., $]
2
[S->[C.C,$]
[C->[.cC, $]
[C->[.d, $]
5
[S->CC., $]
6
[C->c.C, $]
[C->.cC, $]
[C->.d, $]
9
[C->cC., $]
7
[C->d., $]
3
[C->c.C, {c,d}]
[C->.cC, {c,d}]
[C->.d, {c,d}]
8
[C->cC.,{c,d}]
4
[C->d., {c,d}]
(0) S' -> S
(1) S-> CC
(2) C-> cC
(3) C-> d

```
(0)  S'-> S
(1)  S-> CC
(2)  C-> cC
(3)  C-> d
```

Stato	c	d	$	S	C
0	Shift 3	Shift 4		Goto 1	Goto 2
1			Accept		
2	Shift 6	Shift 7			Goto 5
3	Shift 3	Shift 4			Goto 8
4	R. C->d	R. C->d			
5			R. S->CC		
6	Shift 6	Shift 7			Goto 9
7			R. C->d		
8	R. C->cC	R. C->cC			
9			R. C->cC		

PARSER LALR(1)

PARSING LALR(1)

Le tabelle di analisi **LR(1)** sono di solito di **ordini di grandezza maggiori** di quelle **LR(0)**: se una tabella **LR(0)** di un linguaggio di programmazione è intorno ai 10 KB, una tabella **LR(1)** dello stesso linguaggio è intorno ai MB, con il doppio della memoria per costruirla.

Osservazione: se consideriamo l'automa **LR(1)** della grammatica

S'->S, S->CC, c->cC|d

e ignoriamo i lookahead, alcune coppie di stati sono identici: per esempio le coppie di stati (8, 9), (4, 7), (3,6), a prescindere dal simbolo di lookahead, sono uguali. Diciamo che queste coppie hanno lo stesso core.

Il parser **LALR(1)** consiste nell'identificare questi stati, combinando i loro lookahead, con l'obiettivo di ottenere un DFA **LR(1) simile** al DFA **LR(0)**. **Infatti:**

1. Le prime componenti degli item LR(1) sono item LR(0)
2. Se due stati p e q LR(1) hanno lo stesso core e se da p esce una transizione con X verso lo stato p', allora anche da q uscirà una transizione con X verso q' e p' e q' avranno lo stesso core.

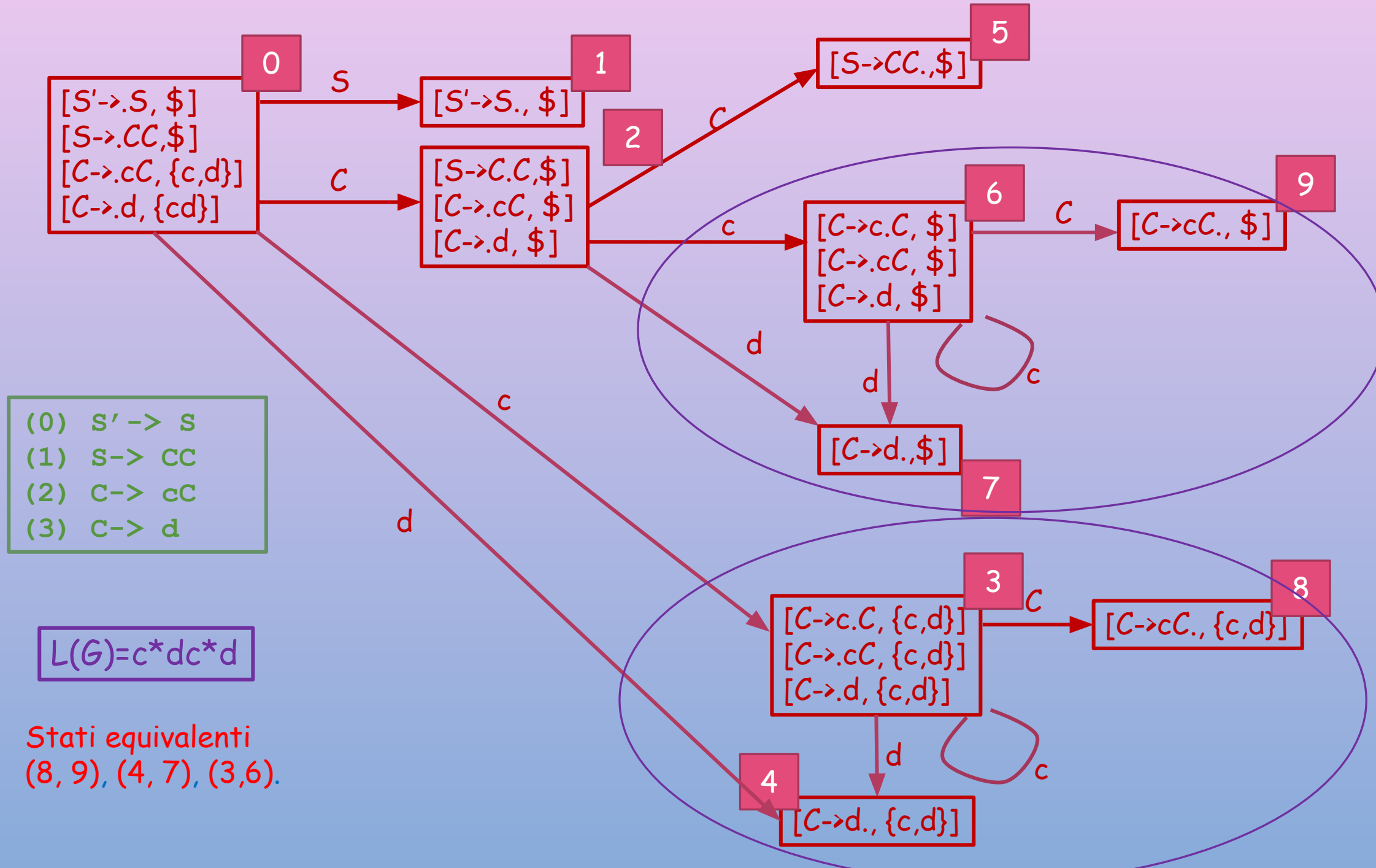
METODO LALR

1. Costruire l'automa con item LR(1)
2. Si accorpano gli stati con core comune.
3. Tabella ACTION costruita come prima ma dal nuovo insieme di stati
4. Tabella GOTO: goto(I,X)=K se
   - I e' l'unione di I_1, …, I_n
   - K e' l'unione degli stati che hanno lo stesso core di Goto(I_1,x)

     Core(goto(I_1,X)) = core(goto(I_2,X)) = …

Una grammatica e' LALR(1) se la tabella ottenuta non ha conflitti

Osservazione: Non si possono introdurre conflitti shift/reduce (altrimenti ci sarebbero stati anche prima dell'accorpamento degli stati). Si potrebbero invece introdurre conflitti reduce/reduce.


0
[S'->.S, $]
[S->.CC,$]
[C->.cC, {c,d}]
[C->.d, {cd}]
S
1
[S'->S., $]
C
2
[S->C.C,$]
[C->.cC, $]
[C->.d, $]
5
[S->CC.,$]
c
6
[C->c.C, $]
[C->.cC, $]
[C->.d, $]
C
9
[C->cC., $]
d
d
c
[C->d.,$]
7
c
d
3
[C->c.C, {c,d}]
[C->.cC, {c,d}]
[C->.d, {c,d}]
C
8
[C->cC., {c,d}]
d
c
4
[C->d., {c,d}]
(0) S' -> S
(1) S-> CC
(2) C-> cC
(3) C-> d
L(G)=c*dc*d
Stati equivalenti
(8, 9), (4, 7), (3,6).

```
(0)  S'-> S
(1)  S-> CC
(2)  C-> cC
(3)  C-> d
```

Sulla tabella questo equivale ad accorpare righe

Stato	c	d	$	S	C
0	Shift 3	Shift 4		Goto 1	Goto 2
1			Accept		
2	Shift 6	Shift 7			Goto 5
3	Shift 3	Shift 4			Goto 8
4	Reduce 3	Reduce 3			
5			Reduce 1		
6	Shift 6	Shift 7			Goto 9
7			Reduce 3		
8	Reduce 2	Reduce 2			
9			Reduce 2		

```
(0)  S'-> S
(1)  S-> CC
(2)  C-> cC
(3)  C-> d
```

Stato	c	d	$	S	C
0	Shift 3	Shift 4		Goto 1	Goto 2
1			Accept		
2	Shift 6	Shift 7			Goto 5
3	Shift 3	Shift 4			Goto 8
4	Reduce 3	Reduce 3	Reduce 3		
5			Reduce 1		
8	Reduce 2	Reduce 2	Reduce 2		

CONDIZIONI LALR

Nota che una grammatica soddisfa la condizione LALR(1) se valgono entrambe le condizioni:

1. Ogni candidata di riduzione ha un insieme di prospezione disgiunto dalle etichette terminali uscenti;
2. Se vi sono due candidate di riduzione i loro insiemi di prospezione sono disgiunti;

GAMMATICA LR(1)

Sia data la grammatica

S->aXb

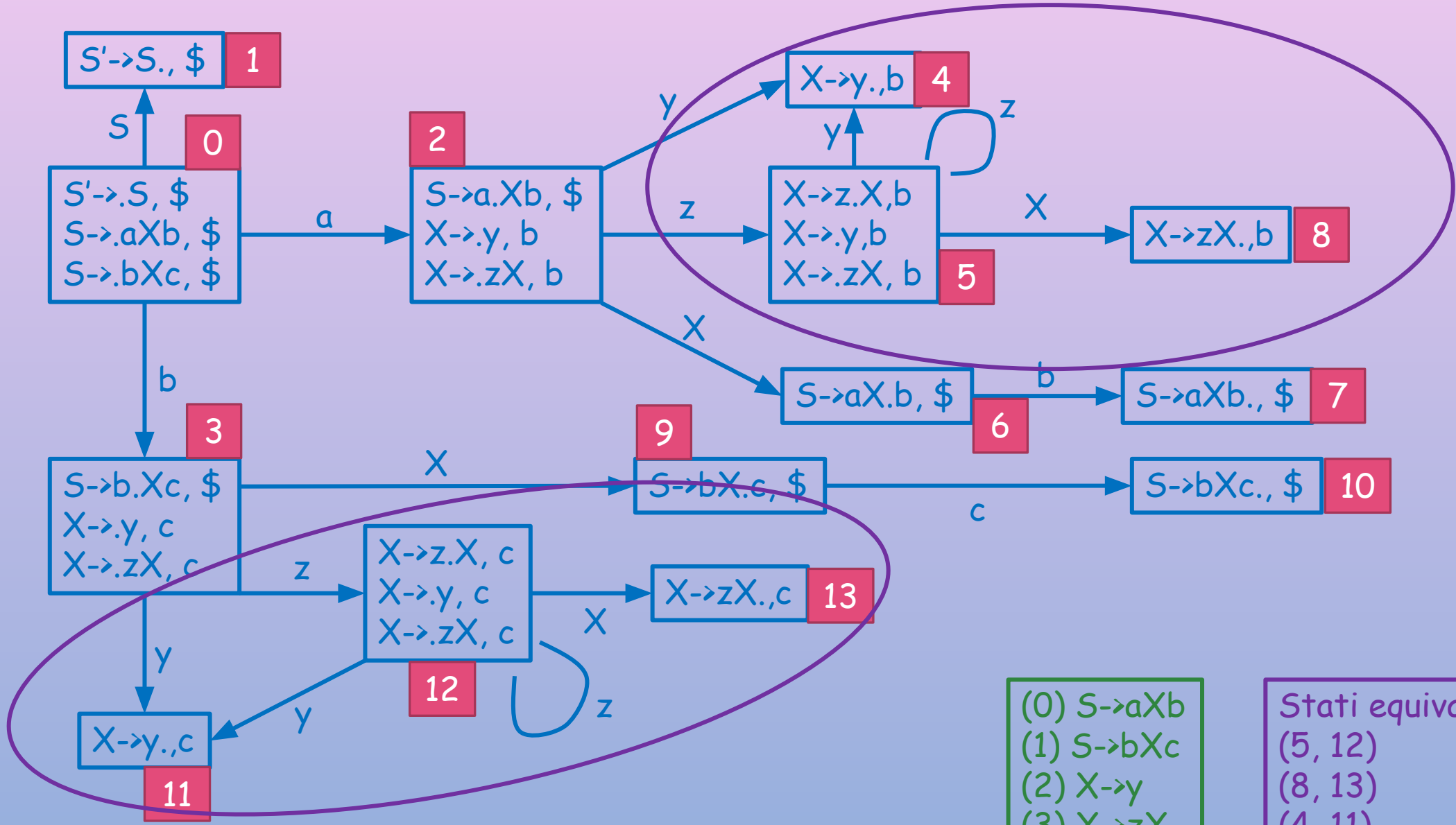
S->bXc

X->y

X->zX

Costruire l'automa LR(1).

Sugg. Ha 14 stati e 3 coppie di stati possono essere fusi, dando luogo all'automa LALR(1) con 11 stati.



(0) S->aXb
(1) S->bXc
(2) X->y
(3) X->zX

Stati equivalenti:
(5, 12)
(8, 13)
(4, 11)

Tabella LR(1)

	a	b	c	y	z	$	S	X
0	Shift 2	Shift 3					Goto 1	
1						Accept		
2				Shift 4	Shift 5			Goto 6
3				Shift 11	Shift 12			Goto 13
4		Reduce X->y						
5				Shift 4	Shift 5			Goto 8
6		Shift 7						
7						Reduce S->aXb		
8		Reduce X->zX						
9			Shift 10					
10						Reduce S->bXc		
11			Reduce X->y					
12				Shift 11	Shift 12			
13			Reduce X->zX					

Tabella LALR(1)

	a	b	c	y	z	$	S	X
0	Shift 2	Shift 3					Goto 1	
1						Accept		
2				Shift 4	Shift 5			Goto 6
3				Shift 11	Shift 12			Goto 13
4		Reduce X->y	Reduce X->y					
5				Shift 4	Shift 5			Goto 8
6		Shift 7						
7						Reduce S->aXb		
8		Reduce X->zX	Reduce X->zX					
9			Shift 10					
10						Reduce S->bXc		

ESEMPIO LR(1) MA NON LALR(1)

S->A | Ba |bAa |bB

A->a

B->a

Un parsing LALR(1) potrebbe generare conflitti che il LR(1) corrispondente non genererebbe (ciò non accade in pratica).

Si dimostra che se una grammatica è LR(1), la tabella LALR(1) non può avere conflitti shift/reduce ma solo reduce/reduce.

E' possibile computare il DFA del LALR(1) direttamente dal DFA del LR(0) attraverso un processo chiamato l**ookahead propaganti**

REGOLE PER RISOLVERE L'AMBIGUITÀ

Spesso può essere comodo usare grammatiche ambigue ed usare delle regole per risolvere l'ambiguità.

Per esempio regole di precedenza ed associatività. La grammatica

E->E+E | E*E | (E) | id

(è ambigua poiché non specifica la precedenza e l'associatività tra gli operatori)

La grammatica non ambigua equivalente è:
E->E+T | T
T-> T * F | F
F -> (E) | id

La prima è preferibile perché:

- Si possono cambiare precedenza e associatività senza cambiare le produzioni
- Il parser con meno produzioni è più veloce.

Stabilire delle regole di precedenza e di associatività fa risolvere al parser i conflitti.

Un altro esempio riguarda l'ambiguità del "dangling else".

PROPRIETÀ DEI LINGUAGGI E DELLE GRAMMATICHE LR(K)

1. La famiglia dei linguaggi verificabili da parser deterministici coincide con quella dei linguaggi generati dalle grammatiche LR(1). Ciò non significa che ogni grammatica il cui linguaggio è deterministico, sia necessariamente LR(1): potrebbe richiedere una prospezione di lunghezza k>1; tuttavia esisterà una grammatica equivalente LR(1);
2. La famiglia dei linguaggi generati dalle grammatiche LR(k) coincide con quella dei linguaggi generati da LR(1). Quindi un linguaggio context free ma non-deterministico non può avere una grammatica LR(k) per nessun k.
3. Per ogni k>=1, esistono grammatiche che sono LR(k) ma non LR(k-1).
4. Data una grammatica, è indecidibile se esista un k>0 per cui tale grammatica risulti LR(k); di conseguenza non è decidibile se il linguaggio generato da una grammatica CF è deterministico. E' decidibile soltanto se k è fissato (si applica la costruzione del parser).

CONSIDERAZIONI FINALI SU LINGUAGGI E GRAMMATICHE LL(K) E LR(K):

1. Ogni linguaggio regolare è LL(1);
2. Ogni linguaggio LL(k) è deterministico, ma vi sono linguaggi deterministici per cui non esiste alcuna grammatica LL(k);
3. Per ogni k>=0, una grammatica LL(k) è anche LR(k);
4. Le grammatiche LL(1) e LR(0) non sono incluse una nell'altra;
5. Quasi tutte le grammatiche LL(1) sono LALR(1)

DOMANDE A RISPOSTA APERTA

- Cosa si intende per front end e back end di un compilatore?
- Qual è il ruolo dell'analizzatore lessicale in un compilatore?

ESERCIZI PER CASA

ESERCIZI

ESERCIZI

Siano date la grammatica G e la stringa badc. Si applichi l'algoritmo di Earley per stabilire se la stringa appartiene al linguaggio generato da G.

G:

S->Sc|bA

A->aA|d

La grammatica G non è LL(1). Si può trasformare in una grammatica LL(1)?

La grammatica è LR(0)?

Effettuare il parsing in tutti i modi possibili.

ESERCIZI

Sia G una grammatica avente X come assioma, {a,b} simboli terminali e le cui produzioni sono di seguito descritte.

G:

$X \rightarrow AA$

$A \rightarrow aA \mid b$

Rispondere alle seguenti domande, motivando le risposte.

. La grammatica g è LR(0)?

. La grammatica è LR(1)?

ESERCIZI

Sia G una grammatica avente S come assioma, {a,b,c,d} simboli terminali e le cui produzioni sono di seguito descritte.

G:

S → TV

T→ a | b

V → d c V | d

La grammatica G è LR(0)?

ESERCIZIO

Sia G una grammatica avente S come assioma, {a,b,c,d} simboli terminali e le cui produzioni sono di seguito descritte.

G:

$S \rightarrow Bd \mid Dc \mid bBc \mid bDd$

$B \rightarrow a$

$D \rightarrow a$

Rispondere alle domande seguenti

1. Dopo aver costruito la tabella LR(1), dedurre se la grammatica è LR(1).
2. La grammatica è LALR(1)? Perché?

ESERCIZIO

Sia data la grammatica G descritta di seguito. Si esegua il riconoscimento della stringa bcc utilizzando l'algoritmo di Earley. Qual è la complessità di tempo dell'algoritmo per questa grammatica?

G:

S -> T|X

T-> bTcc | bcc

X-> bXc | bc

Qual è il linguaggio l(g) generato da g?

. Date le caratteristiche della grammatica g e del linguaggio, è possibile rispondere alle seguenti domande:

1. G è LR(0)?
2. G è Lr(k) per qualche k?

Nota: l'esercizio non richiede la costruzione di tabelle di parsing o di automi LR(0) o LR(1),

ESERCIZI

Si consideri la grammatica G che ha il simbolo S come assioma, {a,b,c,d,f} simboli terminali e le seguenti regole di produzione:

G: $S \rightarrow CB$

$B \rightarrow b \mid f$

$C \rightarrow AB \mid DAB \mid \varepsilon$

$A \rightarrow Aa \mid a$

$D \rightarrow c \mid d$

Si risponda alle seguenti domande:

1. Per quale motivo per tale grammatica non è possibile effettuare un parsing di tipo LL(1)? G è LR(0)?
2. Si trasformi la grammatica G in una grammatica G' adatta all'analisi sintattica discendente. Si costruisca la tabella di parsing LL(1) per tale grammatica. La grammatica G' è LL(1)?
3. Si descriva il funzionamento di un generico parser LL(1).

ESERCIZI

Sia data la seguente grammatica *G*:

$S \rightarrow iAi \mid aAa \mid bAb \mid aBb \mid bBa$

$B \rightarrow i$

$A \rightarrow i$

Verificare se essa genera un linguaggio di tipo:

. LL(1)

. LR(0)

. LR(1)

. LALR(1)

ESERCIZIO

Data la grammatica *G*:

S->a|ab

E' LR(0)? E' LR(1)? E' LALR(1)?

ESERCIZI CON FLEX

ESERCIZIO CON FLEX

Con l'ausilio di flex scrivere un analizzatore lessicale che effettui la trasformazione di un testo scritto in un Linguaggio, chiamato linguaggio AA. Tale linguaggio è case-sensitive e le parole del linguaggio sono costituite solo da caratteri alfanumerici. Due parole possono essere separate l'una dall'altra da spazi, tabulazioni o newline. Ogni parola che contiene almeno un carattere diverso da quelli alfanumerici è considerata una parola non appartenente al lessico.

Le parole del lessico del linguaggio vengono trasformate come di seguito descritto:

. Ogni parola w costituita da lettere maiuscole che inizia per vocale viene trasformata in una parola ottenuta da w rimuovendone le occorrenze di sequenze di consonanti consecutive di lunghezza dispari. Le sequenze di consonanti consecutive di lunghezza pari vengono trasformate in minuscolo. Per esempio la parola ABSFBERTPLDAWRRTDDORR viene trasformata in absfbeawrrtddorr.

. Ogni parola costituita da lettere minuscole e cifre e che iniziano per lettera minuscola viene trasformata in una parola in cui, se le occorrenze di cifre consecutive costituiscono un numero naturale pari, questo viene trasformato nella sequenza di cifre che rappresentano la metà del suddetto numero, altrimenti viene trasformato nel numero successivo.

Per esempio, la parola cd34rftsy567errer20r viene trasformata in cd17rftsy568errer10r.

. Tutte le altre parole del lessico vengono trasformate nella parola ottenuta ripetendo il primo simbolo della parola tante volte quanti sono i simboli della parola stessa. Per esempio, la parola oeriiioodivjk34hhfjj4k5 viene trasformata in 000000000000000000000000.

Le parole che non fanno parte del lessico vengono lasciate inalterate. Alla fine della trasformazione il programma deve riportare il numero delle parole del lessico e di quelle che non appartengono al lessico.

¢ Per esempio se il file in input è:

```
ABSFBERTPLDAWRRTDDORR hfjkdjfdh553552%&&5cchhsGhskdlaks89404345asd
0eriiioodivjk34hhfjj4k5 CD34RFTSY567ERRER20R cd34rftsy567errer20r
```

Il file in output è:

Absfbeawrrtddorr hfjkdjfdh553552%&&5cchhs Ghskdlaks89404346asda 0000000000000000000000 CCCCCCCCCCCCCCCCCCCC cd17rftsy568errer10r

Lessico: 5

No lessico: 1