9. Conclusioni sui parser

LR(k) vs LL(k): Trovare il passo successivo in una derivazione

Nel parsing LR(k), ogni riduzione è rilevabile grazie al contesto completo a sinistra, alla frase riducibile stessa e ai k simboli terminali a destra. Questo consente al parser di prendere decisioni basate su una visione più ampia del contesto.

Nel parsing LL(k), il parser deve selezionare l'espansione basandosi sul contesto completo a sinistra e sui prossimi k simboli terminali. Di conseguenza, il parsing LR(k) esamina un contesto più ampio rispetto al parsing LL(k), permettendo una maggiore flessibilità nella rilevazione delle riduzioni.

Ricorsione Sinistra vs Ricorsione Destra

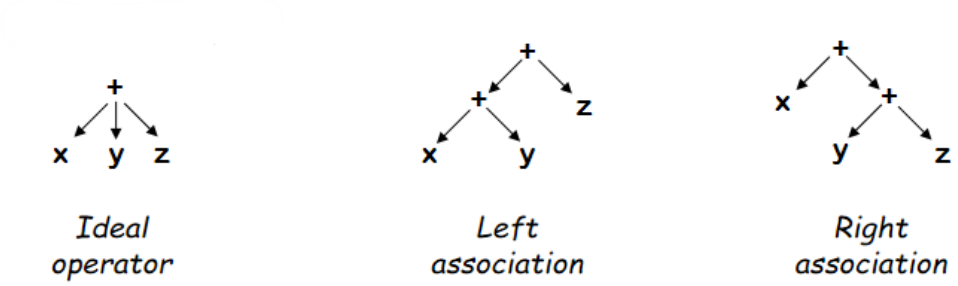
La ricorsione a sinistra e la ricorsione a destra presentano caratteristiche diverse a seconda del tipo di parser utilizzato. La ricorsione a destra è necessaria per la terminazione nei parser top-down, ma tende ad utilizzare più spazio nello stack. Inoltre, una ricorsione a destra ingenua produce un'associatività a destra.

D'altra parte, la ricorsione a sinistra funziona bene nei parser bottom-up e richiede meno spazio nello stack. Una ricorsione a sinistra ingenua porta all'associatività a sinistra.

Come regola generale, la ricorsione a sinistra è più adatta ai parser bottom-up, mentre la ricorsione a destra è preferibile per i parser top-down.

Associatività

L'associatività può avere un impatto significativo, specialmente in operazioni come l'aritmetica in virgola mobile e nelle opportunità di ottimizzazione. Ad esempio, nelle espressioni come $x + y + z$, l'ordine in cui vengono eseguite le operazioni può influenzare il risultato finale.



Se $y + z$ si ripete altrove, o se $x + y$ o $x + z$ si trovano in altre parti del codice, il compilatore potrebbe voler cambiare la "forma" delle espressioni per massimizzare l'efficienza. Inoltre, consideriamo un caso in cui $x = 2$ e $z = 17$. Né l'associatività a sinistra né quella a destra mettono in evidenza il valore 19, che potrebbe risultare utile per ottimizzare il calcolo.

In definitiva, la "forma" migliore di un'espressione dipende dal contesto circostante e dalle opportunità di ottimizzazione che il compilatore può sfruttare

Rilevamento degli errori

Nel parsing con *recursive descent*, il parser utilizza l'ultima clausola *else* in una routine, il che permette al programmatore del compilatore di codificare quasi qualsiasi azione arbitraria per gestire l'errore.

Nel parsing *table-driven LL(1)*, quando il parser si trova nello stato *si* e incontra la parola *x*, se l'entrata corrispondente nella tabella è un errore, viene segnalato. Le voci valide nella riga per *si* rappresentano le possibilità accettabili per proseguire.

Nel parsing *table-driven LR(1)*, la situazione è simile. Se il parser è nello stato *si* e incontra la parola *x* con un'entrata che rappresenta un errore, viene segnalato. Tuttavia, gli stati di shift nella riga corrispondente codificano le possibili azioni da intraprendere. Inoltre, con LR(1) è possibile precomputare messaggi di errore più dettagliati utilizzando gli elementi di LR(1) per fornire feedback migliori.

Recupero dagli errori

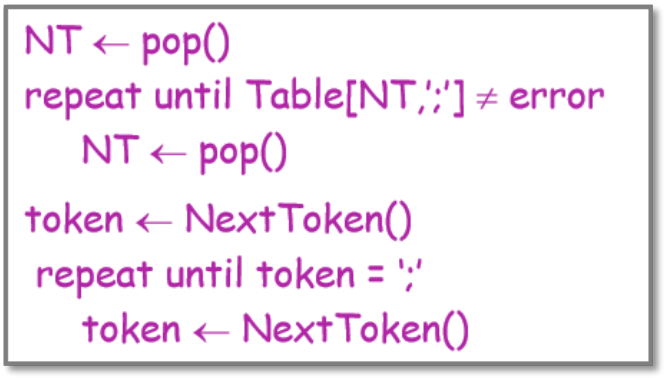
Nel parsing *table-driven LL(1)*, il recupero dagli errori può essere gestito in due modi principali. Se viene trattato come un token mancante, ad esempio una parentesi chiusa ‘)’, il parser può espandere la produzione inserendo il simbolo desiderato. Se invece viene rilevato un token extra, come nel caso di un'espressione errata come ‘x - + y’, il parser può rimuovere l'elemento dallo stack e avanzare oltre l'errore.

Nel parsing *table-driven LR(1)*, il recupero dagli errori segue una logica simile ma con differenze operative. Se il token viene considerato mancante, come una parentesi chiusa ‘)’,

il parser effettuerà uno shift sul token. Se invece si tratta di un token extra, come in ‘x - + y’, il parser non eseguirà lo shift, mantenendo il token attuale per ripristinare lo stato corretto.

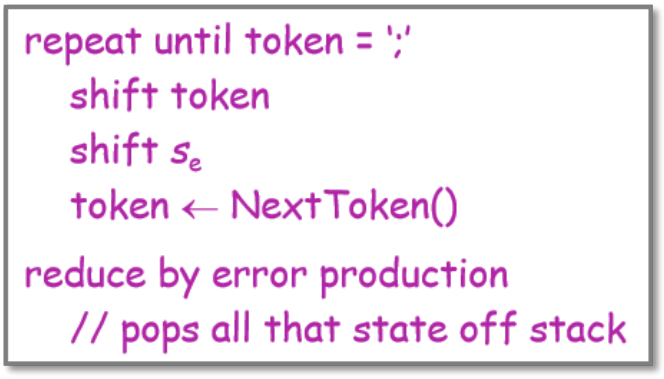
Una strategia comune per il recupero dagli errori è il recupero tramite *“hard token”*. Questo approccio consiste nel saltare avanti nell'input fino a trovare un token "duro", ad esempio un punto e virgola ‘;’, che separa le dichiarazioni e crea una pausa logica nel processo di parsing. Una volta trovato il token, il parser si risincronizza con lo stato, lo stack e l'input per riprendere dopo il token.

Nel parsing *LL(1)*, il recupero avviene svuotando lo stack fino a trovare una riga nella tabella che contenga un'entrata per il token ‘;’.



Resynchronizing an LL(1) parser

Nel parsing *LR(1)*, si svuota lo stack fino a trovare uno stato che consenta una riduzione sul token ‘;’. Questo metodo non corregge l'input, ma permette al parser di continuare l'analisi senza interrompere completamente il processo.



Resynchronizing an LR(1) parser

Ridurre le tabelle *Action* e *Goto*

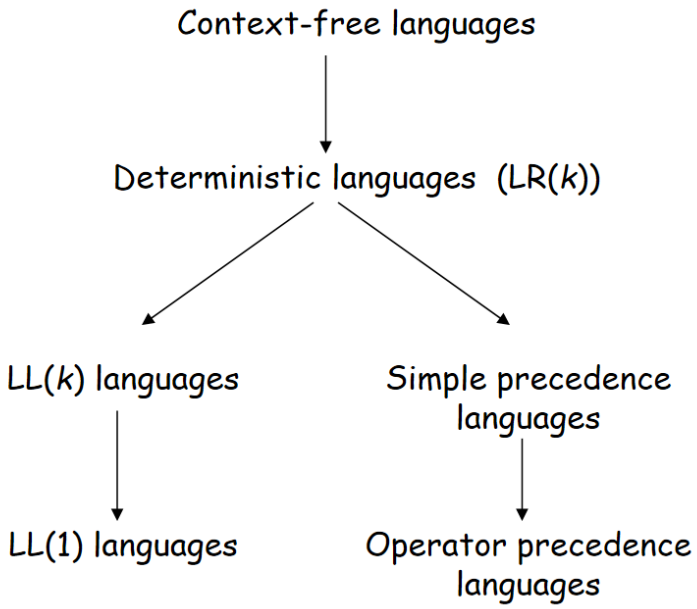
La riduzione delle tabelle *Action* e *Goto* può essere fatta in tre modi principali:

- Il primo metodo è combinare terminali simili, come numeri e identificatori, o operatori come + e -, * e /. Questo approccio rimuove direttamente una colonna e, in alcuni casi,

può anche eliminare una riga.

- Il secondo metodo consiste nel combinare righe o colonne identiche. Si implementa una sola volta la riga o la colonna e si rimappano gli stati, ma ciò richiede un livello di indirezione extra per ogni ricerca. Questo approccio può includere mappe separate per le tabelle *Action* e *Goto*.
- Il terzo metodo è l'uso di un altro algoritmo di costruzione. Sia *LALR(1)* che *SLR(1)* producono tabelle più piccole. *LALR(1)* rappresenta ciascuno stato con i suoi elementi "core", mentre *SLR(1)* utilizza gli elementi *LR(0)* e il set *Follow*. Entrambe le implementazioni sono ampiamente disponibili e offrono un modo efficiente per ridurre la dimensione delle tabelle.

Gerarchia delle grammatiche libere dal contesto



Sommario

Il *Top-down Recursive Descent, LL(1)* offre una serie di vantaggi, tra cui la velocità, grazie alla sua struttura semplice, e una buona località, che permette una facile comprensione e gestione del codice. Inoltre, è considerato semplice da implementare e ha un buon rilevamento degli errori, permettendo una gestione degli errori immediata e personalizzabile. Tuttavia, questo metodo è spesso codificato a mano, il che lo rende ad alta manutenzione, poiché eventuali cambiamenti nelle regole della grammatica richiedono modifiche al codice sorgente. Un altro svantaggio è che tende a gestire l'associatività a destra, il che può essere limitante per alcune espressioni o linguaggi.

L'LR(1), invece, è altrettanto veloce e supporta una gamma più ampia di linguaggi deterministici. Questo metodo può essere automatizzato, riducendo il carico di manutenzione, ed è in grado di gestire l'associatività a sinistra, spesso preferita in molte grammatiche. Tuttavia, l'LR(1) comporta alcuni svantaggi significativi. Richiede la gestione di grandi insiemi di lavoro, e i messaggi di errore prodotti sono spesso meno chiari o utili rispetto a quelli del LL(1). Inoltre, le tabelle necessarie per il parsing LR(1) possono diventare molto grandi, rendendo complesso e inefficiente il loro utilizzo in alcune applicazioni.