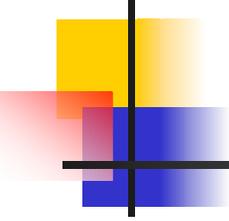


# Tabelle LALR

---

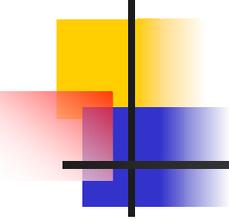
Costruzione delle tabelle LALR



# Metodo LALR

---

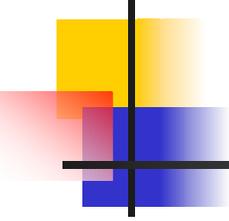
- Introduciamo l'ultimo metodo di costruzione di tabelle per il parsing LR
- Nome: *lookahead*-LR abbreviato in LALR
- Questo metodo è usato spesso dato che le tabelle che si ottengono sono sensibilmente più piccole di quelle ottenute con l'LR canonico
- I tipici costrutti dei linguaggi di programmazione possono venire facilmente catturati da una grammatica LALR



# Metodo LALR

---

- Lo stesso discorso valeva anche per le grammatiche SLR
- Tuttavia ci sono un certo numero di costrutti per i quali è meglio usare grammatiche LALR
- Le tabelle SLR e LALR hanno sempre lo stesso numero di stati
- Numero: alcune **centinaia** di stati per un linguaggio tipo il Pascal
- Per lo stesso linguaggio una tabella LR canonica ha alcune **migliaia** di stati



# Idea

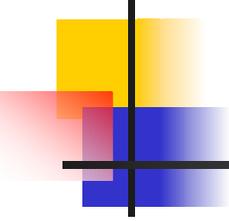
---

- Consideriamo la grammatica usata come esempio per illustrare la costruzione di tabelle LR canoniche:

$$S' \rightarrow S$$

$$S \rightarrow CC$$

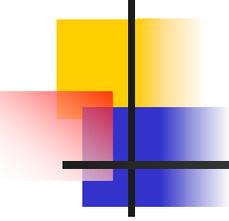
$$C \rightarrow cC \mid d$$



# Idea

---

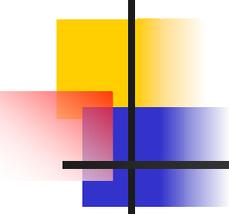
- Alcuni insiemi di item LR(1) che abbiamo costruito per questa grammatica avevano lo stesso insieme di item LR(0) ma differivano per i simboli di lookahead
- $I_4 = \{[C \rightarrow \bullet d, c], [C \rightarrow \bullet d, d] \}$
- $I_7 = \{[C \rightarrow \bullet d, \$]\}$



# Idea

---

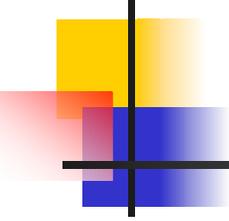
- Osserviamo meglio il comportamento di questi due stati durante il parsing
- La grammatica genera il linguaggio  $c^*dc^*d$
- Durante la lettura di un input  $cc\dots cdcc\dots cd$  il parser fa lo shift di tutto il primo gruppo di  $c$  e della prima  $d$  che le segue entrando nello stato 4 dopo aver letto questa  $d$
- A questo punto il parser ordina una riduzione con  $C \rightarrow d$  se il simbolo successivo è  $c$  o  $d$



# Idea

---

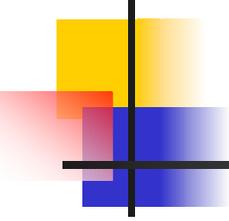
- È giusto: sia  $c$  che  $d$  possono essere l'inizio di  $c*d$
- D'altra parte, se  $\$$  segue la prima  $d$  c'è un errore: ad esempio la stringa  $ccd$  non è nel linguaggio
- E giustamente lo stato 4 segnala errore se il simbolo di input è  $\$$



# Idea

---

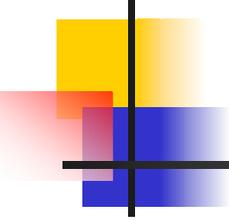
- Dopo la prima  $d$  il parser appila tutte le  $c$  seguenti e anche la seconda  $d$  entrando infine nello stato 7
- A questo punto il simbolo di input deve essere  $\$$  altrimenti la stringa data non è nel linguaggio (errore)
- È giusto che lo stato 7 comandi una riduzione con  $C \rightarrow d$  se l'input è  $\$$  e dia errore se l'input è  $c$  o  $d$ .



# Idea

---

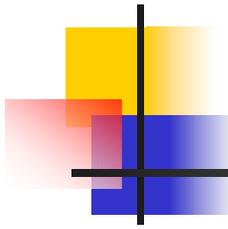
- Rimpiazziamo ora gli insiemi  $I_4$  e  $I_7$  con un unico insieme di item LR(1)  $I_{47}$
- $I_{47}$  è l'unione degli item di  $I_4$  e di  $I_7$
- $I_{47} = C \rightarrow d \bullet, c/d/\$$
- $\text{goto}(I_i, d) = I_{47}$  per  $i=0,2,3,6$
- Le azioni dello stato 47 sono: riduci con  $C \rightarrow d$  per ogni input



# Conseguenze

---

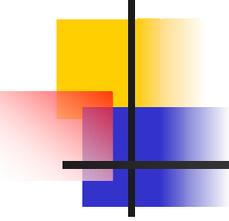
- Il parser così modificato si comporta *quasi* allo stesso modo di quello di partenza
- Potrebbe ridurre con  $C \rightarrow d$  in alcune circostanze nelle quali il parser originale avrebbe segnalato un errore (es: ccd o ccdcdc)
- L'errore però viene comunque rilevato in seguito (in effetti prima che venga appilato un altro simbolo terminale sullo stack)



# In generale

---

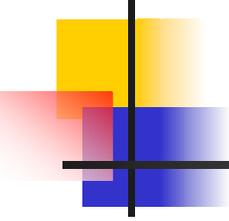
- Cerchiamo gli insiemi di item LR(1) che hanno lo stesso **core**, cioè lo stesso insieme di prime componenti
- Accorpiamo gli insiemi di item LR(1) con lo stesso core in un unico insieme di item LR(1)
- Nell'esempio precedente  $I_4$  e  $I_7$  avevano lo stesso core  $C \rightarrow d \bullet$



# In generale

---

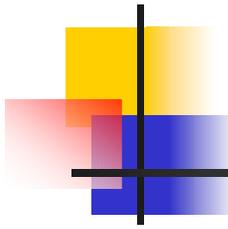
- Ad esempio anche gli insiemi  $I_3$  ed  $I_6$  dell'esempio precedente hanno lo stesso core  $\{C \rightarrow c \bullet C, C \rightarrow \bullet c C, C \rightarrow \bullet d\}$
- Anche  $I_8$  ed  $I_9$  hanno lo stesso core  $\{C \rightarrow c C \bullet\}$
- In generale un core è un insieme di item LR(0)



# In generale

---

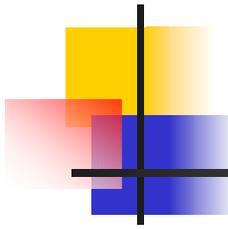
- Il core di  $\text{goto}(I, X)$  dipende solo dal core di  $I$
- Questo significa che i goto degli stati accorpati possono essere anche essi accorpati
- Quindi non c'è problema nella ridefinizione della funzione goto
- La tabella action va modificata in accordo ai nuovi item LR(1) (con gli accorpamenti)



# Conseguenze

---

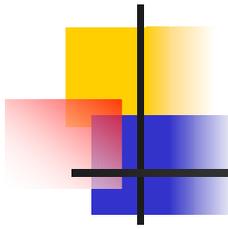
- Supponiamo di avere una grammatica LR(1)
- I suoi insiemi di item LR(1) non producono conflitti
- Se accorpiamo gli insiemi con lo stesso core ci dovremmo aspettare che i nuovi insiemi producano conflitti
- E invece non è così, almeno per i conflitti shift/reduce



# Conseguenze

---

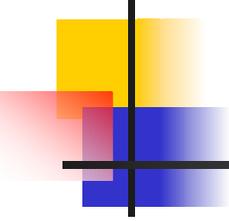
- Supponiamo infatti di avere un conflitto shift/reduce in uno stato accorpato
- Più precisamente c'è un item  $[A \rightarrow \alpha \bullet, a]$  che indica una riduzione per  $a$  e anche un item  $[B \rightarrow \beta \bullet a \gamma, b]$  che invece indica uno shift per  $a$
- Se questo è vero allora almeno un insieme di item (non accorpato) aveva nel core l'item  $[A \rightarrow \alpha \bullet, a]$  e, visto che il core deve essere uguale per tutti gli stati che sono stati accorpati, anche un item  $[B \rightarrow \beta \bullet a \gamma, c]$  per qualche  $c$



# Conseguenze

---

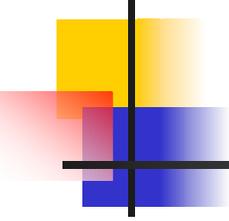
- Ma questo significherebbe che c'era lo stesso conflitto anche sull'insieme di item LR(1) di partenza
- Ciò non è possibile perché siamo partiti dall'ipotesi che la grammatica fosse LR(1)
- Quindi l'accorpamento non produrrà mai conflitti shift/reduce

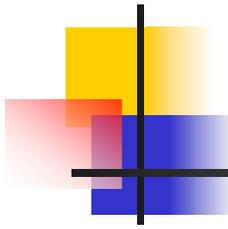


# Conseguenze

---

- Tuttavia è possibile che l'accorpamento provochi conflitti reduce/reduce
- Esempio:
- $S' \rightarrow S$
- $S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
- $A \rightarrow c$
- $B \rightarrow c$

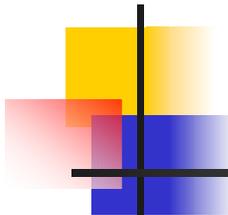
- 
- 
- Questa grammatica genera le quattro stringhe  $acd$ ,  $ace$ ,  $bcd$ ,  $bce$
  - La grammatica è LR(1)
  - Costruendo gli insiemi di item LR(1) si trova
    - $\{[A \rightarrow c\bullet, d], [B \rightarrow c\bullet, e]\}$  i cui item sono validi per il viable prefix  $ac$
    - $\{[A \rightarrow c\bullet, e], [B \rightarrow c\bullet, d]\}$  i cui item sono validi per il viable prefix  $bc$
  - Nessun conflitto



# Esempio

---

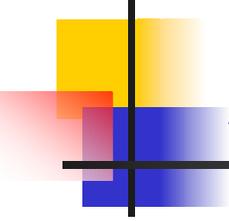
- Proviamo ad accorpare gli stati e vediamo se la grammatica è LALR
- $\{[A \rightarrow c\bullet, d/e], [B \rightarrow c\bullet, d/e]\}$
- Conflitto reduce/reduce: sugli input d ed viene indicato di ridurre sia con  $A \rightarrow c$  che con  $B \rightarrow c$
- Quindi la grammatica è LR(1), ma non è LALR



# Costruzione tabella LALR

---

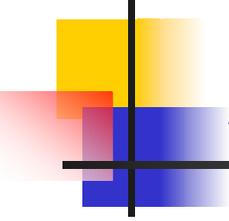
- Esistono due modi
- Il primo, che faremo, è basato sulla costruzione preliminare degli item LR(1) e successivo accorpamento
- È il metodo più semplice, ma anche il più costoso
- Il secondo, che non faremo, genera direttamente gli stati del parser LALR senza passare per gli stati del parser LR



# Algoritmo

---

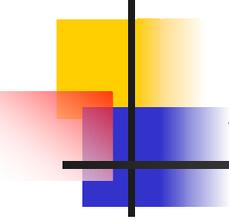
- Input: una grammatica aumentata  $G'$
  - Output: le funzioni *action* e *goto* della tabella di parsing LALR
1. Costruisci  $C = \{I_0, \dots, I_n\}$ , la collezione di insiemi di item LR(1)
  2. Per ogni core degli insiemi di item LR(1) accorpa tutti gli insiemi con quel core nello stesso insieme



# Algoritmo

---

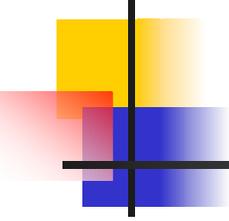
3. Sia  $C' = \{J_0, \dots, J_m\}$  la collezione risultante. I valori della tabella *action* per lo stato  $i$  sono costruiti a partire dall'insieme  $J_i$  utilizzando lo stesso algoritmo visto per la tabella LR canonica. Se ci sono conflitti allora la grammatica non è LALR(1)



# Algoritmo

---

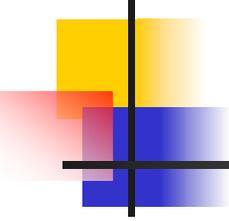
4. La tabella *goto* è costruita come segue:
  - Sia  $J$  l'unione di uno o più insiemi di item LR(1):  $J = I_1 \cup I_2 \cup \dots \cup I_k$
  - Allora i core di  $\text{goto}(I_1, X)$ ,  $\text{goto}(I_2, X), \dots, \text{goto}(I_k, X)$  sono gli stessi, dato che  $I_1, I_2, \dots, I_k$  hanno lo stesso core
  - Sia  $K$  l'unione di insiemi di item che hanno lo stesso core di  $\text{goto}(I_1, X)$
  - Allora  $\text{goto}(J, X) = K$



# Esempio

---

- Riprendiamo la grammatica per  $c^*dc^*d$  che abbiamo usato per illustrare la costruzione della tabella LR canonica
- Abbiamo visto sopra che si possono accorpare gli stati 4 e 7
- Inoltre accorpiamo 3 e 6
- E anche 8 e 9



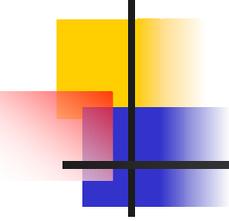
# Esempio

---

- $I_{47} = C \rightarrow d\bullet, c/d/\$$
- $I_{36} = C \rightarrow c\bullet C, c/d/\$$   
 $C \rightarrow \bullet cC, c/d/\$$   
 $C \rightarrow \bullet d, c/d/\$$
- $I_{89} = C \rightarrow cC\bullet, c/d/\$$

# Tabella LALR

STATO		action		goto	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3			
5			r1		
89	r2	r2			



# Considerazioni

---

- Se presentiamo al parser LALR una stringa corretta (nel nostro esempio una stringa di  $c^*dc^*d$ ) il parser LALR esegue esattamente le stesse mosse del parser LR canonico
- Se presentiamo una stringa che non appartiene al linguaggio i due parser hanno un comportamento differente, ma entrambi segnalano l'errore
- Provare con l'input  $ccd\$$