UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Dipartimento di Ingegneria
Corso di laurea magistrale
"Ingegneria Informatica
e dell'Automazione"

Riconoscitori LR(1)

Linguaggi e traduttori

A.A. 2017/18

Prof. Marco Gavanelli

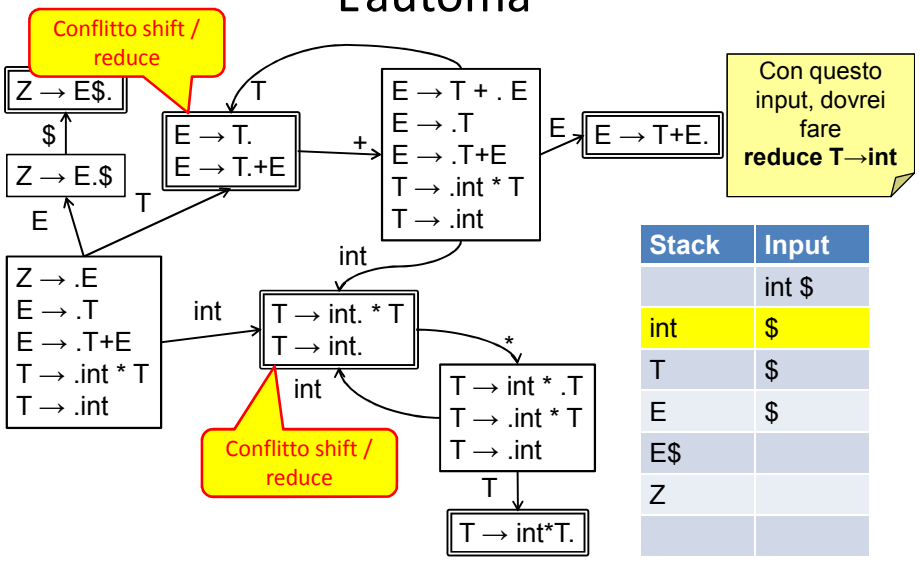
Esercizio

- Si scriva l'automa LR(0) per la grammatica:

  E → T + E | T

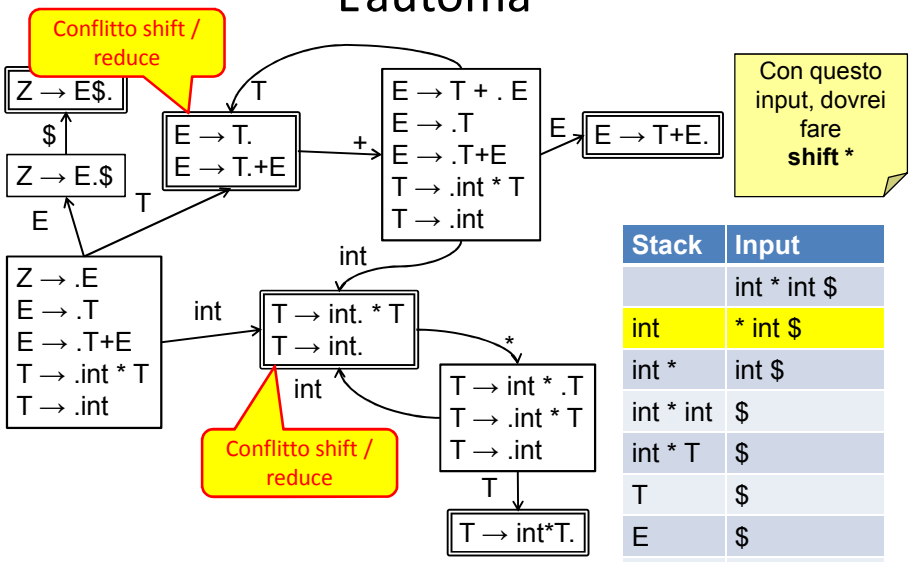
  T → int * T | int
- Ci sono dei conflitti?
- Si mostri il comportamento dell'automa con input
  - int
  - int * int

L'automa



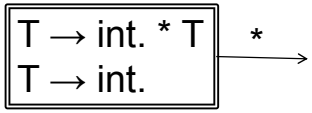
Stack	Input
	int $
int	$
T	$
E	$
E$	
Z	

L'automa



Stack	Input
	int * int $
int	* int $
int *	int $
int * int	$
int * T	$
T	$
E	$
E$	
Z	

Conflitti



- Se c'è uno stato finale (di riduzione) che ha degli archi uscenti etichettati con un simbolo terminale, non sappiamo scegliere se fare shift o reduce. Si dice che c'è un conflitto shift/reduce. Questo avviene se in uno stato ci sono un item di riduzione e uno di shift:
  - $X \rightarrow \alpha.$ e $Y \rightarrow \beta.t\omega$
- Si ha un conflitto reduce/reduce se in uno stato ci sono due diversi item reduce:
  - $X \rightarrow \alpha.$ e $Y \rightarrow \beta.$
- Se ci sono conflitti, la grammatica non è LR(0).
- Un parser LR(**0**) non utilizza l'input per decidere quale mossa fare: usa solo lo stack
- Per migliorare il parser, si può cercare di distinguere in base al prossimo simbolo di input: analisi LR(**1**)

ANALISI LR(1)

La maggior parte dei linguaggi di interesse è descrivibile con grammatiche LR(1), che richiedono

- **i contesti LR(0)**
- ***il simbolo successivo***

per "guidare a colpo sicuro" il processo di riduzione, risolvendo i conflitti.

In generale, l'analisi di una grammatica LR(k) si basa su un algoritmo analogo a quello del caso LR(0), tranne per il fatto che *tutte le riduzioni sono ritardate di k simboli.*

Le definizioni di contesto e contesto sinistro si estendono analogamente al caso K>0, come pure il procedimento operativo, che diventa però assai più complesso sia computazionalmente, sia strutturalmente, per l'elevato numero di configurazioni di cui occorre tenere conto.

CONTESTI LR(k) e AUTOMA CARATTERISTICO

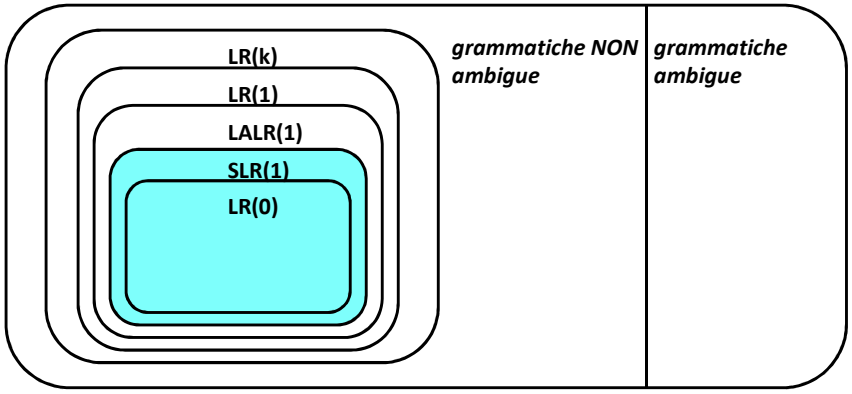
In linea di principio, si procede come nel caso LR(0):

- **si calcolano le espressioni regolari per i contesti LR(k)**
- **e si usano per costruire l'automa caratteristico.**

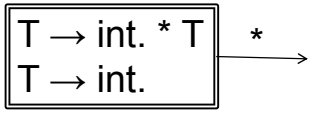
Tuttavia tale approccio, già non banale nel caso k=0, *diviene ancora più lungo e complesso* quando k>0.

- **Una grammatica G con *n* metasimboli e *t* terminali comporta una grammatica dei contesti sinistri LR(*k*) avente potenzialmente $(n\text{-}1)t^k + 1$ metasimboli: nel caso LR(0) sarebbero stati al più *n*.**
- **Per un tipico linguaggio con 50–100 terminali, ciò significa una grammatica dei contesti sinistri LR(1) *50-100 volte più grande* del caso LR(0): *praticamente intrattabile.***
- **Per questo, l'approccio LR(1) "completo" è spesso sostituito da versioni semplificate, più trattabili.**

Simple LR(1) o SLR(1)



Conflitti



- Conflitto shift/reduce: in uno stato ci sono un item di riduzione e uno di shift:
  - X → α. e Y → β.tω
- Conflitto reduce/reduce: in uno stato ci sono due diversi item reduce:
  - X → α. e Y → β.
- Per migliorare il parser, si può cercare di distinguere in base al prossimo simbolo di input
  - Chiaramente, lo shift viene scelto se il prossimo simbolo è quello indicato sull'arco
  - In quali casi si applica reduce?

Simple LR(1)

- Idea semplice per decidere quando applicare reduce basandosi sul simbolo successivo in input:
- Se faccio una reduce con una regola A → α, significa che da una situazione

  β α |xwzy

  passo a

  β A |xwzy
- Questo significa che il primo simbolo x nell'input deve seguire il nonterminale A, ovvero deve appartenere al FOLLOW(A)

LR(0) parsing

- Se
  - lo stack contiene αβ
  - il prossimo input è t
  - l'automa con input αβ termina nello stato s
- riduci con X → β se
  - s contiene l'item X→β.
- effettua uno shift se
  - s contiene l'item X → β.tω

stack | input

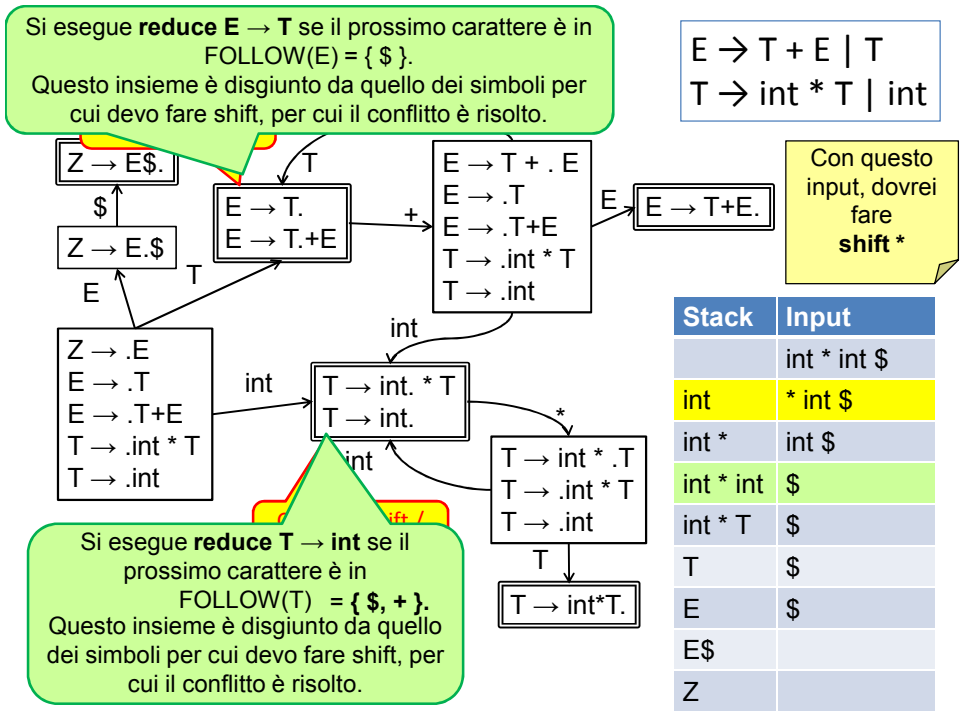
αβ | tx

s: X→β.

s: X→β.tω

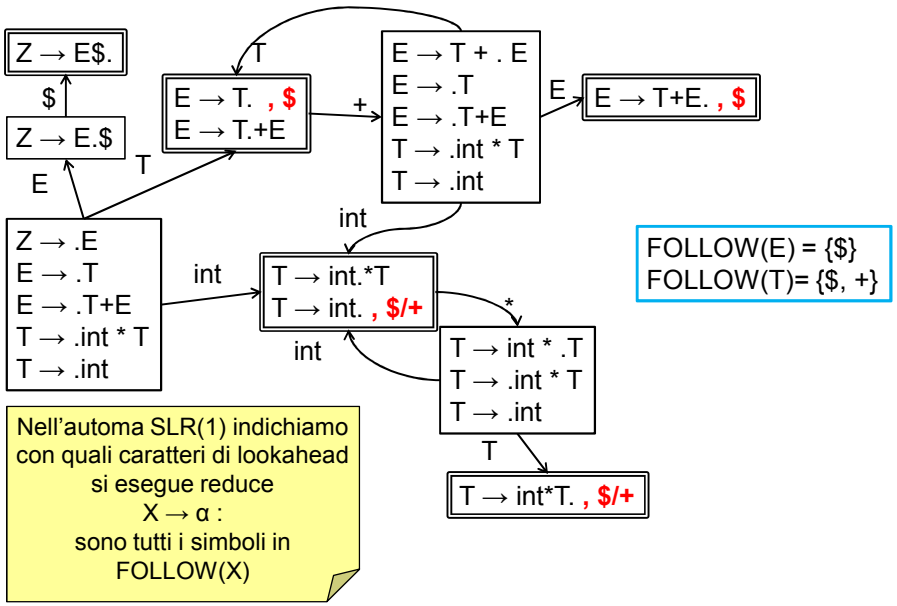
SLR(1) parsing

- Se
  - lo stack contiene αβ
  - il prossimo input è t
  - l'automa con input αβ termina nello stato s
- riduci con X → β se
  - s contiene l'item X→β.
  - **t ∈FOLLOW(X)**
- effettua uno shift se
  - s contiene l'item X → β.tω



Stack	Input
	int * int $
int	* int $
int *	int $
int * int	$
int * T	$
T	$
E	$
E$	
Z	

L'automa



Conflitti e SLR(1)

- Se con l'algoritmo precedente non ci sono conflitti, allora la grammatica è SLR(1).
- Altrimenti, se c'è almeno uno stato che contiene:
  - X → α.yω (ovvero, si può fare shift y)
  - Y → α. (ovvero, uno stato di riduzione)
  - y ∈ FOLLOW(Y)

  allora c'è un conflitto shift-reduce e la grammatica non è SLR(1).

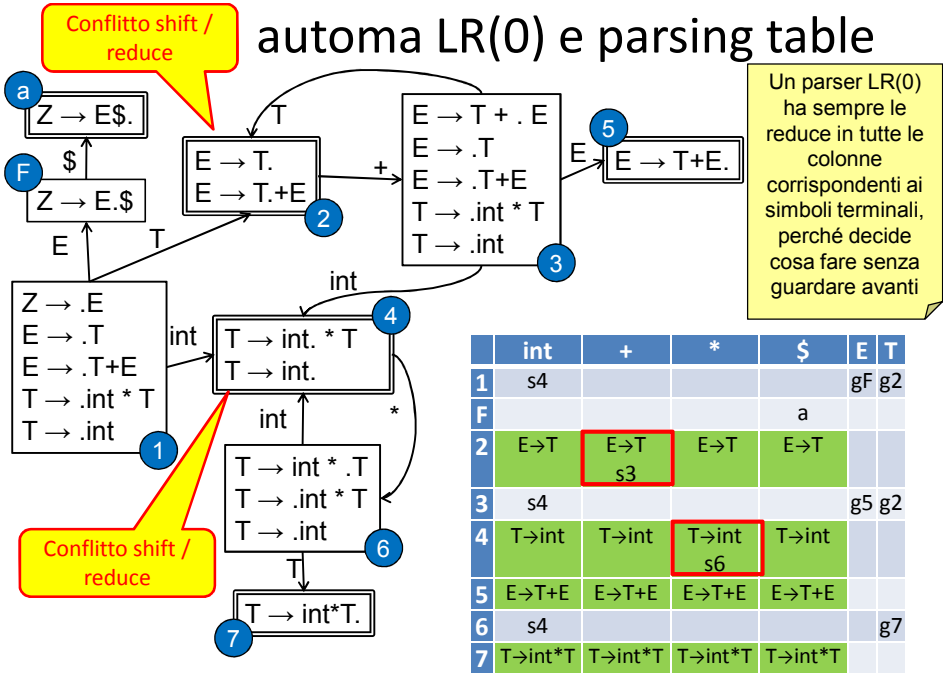
X → α.yω
Y → α. , y
y

- Se c'è almeno uno stato che contiene
  - X → α.
  - Y → β.
  - FOLLOW(X) ∩ FOLLOW(Y) ≠ ∅

  allora c'è un conflitto reduce-reduce e la grammatica non è SLR(1).

X → α. , y
Y → β. , y

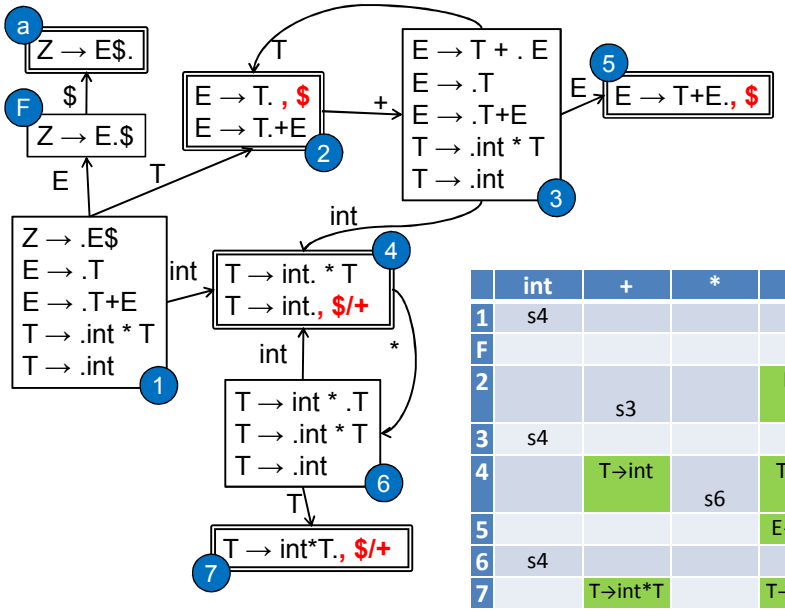
automa LR(0) e parsing table



Un parser LR(0) ha sempre le reduce in tutte le colonne corrispondenti ai simboli terminali, perché decide cosa fare senza guardare avanti

	int	+	*	$	E	T
1	s4				gF	g2
F				a		
2	E→T	E→T s3	E→T	E→T		
3	s4				g5	g2
4	T→int	T→int	T→int s6	T→int		
5	E→T+E	E→T+E	E→T+E	E→T+E		
6	s4					g7
7	T→int*T	T→int*T	T→int*T	T→int*T		

automa SLR(1) e parsing table



	int	+	*	$	E	T
1	s4				gF	g2
F				a		
2		s3		E→T		
3	s4				g5	g2
4		T→int	s6	T→int		
5				E→T+E		
6	s4					g7
7		T→int*T		T→int*T		

Esempio

- Sia data la grammatica

  D → bBd | Cb

  C → bAa

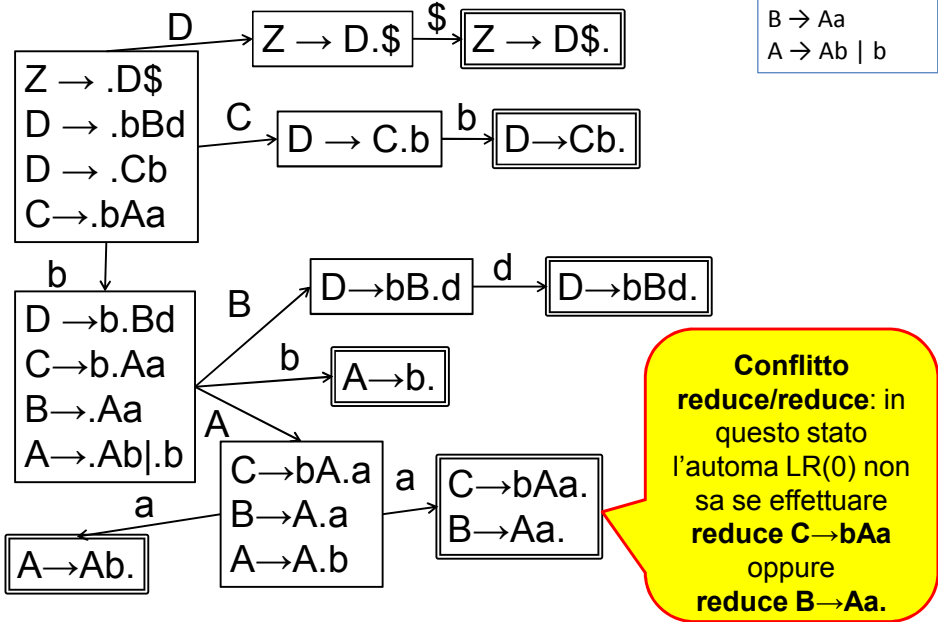
  B → Aa

  A → Ab | b

- Si scriva l'automa LR(0) e si mostrino eventuali conflitti.
- Passando a SLR(1) i conflitti vengono risolti?

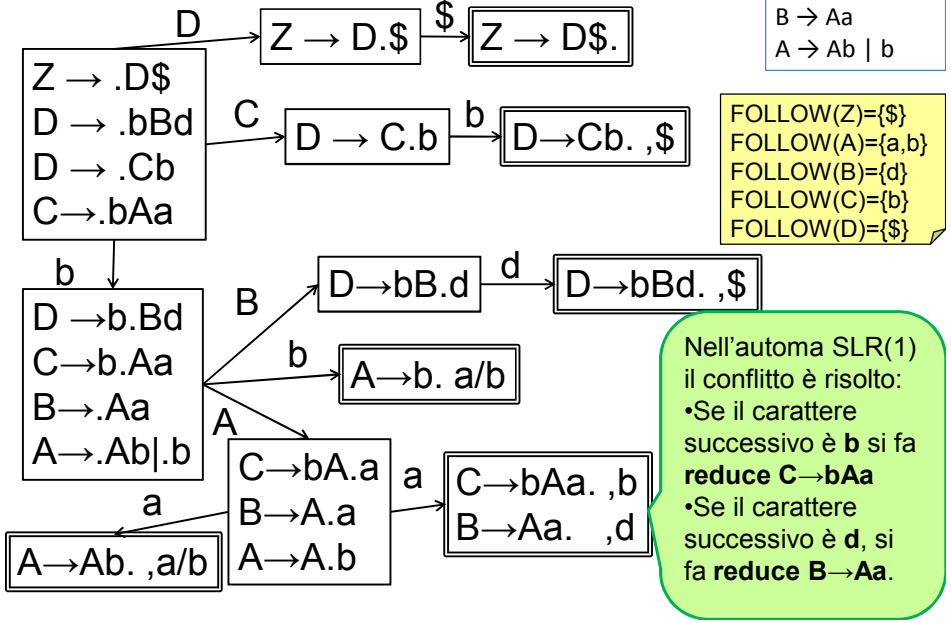
Automa LR(0)

Z → D$
D → bBd | Cb
C → bAa
B → Aa
A → Ab | b



Automa SLR(1)

Z → D$
D → bBd | Cb
C → bAa
B → Aa
A → Ab | b



Esempio

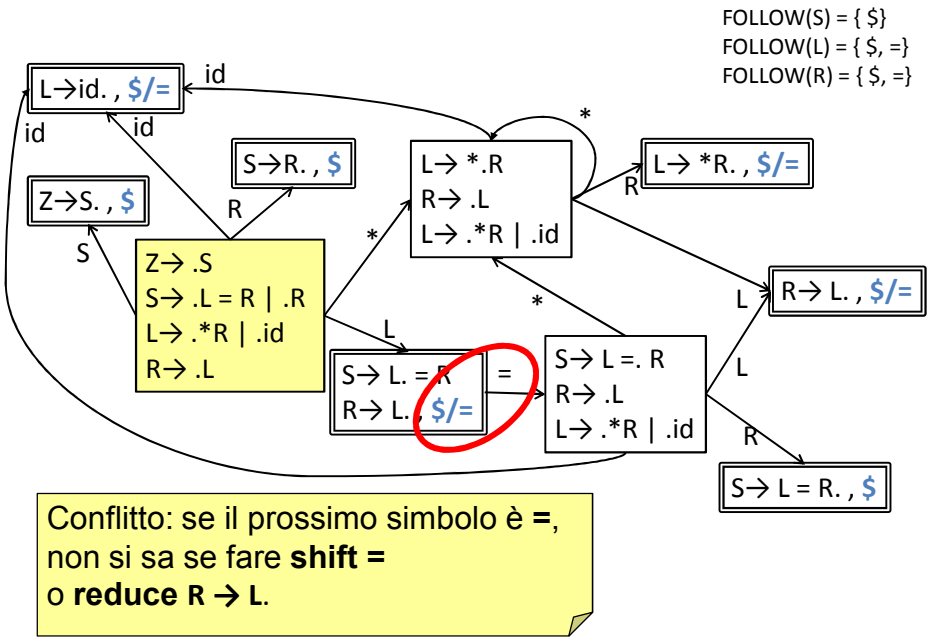
- Sia data la grammatica:

  S → L = R | R

  L → *R | id

  R → L

- Si scriva l'automa SLR(1) e si indichino eventuali conflitti

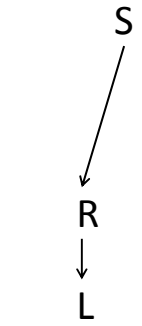


Conflitto: se il prossimo simbolo è **=**, non si sa se fare **shift =** o **reduce R → L**.

Ragionando sulla grammatica …

- Nel linguaggio generato dalla grammatica, il simbolo **=** può comparire 0 o 1 volta.
- La riduzione **R → L** significa che, al livello superiore, si può arrivare a S in due modi
- **Modo 1**: riducendo **S → R**
- In questo caso, l'**=** non compare. Ma allora perché fare **reduce R → L** in corrispondenza del simbolo **=** ?

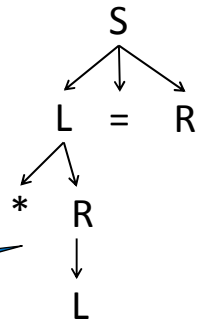
S → L = R | R
L → *R | id
R → L



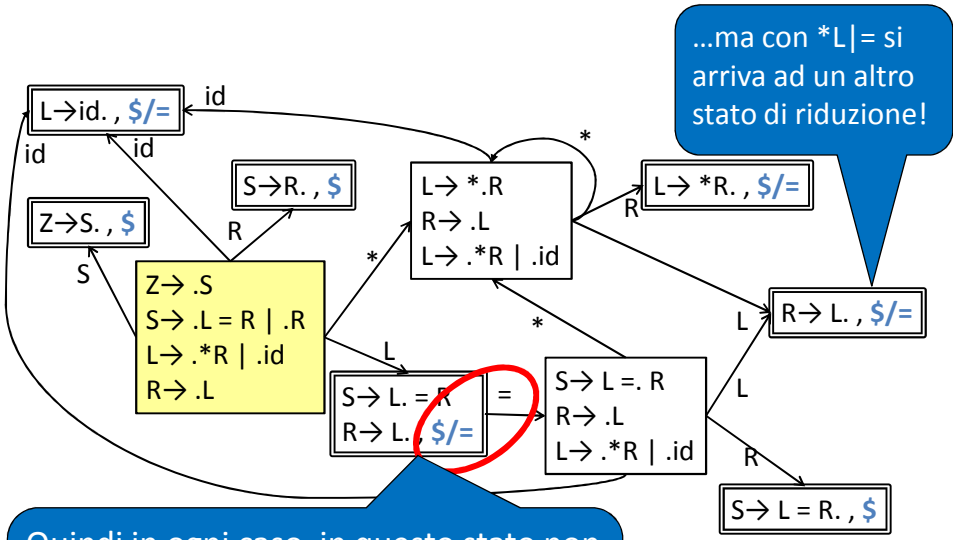
Ragionando sulla grammatica …

- Nel linguaggio generato dalla grammatica, il simbolo **=** può comparire 0 o 1 volta.
- La riduzione **R → L** significa che, al livello superiore, si può arrivare a S in due modi
- **Modo 2**: con un primo passaggio di **reduce L → *R**
- In questo caso, prima ci deve essere stato un *, ma allora sarei arrivato in un altro stato!

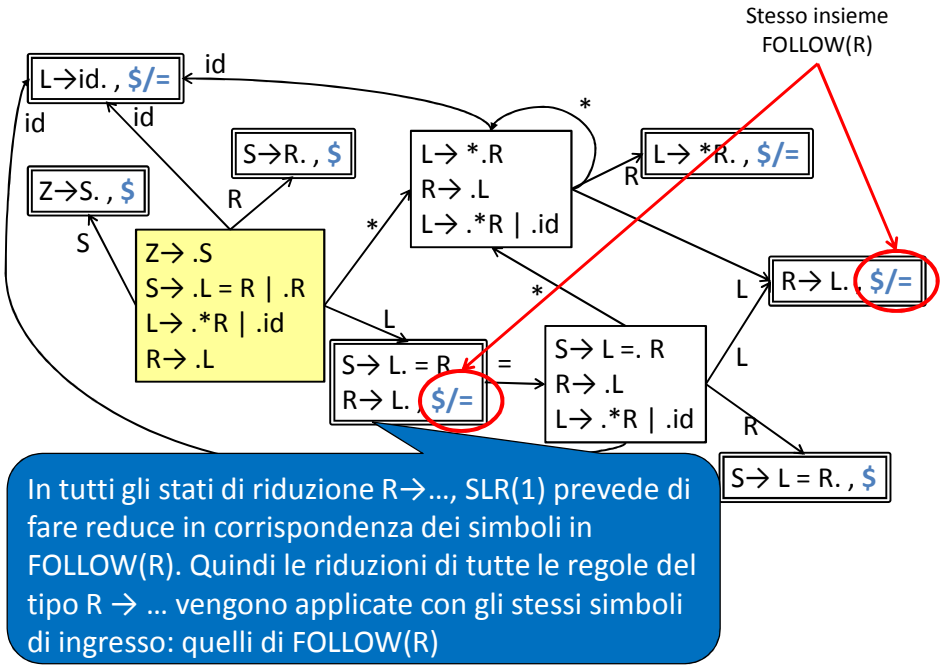
S → L = R | R
L → *R | id
R → L



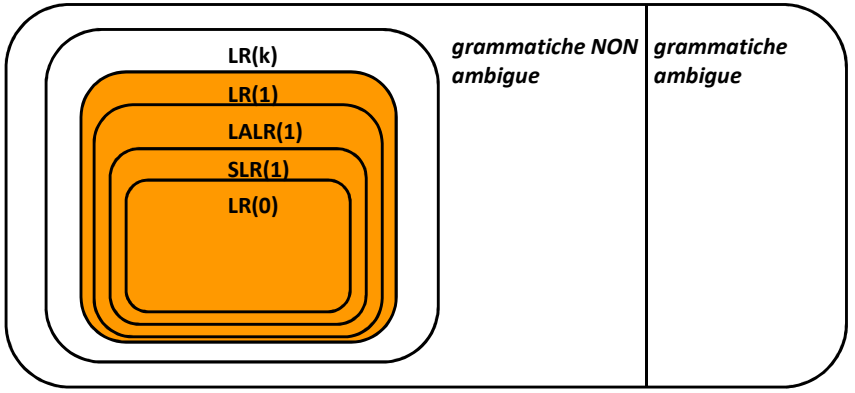
Qui la forma di frase comincia con *L|= …



Quindi in ogni caso, in questo stato non si fa mai reduce se il prossimo simbolo di input è '='. SLR(1) non se ne accorge e genera il conflitto



Canonical LR(1) o LR(1)

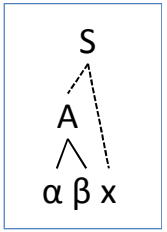


LR(1)

- SLR(1) prescrive di fare **reduce X→α** se il prossimo simbolo di input è nel **FOLLOW(X)**. Quindi le riduzioni per regole **X → …** sono tutte applicate con lo stesso insieme **FOLLOW(X)**
- SLR(1) è una semplificazione di LR(1)
  - Quindi le grammatiche SLR(1) sono un sottoinsieme delle grammatiche LR(1)
- Invece di indicare (come in SLR(1)) solo negli item di riduzione quali sono i simboli successivi (**A → α. , x**)
- in LR(1) si indica in tutti gli item (anche quelli non di riduzione) quali saranno i simboli successivi (**A → α.β , x**)

LR(1): procedimento operativo

- Invece di indicare (come in SLR(1)) solo negli item di riduzione quali sono i simboli successivi accettabili per quella regola (**A → α. , x**)
- si indica in tutti gli item (anche quelli non di riduzione) quali saranno i simboli successivi (**A → α.β , x**)
- Un item RL(1) è quindi costituito da
  - un item LR(0) **A → α . β**
  - un simbolo **x**, che rappresenta il simbolo che può seguire **A** e **αβ** in una derivazione canonica destra



- In generale, un item RL(*k*) è costituito da
  - un item LR(0)
  - una sequenza di *k* simboli, che rappresenta una sequenza di simboli che può seguire **A** e **αβ** in una derivazione canonica destra

Item RL(1)

- Un item RL(1)

$$A \rightarrow \alpha.\beta \;,\; a$$

- rappresenta il fatto che è possibile avere una derivazione canonica destra:

$$S \rightarrow^* \delta A w \rightarrow \delta\alpha\beta w$$

- in cui il primo carattere di **w** è **a**
  (**a** potrebbe anche essere il terminatore **$**)

Stati RL(1)

s

A→α.Bβ , a
B→.γ , b

- Uno stato RL(1) è costituito da un insieme di item RL(1).
  Per costruire lo stato *s* si effettua la seguente operazione di **chiusura**:

per ogni item [A→α.Bβ , a] in *s*
  per ogni produzione B → γ in G'
    per ogni terminale b in FIRST(βa)
      aggiungi [B→. γ , b] ad *s*

Le parti in nero sono uguali anche per la costruzione degli stati LR(0)
Le parti in azzurro sono specifiche per LR(1)
G' è la grammatica aumentata con la produzione Z → S$

Automa RL(1)

Per costruire l'automa RL(1), si procede aggiungendo stati all'automa in maniera analoga al caso RL(0) (la differenza principale è nella costruzione degli stati, come visto prima)

- si parte dallo stato iniziale
  (costruito con la chiusura a partire dall'item [**Z → .S$** , **?**]),
- per ogni stato ***I*** che contiene un item

  [**A→α.Xβ** , **a**]

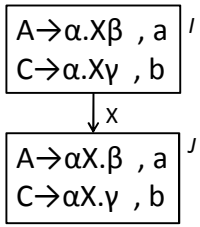
  dove **X** può essere un terminale o un nonterminale
  - Si aggiunge un arco etichettato con **X** verso un altro stato ***J***.
  - Per ogni item in ***I*** del tipo:

    [**A→α.Xβ** , **a**]

    lo stato ***J*** contiene l'item

    [**A→αX.β** , **a**]

    e vi si applica quindi la chiusura



PROCEDIMENTO OPERATIVO

Per <u>adattare il procedimento operativo</u>, in ogni stato si dovrà ora tenere conto anche del simbolo successivo *(lookahead symbol)*

NOTAZIONE: in ogni stato, si specifica anche il simbolo successivo che rende valida l'azione.

Solitamente, si adotta anche un *simbolo jolly* (? o *) per denotare il caso "*ANYTHING*".

Z → . S$	**?**
S → . aSAB	**a**
S → . BA	**a,=**
B → . b	**$**

La costruzione degli stati diventa più complessa del caso LR(0), perché ora, a ogni passaggio, bisogna anche computare ***il nuovo insieme di caratteri di lookahead***, svolgendo in pratica passo per passo le stesse elaborazioni che portano alla costruzione dei contesti LR(1).

Lo vediamo direttamente sull'esempio.

PROCEDIMENTO OPERATIVO (1/9)

Si consideri la seguente grammatica:

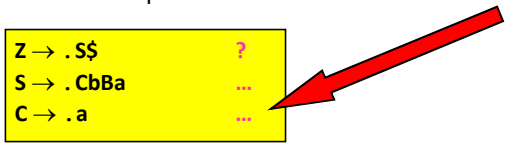
Z → S$	S → CbBA	A → ab | Aab
B → C | Db	C → a	D → a

Si parte come sempre dalla regola di top-level: lì <u>per ipotesi</u> il set di caratteri di look-ahead è *ANYTHING*, in quanto in realtà non si andrà mai oltre il terminatore $.

Si mette poi in gioco la regola S → . CbBA

Come si calcola il suo lookahead set?

Inoltre, poiché dopo il cursore c'è C, occorre mettere in gioco anche la produzione C → . a e anche qui c'è da calcolare il lookahead set.



PROCEDIMENTO OPERATIVO (2/9)

Si consideri la seguente grammatica:

Z → S$	S → CbBA	A → ab | Aab
B → C | Db	C → a	D → a

CALCOLO DEL LOOK-AHEAD SET della produzione S → . CbBA

1) si guarda quali caratteri possono <u>seguire S</u> nella produzione di livello superiore <u>qui usata</u>, che è Z → S$; il solo possibile è $
2) si <u>concatena</u> con il look-ahead set attuale, ottenendo $?
3) si prende <u>l'iniziale</u> della stringa appena ottenuta: quindi, $

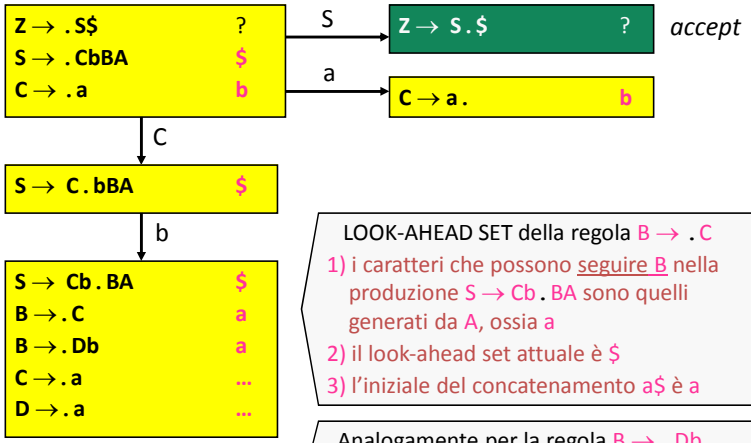
Z → . S$	?
S → . CbBa	$
C → . a	…

Z → . S$	?
S → . CbBa	$
C → . a	b

CALCOLO DEL LOOK-AHEAD SET della produzione C → . a

1) si guarda quali caratteri possono <u>seguire C</u> nella produzione di livello superiore qui usata, che è S → . CbBA : qui, il solo possibile è b
2) si <u>concatena</u> con il look-ahead set attuale ($), ottenendo b$
3) si prende l'iniziale, che è b

PROCEDIMENTO OPERATIVO (3/9)



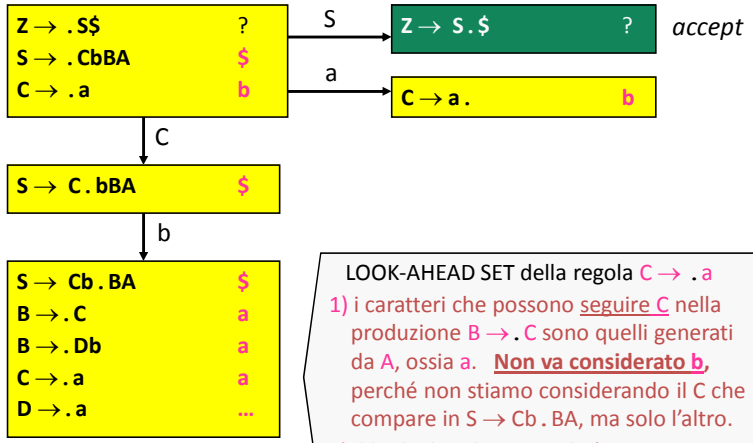
LOOK-AHEAD SET della regola B → . C

1) i caratteri che possono <u>seguire B</u> nella produzione S → Cb . BA sono quelli generati da A, ossia a
2) il look-ahead set attuale è $
3) l'iniziale del concatenamento a$ è a

Analogamente per la regola B → . Db

1) il carattere che può <u>seguire B</u> è sempre a
2) il look-ahead set attuale è $
3) l'iniziale del concatenamento a$ è a

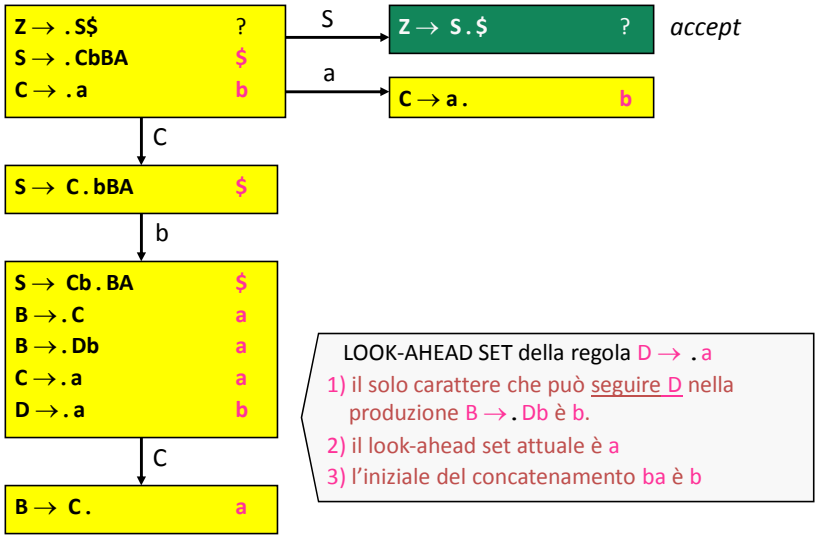
PROCEDIMENTO OPERATIVO (4/9)



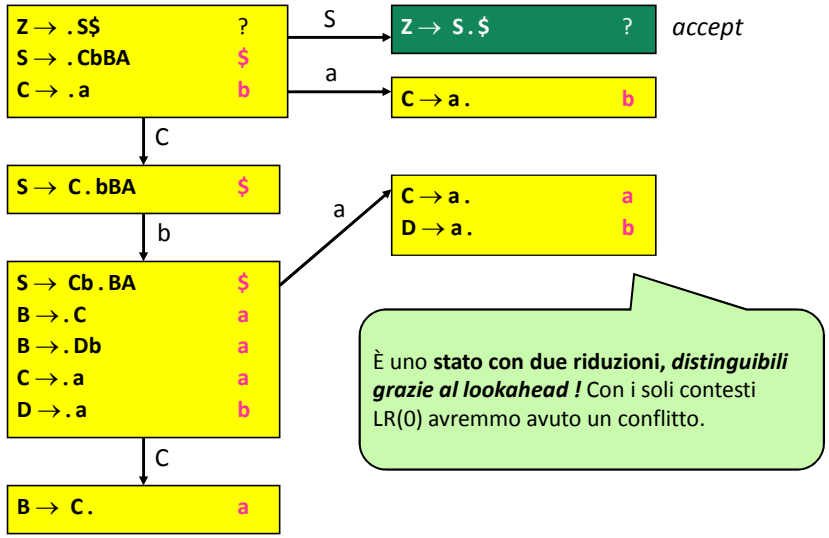
LOOK-AHEAD SET della regola C → . a

1) i caratteri che possono <u>seguire C</u> nella produzione B → . C sono quelli generati da A, ossia a. **<u>Non va considerato b</u>,** perché non stiamo considerando il C che compare in S → Cb . BA, ma solo l'altro.
2) il look-ahead set attuale è a
3) l'iniziale del concatenamento aa è a

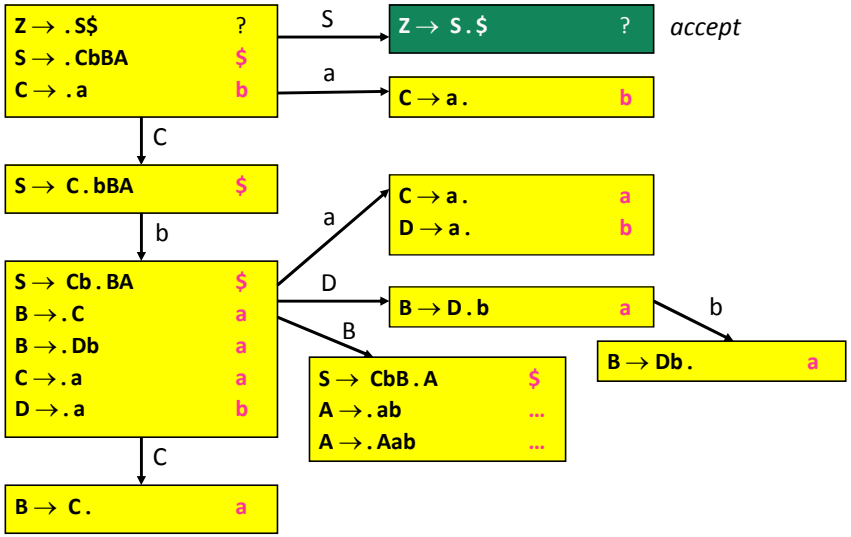
PROCEDIMENTO OPERATIVO (5/9)



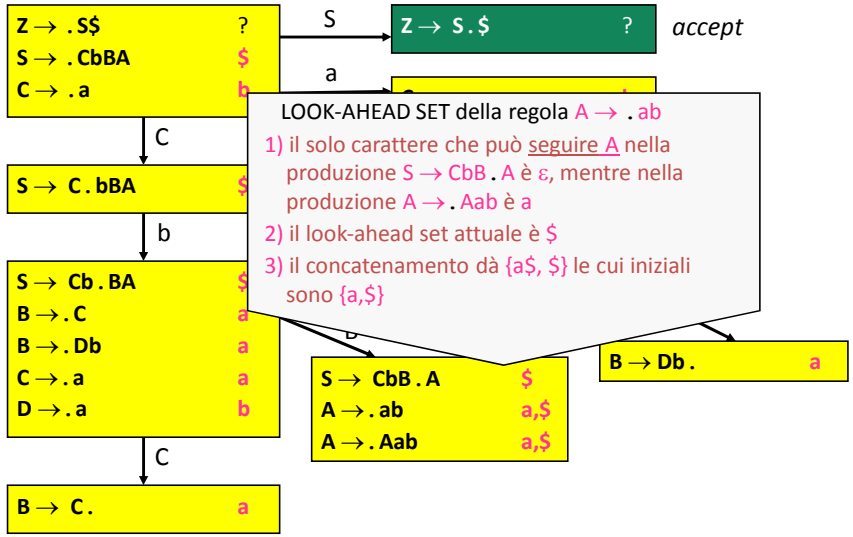
PROCEDIMENTO OPERATIVO (6/9)



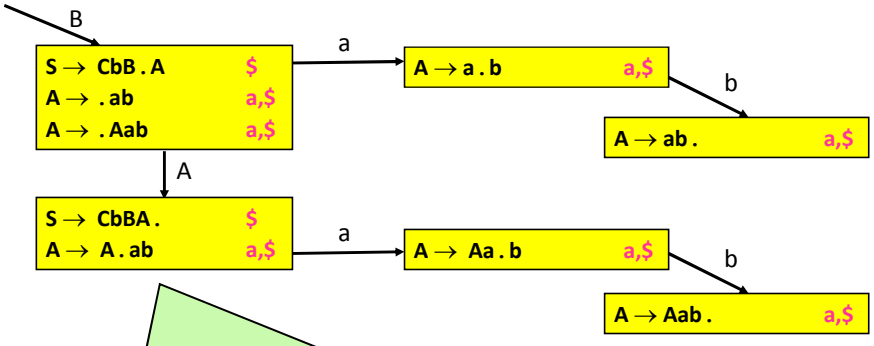
PROCEDIMENTO OPERATIVO (7/9)



PROCEDIMENTO OPERATIVO (8/9)



PROCEDIMENTO OPERATIVO (9/9)



È uno stato con uno shift e una riduzione, di nuovo distinguibili grazie al lookahead:
quando il cursore è al termine di una produzione, per quel simbolo di lookahead (qui, $) *si adotta sempre la riduzione*.

Riprendendo la grammatica

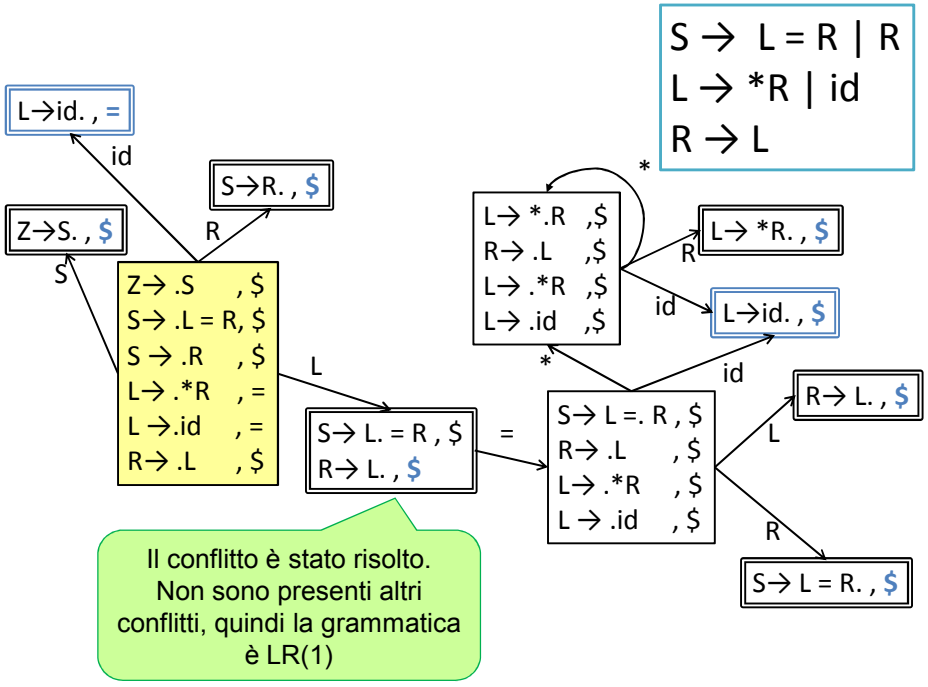
- Riprendiamo ora la grammatica lasciata in sospeso:

  S → L = R | R

  L → *R | id

  R → L

- e costruiamo l'automa LR(1)



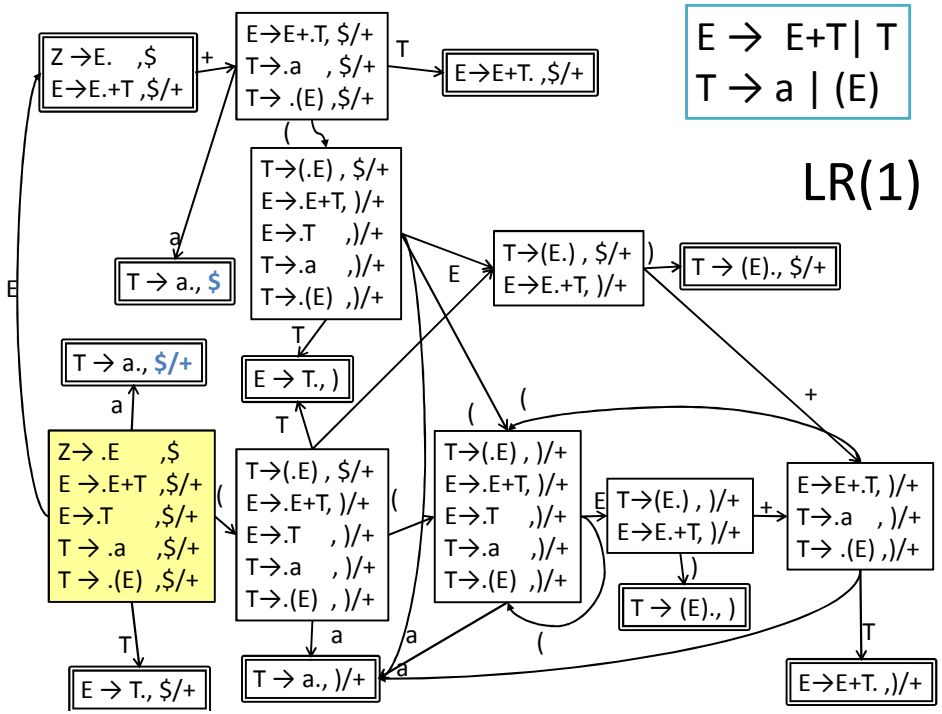
Grammatica delle espressioni

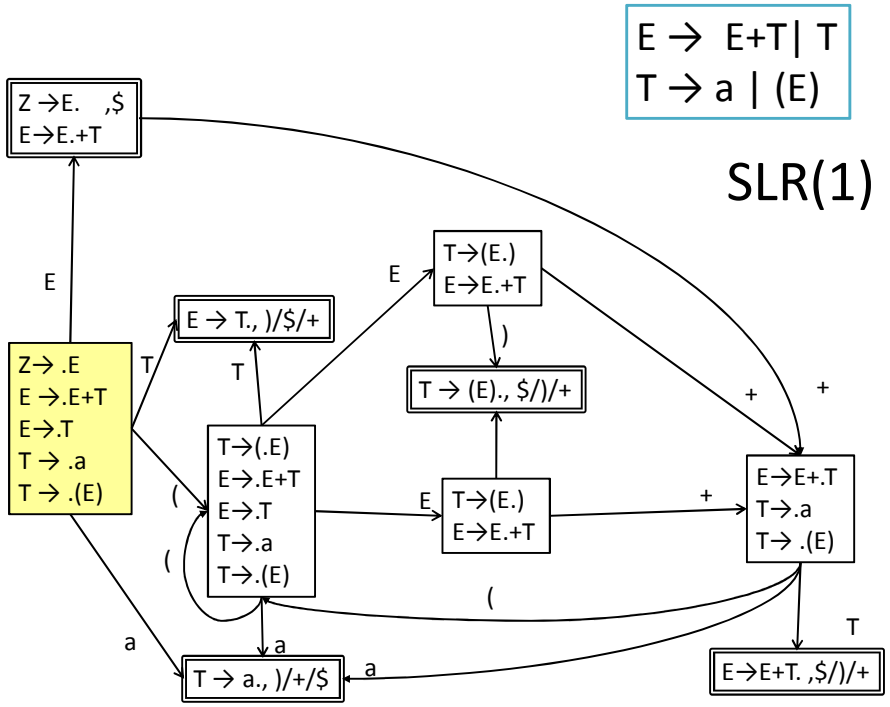
- Consideriamo ora un sottoinsieme della grammatica delle espressioni:

  E → E+T| T

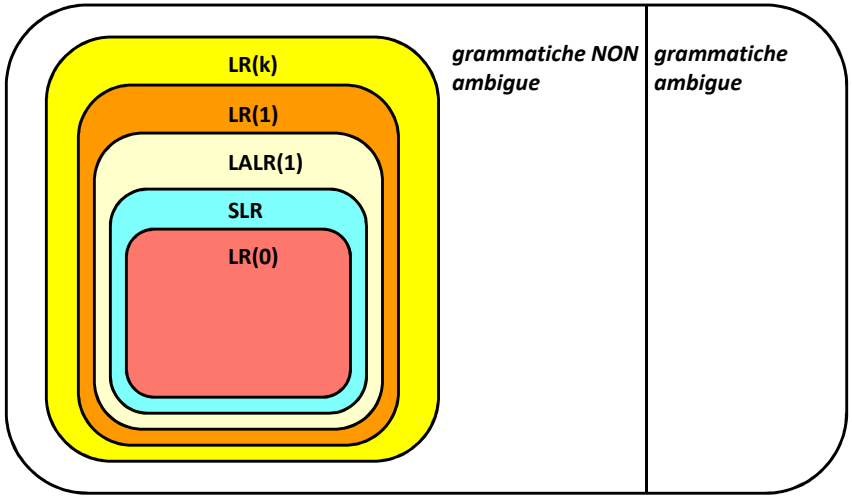
  T → a | (E)

- Mostriamo gli automi LR(1) e SLR(1).

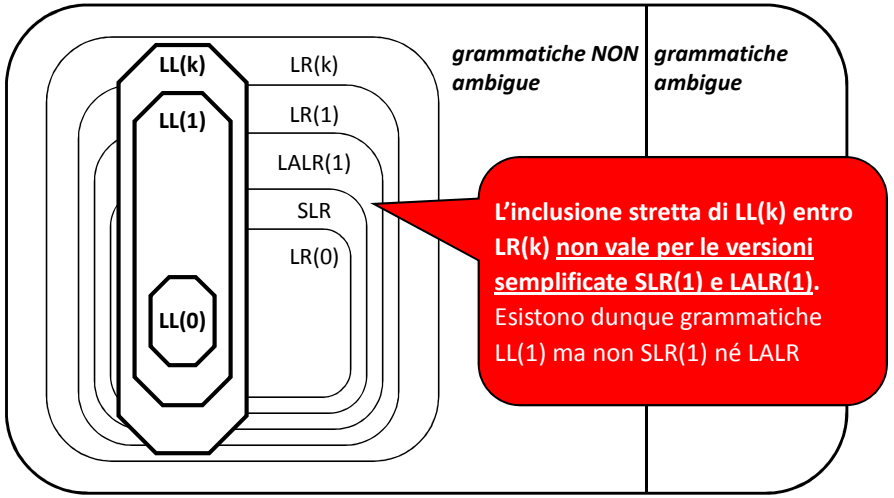




PARSING LR: RIEPILOGO (1/4)



PARSING LR: RIEPILOGO (2/4)



PARSING LR: RIEPILOGO (3/4)

RISULTATI "NEGATIVI"

- **Esistono grammatiche LR(1) che però non sono SLR(k) per nessun valore di k.** Un esempio è la nota grammatica:

  $Z \rightarrow S\$$ $\quad S \rightarrow CbBA$ $\quad A \rightarrow ab \mid Aab$

  $B \rightarrow C \mid Db$ $\quad C \rightarrow a$ $\quad D \rightarrow a$

- **Esistono grammatiche LL(1) che però non sono SLR(1).** Un esempio è dato dalla grammatica:

  $Z \rightarrow S\$$ $\quad S \rightarrow AaAb \mid BbBa$

  $A \rightarrow \varepsilon$ $\quad B \rightarrow \varepsilon$

  che è palesemente LL(1), ma non SLR(1).

PARSING LR: RIEPILOGO (4/4)

RISULTATI "POSITIVI"

- **Ogni grammatica SLR(k) è anche LR(k).**
  È conseguenza dell'inclusione dei contesti LR(k) nei contesti SLR(k).

- **Ogni grammatica LL(k) priva di produzioni inutili e aumentata della produzione $Z \rightarrow S\$$ è anche LR(k).**