Tabelle LALR

Costruzione delle tabelle di parsing LALR

Maria Rita Di Berardini

Dipartimento di Matematica e Informatica
Università di Camerino
mariarita.diberardini@unicam.it

Metodo LALR

- Introduciamo l'ultimo metodo di costruzione di tabelle di 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
- Le tabelle SLR e LALR per una data grammatica hanno sempre lo stesso numero di stati
- Per un linguaggio tipo il Pascal otteniamo una tabella con alcune centinaia di stati per un
- Per lo stesso linguaggio una tabella LR canonica ha alcune migliaia di stati

Idea

- Consideramo la grammatica utilizzata per illustrare la metodologia LR canonica:

$$S' \to S$$
$$S \to CC$$
$$C \to cC \mid d$$

- Alcuni insiemi di item LR(1) della collezione canonica per questa grammatica soddisfano un'interessante proprietà: stesso insieme di item LR(0) ma diverso simbolo di lookahead (la seconda componente)
- Ad esempio: $I_4 = \{[C \to d\bullet, c/d]\}$ ed $I_7 = \{[C \to d\bullet, \$]\}$
- Osserviamo meglio il comportamento di questi due stati durante l'operazione di parsing
- La grammatica genera stringhe della forma $c^n d c^m d$ con $n, m \geq 0$

Parsing della stringa *ccdcd*

	c	d	\$	S	C
s_0	$S3$	$S4$		1	2
s_1			ACC		
s_2	S6	S7			5
s_3	S3	S4			8
s_4	R3	R3			
s_5			R1		
s_6	S6	S7			9
s_7			R3		
s_8	R2	R2			
s_9			R2		

Stack	Input	Azione
s_0	$ccdcd\$$	shift S3
$s_0\, c\, s_3$	$cdcd\$$	shift S3
$s_0\, c\, s_3\, c\, s_3$	$dcd\$$	shift S4
$s_0\, c\, s_3\, c\, s_3\, d s_4$	$cd\$$	red. $C \to d$
$s_0\, c\, s_3\, c\, s_3\, C s_8$	$cd\$$	red. $C \to cC$
$s_0\, c\, s_3\, C s_8$	$cd\$$	red. $C \to cC$
$s_0\, C s_2$	$cd\$$	shift S6
$s_0\, C s_2\, c\, s_6$	$d\$$	shift S7
$s_0\, C s_2\, c\, s_6\, d\, s_7$	\$	red. $C \to d$
$s_0\, C s_2\, c\, s_6\, C\, s_9$	\$	red. $C \to cC$
$s_0\, C s_2\, C s_5$	\$	red. $S \to CC$
$s_0 S s_1$	\$	acc

Idea

- Durante la lettura di un input $c^n d c^m d$ il parser shifta nello stack il primo gruppo di c ed infine la prima d arrivando nello stato s_4
- A questo punto il parser ordina una riduzione in base alla produzione $C \longrightarrow d$ se il prossimo simbolo input è una c o una d
- Giusto: una stringa della forma $c^m d$, con $m \geq 0$ può cominciare sia con c che con d
- D'altra parte se la prima d è seguita dal \$ il parser deve segnalare un errore: la stringa ccd non appartiene al linguaggio generato dalla grammatica
- Infatti il parser segnala un errore se nello stato s_4 legge un \$

Idea

- Dopo la prima d il parser appila il secondo gruppo di c e quindi la seconda d entrando infine nello stato s_7
- A questo punto il simbolo di input deve essere \$ altrimenti la stringa data non è nel linguaggio (errore)
- IL parser nello stato s_7 ordina una riduzione in base alla produzione $C \longrightarrow d$ se il prossimo simbolo in input è il \$ e un errore se il prossimo simbolo in input è una c o una d

Idea

- Cosa succede se rimpiazziamo gli insiemi di item $I_4 = \{[C \to d\bullet, c/d]\}$ ed $I_7 = \{[C \to d\bullet, \$]\}$ con un unico insieme $I_{47} = \{[C \to d\bullet, c/d/\$]\}$
- Le azioni dello stato corrispondente s_{47} sono "riduci per ogni simbolo in input"
- Dobbiamo modificare anche la goto tenendo conto del fatto che $goto(I_0, d) = I_4$, $goto(I_2, d) = I_7$, $goto(I_3, c) = I_4$ e $goto(I_6, c) = I_7$
- Otteniamo $goto(I_0, d) = goto(I_2, d) = goto(I_3, c) = goto(I_6, c) = I_{47}$
- Il parser così modificato si comporta quasi allo stesso modo di quello originario
- Potrebbe ridurre in alcune circostanze in cui il parser originario avrebbe segnalato un errore, ad esempio ccd o ccdcdc
- L'errore viene rilevato in una fase successiva e comunque prima che venga shiftato il prossimo simbolo terminale sullo stack

Parsing della stringa *ccdcd*

	c	d	\$	S	C
s_0	S3	S47		1	2
s_1			ACC		
s_2	S6	S47			5
s_3	S3	S47			8
s_{47}	R3	R3	R3		
s_5			R1		
s_6	S6	S47			9
s_8	R2	R2			
s_9			R2		

Stack	Input	Azione
s_0	$ccdcd\$$	shift S3
$s_0\, c\, s_3$	$cdcd\$$	shift S3
$s_0\, c\, s_3\, c\, s_3$	$dcd\$$	shift S47
$s_0\, c\, s_3\, c\, s_3\, d s_{47}$	$cd\$$	red. $C \to d$
$s_0\, c\, s_3\, c\, s_3\, C s_8$	$cd\$$	red. $C \to cC$
$s_0\, c\, s_3\, C s_8$	$cd\$$	red. $C \to cC$
$s_0\, C s_2$	$cd\$$	shift S6
$s_0\, C s_2\, c\, s_6$	$d\$$	shift S47
$s_0\, C s_2\, c\, s_6\, d\, s_{47}$	\$	red. $C \to d$
$s_0\, C s_2\, c\, s_6\, C\, s_9$	\$	red. $C \to cC$
$s_0\, C s_2\, C s_5$	\$	red. $S \to CC$
$s_0 S s_1$	\$	acc

Parsing della stringa *ccd*

	c	d	\$	S	C
s_0	S3	S47		1	2
s_1			ACC		
s_2	S6	S47			5
s_3	S3	S47			8
s_{47}	R3	R3	R3		
s_5			R1		
s_6	S6	S47			9
s_8	R2	R2			
s_9			R2		

Stack	Input	Azione
s_0	$ccd\$$	shift S3
$s_0 c\, s_3$	$cd\$$	shift S3
$s_0 c\, s_3 c\, s_3$	$d\$$	shift S47
$s_0 c\, s_3 c\, s_3 d s_{47}$	\$	red. $C \longrightarrow d$
$s_0 c\, s_3\, c\, s_3 C s_8$	\$	errore

In generale

- Cerchiamo insiem di item con core comune
- Il core di un insieme di item LR(1) è determinato dall'insieme delle loro prime componenti; in generale è un insieme di item LR(0)
- E li fondiamo in un unico insieme
- Nell'esempio precedente gli insiemi I_4 ed I_4, con core $C \to d\bullet$, vengono fusi in un unico item I_{47}
- $I_3 = \{[C \to c \bullet C, c/d], [C \to \bullet cC, c/d], [C \to \bullet d, c/d]\}$ ed $I_6 = \{[C \to c \bullet C, \$], [C \to \bullet cC, \$], [C \to \bullet d, \$]\}$ vengono fusi in $I_{36} = \{[C \to c \bullet C, c/d/\$], [C \to \bullet cC, c/d/\$], [C \to \bullet d, c/d/\$]\}$

- $I_8 = \{[C \to cC\bullet, c/d]\}$ ed $I_9 = \{[C \to cC\bullet, \$]\}$ vengono fusi in $I_{89} = \{[C \to cC\bullet, c/d/\$]\}$

La *goto* di insiemi accorpati

- La goto di insiemi accorpati restituisce insiemi accorpati; esempio:

$$\begin{array}{l} goto(I_3, C) = I_8 \\ goto(I_6, C) = I_9 \end{array} \implies goto(I_{36}, C) = I_{89}$$

$$\begin{array}{l} goto(I_3, c) = I_3 \\ goto(I_6, c) = I_6 \end{array} \implies goto(I_{36}, c) = I_{36}$$

$$\begin{array}{l} goto(I_3, d) = I_4 \\ goto(I_6, d) = I_7 \end{array} \implies goto(I_{36}, d) = I_{47}$$

- Questo dipende essenzialmente dal fatto che il core di $goto(I, X)$ è determinato dal core di I
- L'operazione di fusione degli insiemi di item con core comune comporta la creazione di una collezione canonica diversa e quindi la costruzione di una tabella diversa

Conseguenze

- Supponiamo di avere una grammatica LR(1)
- La sua collezione canonica di item LR(1) non produce conflitti
- Se accorpiamo gli insiemi di item con core comune ci potremmo aspettare che i nuovi item producano dei conflitti
- In realtà non è così almeno per quanto rigurda i conflitti shift-reduce

Conseguenze

- Supponiamo di avere un coflitto shift-reduce in uno stato accorpato I_{jk}
- La presenza di questo conflitto ci dice che I_{jk} deve contenere due item della forma:

$$\begin{array}{ll} [A \to \alpha\bullet, a] & action[s_{jk}, a] = \text{ red. } A \to \alpha \\ [A \to \beta \bullet a\gamma, b] & action[s_{jk}, a] = \text{ shift} \end{array}$$

- Se $[A \to \alpha\bullet, a] \in I_{jk}$ allora $[A \to \alpha\bullet, a]$ deve appartenere ad uno degli insiemi originari, diciamo I_j
- Dato che I_{jk} è stato ottenuto fondendo insiemi con core comune I_j deve anche contenere un item della forma $[A \to \beta \bullet a\gamma, c]$ (prima componente uguale, la seconda può anche essere diversa)
- Ma se i entrambi gli item $[A \to \alpha\bullet, a]$ e $[A \to \beta \bullet a\gamma, c]$ sono in I_j, allora lo stesso conflitto c'è anche sull'insieme di item LR(1) di partenza
- Il che contraddice l'ipotesi di partenza

Conflitti reduce/reduce

- È possibile che l'accorpamento provochi conflitti reduce/reduce; consideramo la seguente grammatica LR(1) :

$$
\begin{aligned}
S' &\to S \\
S &\to aAd \mid bBd \mid aBe \mid bAe \\
A &\to c \\
B &\to c
\end{aligned}
$$

  genera le stringhe acd, bcd, ace, bce
- Costruendo la collezione canonica LR(1) identifichiamo gli insiemi di item $\{[A \to c\bullet, d], [B \to c\bullet, e]\}$ e $\{[A \to c\bullet, e], [B \to c\bullet, d]\}$
- Che accorpati producono: $\{[A \to c\bullet, d/e], [B \to c\bullet, d/e]\}$
- Conflitto reduce/reduce: sui simboli in input d ed e viene indicato di ridurre sia con $A \to c$ che con $B \to c$
- La grammatica è LR ma non LALR

Costruzione della tabella LALR

- Esistono due metodi
- Il primo è basato sulla costruzione preliminare della collezione canonica LR(1)
- È il più semplice ma anche il più costoso
- Il secondo genera direttamente gli stati del parser LALR senza passare per gli stati del parser LR
- Vediamo solo la prima metodologia che si basa sull'accorpamento di insiemi di item LR(1) con core comune

Algoritmo

- Input: una grammatica G' aumentata
- Output: la tabella di parsing LALR

1. Costruisci la collezione canonica di item LR(1) $C = \{I_0, I_1, \ldots, I_n\}$
2. Identifica e accorpa tutti gli insiemi di item con core comune. Il risultato di questa operazione è una nuova collezione $C' = \{J_0, J_1, \ldots, J_m\}$
3. I valori della tabella nella parte action per lo stato s_k sono costruiti a partire dall'insieme J_k utilizzando lo stesso algoritmo visto per la costruzione della tabella LR canonica. Se ci sono conflitti allora la grammatica non è LALR(1)

Algoritmo

4. La parte goto della tabella è costruita come segue:
   - sia J_k l'unione di insiemi di item LR(1): $J_k = I_1 \cup I_2 \cup \ldots \cup I_j$
   - dal fatto che $I_1, I_2 \ldots I_j$ sono stati accorpati in un unico insieme possiamo dedurre che $I_1, I_2 \ldots I_j$ hanno lo stesso core
   - e, quindi, anche $goto(I_1, X), goto(I_2, X), \ldots goto(I_j, X)$ hanno lo stesso core
   - sia K il core degli insiemi $goto(I_1, X), goto(I_2, X), \ldots goto(I_j, X)$
   - poniamo $goto(J, X) = K$

Idea

- Riprendiamo la grammatica

$$S' \to S$$
$$S \to CC$$
$$C \to cC \mid d$$

- Abbiamo accorpato gli insiemi I_4 e I_7, I_3 e I_6 ed I_8 e I_9 ottenendo:
  $I_{47} = \{[C \to d\bullet, c/d/\$]\}$
  $I_{36} = \{[C \to c \bullet C, c/d/\$], [C \to \bullet cC, c/d/\$], [C \to \bullet d, c/d/\$]\}$
  $I_{89} = \{[C \to cC\bullet, c/d/\$]\}$
  Inoltre:

$$goto(I_{36}, C) = I_{89}$$
$$goto(I_{36}, c) = I_{36}$$
$$goto(I_{36}, d) = I_{47}$$

La tabella di parsing

	c	d	\$	S	C
s_0	$S3$	$S4$		1	2
s_1			ACC		
s_2	S6	S7			5
s_3	S3	S4			8
s_4	R3	R3			
s_5			R1		
s_6	S6	S7			9
s_7			R3		
s_8	R2	R2			
s_9			R2		

	c	d	\$	S	C
s_0	S36	S47		1	2
s_1			ACC		
s_2	S36	S47			5
s_{36}	S36	S47			98
s_{47}	R3	R3	R3		
s_5			R1		
s_{89}	R2	R2	R2		

Considerazioni finali

- Se presentiamo al parser LALR una stringa sintatticamnte corretta (nel nostro esempio una stringa di ccdcd) esso 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
- Vedere l'esempio del parsing della stringa *ccd* qualche slide fa