Dispensa 3

3.1 Tecnica LR canonica

Il punto debole della tecnica SLR illustrata in Dispensa 2 sta nelle riduzioni, che, ricordiamo, vengono inserite nella parsing table in accordo alla seguente regola:

"Se in uno stato c'è l'LR(0) item $A \rightarrow \alpha$. (ossia un item col punto alla fine) è invocata una riduzione per ogni terminale *a* in Follow(A)"

Questa regola è corretta ed anche facilmente intuibile, infatti se il punto sta alla fine di una produzione vuol dire che il parser ne ha già esaminato tutto il contesto (cioè A), ed è quindi giusto sia ridurre sia che il parser si aspetti come prossimo simbolo proprio uno di quelli in Follow(A). Tuttavia può capitare che tra questi terminali in Follow(A) ce ne sia qualcuno per cui la riduzione è inutile, e quindi inserendone l'azione di reduce nella parsing table si inserisce un'entrata in più inutile e che può aumentare la probabilità di conflitti. Per capire come questa problematica si può presentare si consideri come esempio la seguente coppia di produzioni context free:

$$S \rightarrow A\ b\ A\ c$$
$$A \rightarrow e$$

dove per il non-terminale A vale Follow(A) = {b, c} come si può facilmente ricavare dal lato destro della prima produzione.
Se costruiamo la collezione di insiemi di LR(0) items per questa semplice grammatica si ha:

I_0 = { closure(S' → . S) } = {S' → . S
S → . A b A c
A → . e }
I_1 = goto(I_0, S) = { S' → S . }
I_2 = goto(I_0, A) = { S → A . b A c }
I_3 = goto(I_0, e) = goto(I_4, e) = { A → e . }
I_4 = goto(I_2, b) = { S → A b . A c
A → . e }
I_5 = goto(I_4, A) = { S → A b A . c }
I_6 = goto(I_5, c) = { S → A b A c . }

Ora, se restringiamo la nostra attenzione allo stato I_3 = { A → e . }, poiché in esso c'è un item col punto alla fine allora la tecnica SLR farà ridurre sia sotto *b* che sotto *c* dato che entrambi i terminali sono in Follow(A). Si noti che il parser quando si viene a trovare nello stato I_3 ci può essere giunto (come si può osservare dalla collezione di items o dal corrispondente goto grafo) o provenendo da I_0 oppure da I_4. Ora se il parser proviene da I_0 allora si dovrebbe ridurre solo sotto *b*, infatti in I_0 c'è l'item S → . A b A c e quindi il look-ahead dell'attuale contesto di A è solo *b*; viceversa se il parser proviene da I_4 allora si dovrebbe ridurre solo sotto *c*, infatti in I_4 c'è l'item S → A b . A c e quindi il look-ahead dell'attuale contesto di A è solo *c*. Il problema è che la tecnica SLR non tiene traccia del contesto, sostanzialmente è "cieca" e quindi nell'esempio essa non discrimina fra i due contesti di A diversi e non distingue quindi i due casi diversi illustrati del percorso di provenienza del parser.

La novità della tecnica LR canonica sta proprio nel fatto che essa "spacca" gli stati per tenere una traccia precisa delle riduzioni, e grazie a questo risulta più potente. Ciò è possibile grazie a una nuova definizione di item: infatti mentre la tecnica SLR poggia sulla nozione di LR(0) item, la tecnica LR canonica si basa sul concetto esteso di LR(1) item così definito:

Definizione (LR(1) item):
Un LR(1) item è una coppia formata da una produzione con un punto da qualche parte nel lato destro (cioè un LR(0) item) e da un simbolo chiamato look-ahead che può essere un terminale o l'end-marker $

Formato generico di LR(1) item: $(A \rightarrow \alpha \, . \, \beta \;\; , a)$

E' importante sottolineare che quando il punto sta all'inizio o all'interno della produzione, allora il simbolo di look-ahead (la seconda componente *a*) non contribuisce e quindi in questo caso le nozioni di LR(1) item e LR(0) item diventano equivalenti; se invece il punto sta alla fine, cioè l'LR(1) item è del tipo $(A \rightarrow \gamma \, . \, , a)$ allora in questo caso verrà invocata la riduzione solo su *a* e non su tutti i terminali in Follow(A).
E' questa l'essenza della nuova tecnica LR canonica la quale nell'esempio precedente avrebbe spaccato lo stato $I_3 = \{ A \rightarrow e \, . \, \}$ in due diversi stati $\{ A \rightarrow e \, . \, , b \}$ e $\{ A \rightarrow e \, . \, , c \}$ per mantenere traccia dei contesti, evitando di inserire azioni di riduzione inutili nella parsing table col rischio di aumentare la probabilità di conflitti. Ne consegue, che la tecnica LR canonica aumenta il numero di stati rispetto alla SLR (parsing table più grandi) ma è più precisa e genera meno conflitti (maggior potere riconoscitivo).

Per poter lavorare su LR(1) items le funzioni *closure* e *goto* vanno leggermente modificate e così ridefinite nel contesto LR canonico:

Funzione *closure*

La funzione *closure* si applica a un insieme di LR(1) items e produce ancora un insieme di LR(1) items. In termini operativi essa è composta dalle seguenti due regole:

closure(I)
1) ogni LR(1) item di I è inserito in closure(I)
2) per ogni LR(1) item $(A \rightarrow \alpha \, . \, B \, \beta \, , a)$ in closure(I)
   e per ogni produzione $B \rightarrow \gamma$ in G
   aggiungi $(B \rightarrow . \, \gamma \;\; , b)$ in closure(I) con *b* in First(βa)

Funzione *goto*

La funzione *goto* si applica a un insieme di LR(1) items e ad un simbolo grammaticale, e produce un insieme di LR(1) items. In termini dichiarativi essa è così definita:

goto(I, x) è la closure dell'insieme di tutti gli LR(1) items $(A \rightarrow \alpha \;\; x \, . \, \beta \, , a)$ tali che $(A \rightarrow \alpha \, . \, x \, \beta \, , a)$ è in I

Per concludere, diamo solo qualche cenno all'*algoritmo di costruzione di parsing table LR canonica* (ne omettiamo la stesura completa formale):

G context free $\rightarrow$ Parsing Table LR canonica

Le linee generali dell'algoritmo sono simili a quelle del corrispondente algoritmo SLR, applicate ovviamente a LR(1) items. A partire dalla grammatica aumentata della grammatica context free sorgente il passo centrale per la costruzione della sua parsing table LR canonica sta nella costruzione della sua collezione di insiemi di LR(1) items (o equivalentemente nella costruzione del goto grafo LR canonico se ne si considera la rappresentazione grafica) applicando iterativamente la nuova funzione goto (e le successive closure) che stavolta verrà innescata sull'insieme di partenza I_0 = { closure(S' → . S, $) }. Gli insiemi ottenuti I_0 I_1 … I_n corrispondono agli stati 0, 1, … n della parsing table LR canonica, dove lo stato iniziale 0 corrisponde a I_0 = { closure(S' → . S, $) }. I passi che riempiono le entrate della parsing table con le azioni di "shift", "ACCETTA" ed "ERRORE" sono equivalenti a quelli dell'algoritmo SLR (sempre ovviamente applicati a LR(1) items). La differenza cruciale sta, come detto, nel passo che inserisce le entrate di "reduce":
stavolta per ogni insieme I_i che ha al suo interno un LR(1) item col punto alla fine ($A \rightarrow \alpha$. , *a*) si inserisce l'azione $reduce_{A \rightarrow \alpha}$ nell' entrata ACTION[i, *a*] della parsing table non sotto tutti i terminali appartenenti a Follow(A), ma soltanto sotto il simbolo *a* che sta nella parte look-ahead di quell'item.

Come al solito, se dopo aver inserito le varie azioni la parsing table contiene almeno un conflitto, ossia un' entrata multipla del tipo shift/reduce oppure reduce/reduce nella parte action, allora il parsing fallisce e la grammatica input G è detta essere non LR canonica.

Nota: Le nozioni di LR(0) e LR(1) items usate nelle tecniche SLR e LR canonica, rispettivamente, sono in realtà casi particolare del concetto generale di *LR(k) item*, così definito:

Definizione (LR(k) item):
Un LR(k) item è una coppia formata da una produzione con un punto da qualche parte nel lato destro e da una stringa di k simboli terminali, di cui l'ultimo può anche essere l'end-marker $

Nel caso k=0, la seconda componente (cioè la stringa) manca del tutto, e ci si riduce quindi alla nozione di LR(0) item. Nel caso k=1, la stringa si riduce ad un unico simbolo (terminale o $) determinando quindi la nozione di LR(1) item.

3.2 Tecnica LALR

La tecnica LALR è un buon compromesso in potere riconoscitivo e taglia della parsing table rispetto alle tecniche SLR e LR canonica. Analogamente al metodo LR canonico, la tecnica LALR usa i look-ahead negli items (ossia usa LR(1) items) per decidere le riduzioni, ed è quindi più precisa del metodo SLR, tuttavia riesce anche a "spaccare" un po' di meno della tecnica LR canonica accorpando gli stati e riducendo quindi il numero di stati della parsing table finale.
Il concetto fondamentale su cui poggia la tecnica LALR e che permette questo accorpamento di stati è quello di *core*, che è così definito:

Definizione (core):
Dato un insieme di LR(1) items, il core è l'insieme formato dalle sue prime componenti

Esempio:

I = { (S → C . C , $)
(C → . c C , $)
(C → . d , $)}

core(I) = { $S \rightarrow C \,.\, C$
$C \rightarrow \,.\, c\, C$
$C \rightarrow \,.\, d$ }

In altre parole, il core di un insieme di LR(1) items è ottenuto considerandone solo gli LR(0) items che lo compongono senza i look-ahead.

Una volta introdotta la nozione di core, possiamo delineare la filosofia generale del metodo LALR (G context free $\rightarrow$ Parsing Table LALR) che è la seguente:
" Data la grammatica aumentata di una grammatica context free sorgente G si calcolano gli insiemi di LR(1) items $I_0\ I_1\ \ldots\ I_n$ (che costituirebbero gli stati di un parser LR canonico) e si fondono tra loro gli insiemi con core comune. In questo modo, gli insiemi fusi diventano un unico insieme riducendo quindi la taglia. Gli insiemi di LR(1) items $J_0\ I_1\ \ldots\ J_m$ accorpati dopo le fusioni sono gli stati del parser LALR; a partire da questa collezione accorpata di LR(1) items si possono applicare i canonici passi per riempire le entrate della parsing table con le azioni di "shift", "reduce", "ACCETTA" ed "ERRORE" ottenendo la parsing table LALR per G "

Nota: Data una grammatica context free G si ha sempre che:
- numero stati parsing table LALR di G $\leq$ numero stati parsing table LR canonica di G
- numero stati parsing table LALR di G = numero stati parsing table SLR di G

Concludiamo lo studio della tecnica LALR con un importante risultato contenuto nel seguente teorema:

Teorema: Data una collezione di insiemi di LR(1) items senza conflitti le fusioni LALR non possono produrre conflitti shift-reduce

Dimostrazione: Supponiamo per assurdo che una volta applicate le fusioni LALR alla collezione di insiemi di LR(1) items di partenza si ottiene uno stato I (per esempio prodotto dall'accorpamento di due stati J e K) contenente fra gli altri i due items ($A \rightarrow \alpha \,.\,,\ a$) e ($B \rightarrow \beta \,.\, a\,\gamma\ ,\ b$) e quindi un conflitto shift-reduce sul simbolo *a*. Poichè I è il prodotto di una fusione LALR ciò vuol dire che, ad esempio, lo stato J che ha contribuito all'accorpamento doveva contenere l'item ($A \rightarrow \alpha \,.\,,\ a$) e, essendo i core di I, J e K uguali, anche un item ($B \rightarrow \beta \,.\, a\,\gamma\ ,\ c$) per qualche *c*. In questo modo lo stato J presenterebbe anch'esso un conflitto shift-reduce sul simbolo *a*, e ciò non è possibile essendo per ipotesi la collezione di partenza priva di conflitti.

3.3 Gestione di grammatiche ambigue

Al di fuori della classe di grammatiche della gerarchia LR ci sono le grammatiche ambigue (costrutti di linguaggi ambigui hanno più parse tree e quindi il loro riconoscimento non può essere deterministico) infatti valgono i seguenti risultati:
- se G è LR allora G è non ambigua
- se G è ambigua allora G non è LR

In presenza di una grammatica ambigua si può pensare a due soluzioni: cercare di trovare una grammatica equivalente non ambigua oppure usare la stessa grammatica ambigua ed imporre delle regole disambiguanti sui conflitti della parsing table

Esempio Si consideri la seguente semplice grammatica G1 per espressioni aritmetiche:

E → E + E
E → E * E
E → (E)
E→ id

G1 è ambigua e quindi non LR, e può essere trasformata nella seguente grammatica equivalente G2 che, forzando le precedenze e le associatività, diventa non ambigua ed è LR:

E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Tuttavia si può decidere di usare G1 in quanto molto generale e naturale imponendo le precedenze e le associatività con delle regole sui conflitti della parsing table e senza predefinirle nella grammatica come fa G2.
Ad esempio, costruendo la collezione di insiemi di LR(0) items per G1 si consideri lo stato I7 contenente gli items:

E → E + E .
E → E . + E
E → E . * E

Poichè i terminali + e * sono nella Follow(E), nella riga corrispondente della parsing table avremo un conflitto s/r sia su + che su *
Consideriamo per esempio il conflitto su *
Analizziamo l'input **id + id * id**. Lo stato I7 è relativo a un momento di parsing in cui l'input è stato esaminato fino a **id + id**, il prossimo simbolo input è * e nello stack c'è la stringa E + E più gli stati associati con lo stato 7 al top.

CASO 1:
Se si vuole imporre che * abbia precedenza su + allora l'azione dovrebbe essere quella di shiftare * sullo stack per preparare la riduzione E * E. Quindi in questo caso la regola disambiguante risolve il conflitto a favore dello shift.
CASO 2:
Se si vuole imporre che + abbia precedenza su * allora il parser dovrebbe ridurre E + E ad E. Stavolta la regola disambiguante risolve il conflitto a favore del reduce.

Con un ragionamento analogo si può risolvere il conflitto su + analizzando l'input **id + id + id** e imponendo una regola disambiguante sull'associatività sinistra o destra dell'addizione.