7. Costruire un parser top-down

Quando si ha a disposizione una grammatica LL(1) e si sono determinati gli insiemi $FIRST$

```
for each x ∈ T, FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← Ø

while (FIRST sets are still changing) do
   for each p ∈ P, of the form A→β do
      if β is B1B2…Bk then begin;
         rhs ← FIRST(B1) - {ε}
         for i ← 1 to k-1 by 1 while ε ∈ FIRST(Bi) do
             rhs ← rhs ∪ (FIRST(Bi+1) - { ε })
             end  // for loop
       end        // if-then
       if  i = k and ε ∈ FIRST(Bk )
          then rhs ← rhs ∪ {ε}
        FIRST(A) ← FIRST(A) ∪ rhs
        end  // for loop
    end      // while loop
```

e $FOLLOW$

```
for each A ∈ NT, FOLLOW(A) ← Ø

FOLLOW(S) ← {EOF}

while (FOLLOW sets are still changing)
    for each p ∈ P, of the form A→B1B2 … Bk
        TRAILER ← FOLLOW(A)
        for i ← k down to 1
            if Bi ∈ NT then                          // domain check
                FOLLOW(Bi) ← FOLLOW(Bi) ∪ TRAILER
                if ε ∈ FIRST(Bi )                    // add right context
                  then TRAILER ← TRAILER ∪ ( FIRST(Bi) - {ε})
                  else TRAILER ← FIRST(Bi)  // no ε => no right context
            else TRAILER ← {Bi}                      // Bi ∈ T => only 1 symbol
```

il processo di costruzione di un parser top-down può essere automatizzato seguendo un approccio **recursive-descent** (discesa ricorsiva).

Per ciascun non-terminale della grammatica, si emette una routine che controlla le alternative rhs utilizzando un insieme di istruzioni `if-then-else`. Queste routine hanno il compito di verificare quale produzione applicare. Se la routine riesce a determinare la produzione correttamente, restituisce `true`, altrimenti lancia un errore.

Questo approccio, pur essendo semplice e funzionante, ha alcuni limiti in termini di efficienza. Infatti, l'uso intensivo di strutture `if-then-else` può risultare lento, specialmente quando le alternative sono molteplici.

Per migliorare le prestazioni del parser, una soluzione più efficiente potrebbe essere l'uso di una struttura `case statement`, che consente di gestire le alternative in modo più rapido e ordinato. Ancora meglio, si potrebbe considerare la costruzione di una tabella che codifica le varie opzioni per ogni non-terminale.

In questo modo, si otterrebbe un parser non solo più veloce, ma anche più organizzato, riducendo la complessità del codice e migliorando la sua leggibilità e manutenzione.

Algoritmo per costruire la Tabella LL(1)

Per costruire una tabella LL(1), bisogna codificare la conoscenza della grammatica in una struttura tabellare che permette al parser di eseguire scelte corrette durante il processo di parsing. La tabella LL(1) viene costruita considerando le produzioni della grammatica, gli insiemi FIRST e FOLLOW, e le transizioni tra i non-terminali (NT) e i terminali (T). Ecco i passi principali per costruire la tabella:

Inizializzazione della Tabella. La tabella LL(1) è una matrice bidimensionale in cui:

- Ogni riga corrisponde a un non-terminale (NT).
- Ogni colonna corrisponde a un terminale (T) o al simbolo `EOF`

Popolamento della Tabella. Per ciascuna produzione $A \to \alpha$, dove A è un non-terminale e α è una stringa di terminali e/o non-terminali:

- Per ogni simbolo t nell'insieme FIRST(α), inserisci la produzione $A \to \alpha$ nella cella della tabella corrispondente a (A, t).
- Se $\epsilon \in$ FIRST(α), allora per ogni simbolo b nell'insieme FOLLOW(A), inserisci la produzione $A \to \epsilon$ nella cella corrispondente a (A, b).
- Se $\epsilon \in$ FIRST(α) e $FOLLOW(A)$ contiene il simbolo `EOF`, inserisci la produzione $A \to \epsilon$ nella cella (A, `EOF`).

Gestione dei Conflitti. Se in qualsiasi cella della tabella LL(1) vengono inserite più di una produzione, la grammatica non è LL(1) e il parser non sarà deterministico. In tal caso, la grammatica deve essere modificata per eliminare i conflitti (come l'eliminazione di ricorsione sinistra o fattorizzazione).

Costruzione del parser

Una volta costruita la tabella, il parser utilizza questa struttura per eseguire il parsing. Ecco come si può costruire un parser basato sulla tabella LL(1):

Inizializzazione

- Mantieni uno stack che inizia con il simbolo di partenza della grammatica (NT_{start}) e il simbolo `EOF`.
- Considera il primo simbolo dell'input corrente.

Loop principale del parsing (fase di controllo dello stack)

- Se il simbolo in cima allo stack è un terminale, confrontalo con il simbolo corrente dell'input
- Se i due simboli corrispondono, rimuovi il simbolo dallo stack e passa al successivo simbolo di input
- Se non corrispondono, segnala un errore di parsing
- Se il simbolo in cima allo stack è un non-terminale, consulta la tabella LL(1) usando la coppia (non-terminale, simbolo di input corrente) per determinare la produzione da applicare

Applica la produzione

- Sostituisci il non-terminale in cima allo stack con i simboli della produzione trovata nella tabella (da destra a sinistra)
- Se la produzione è $A \to \epsilon$, semplicemente rimuovi il non-terminale dallo stack

Terminazione

- Il parsing ha successo se lo stack è vuoto e tutti i simboli di input sono stati consumati
- Se lo stack è vuoto ma ci sono ancora simboli di input, oppure se ci sono simboli nello stack ma l'input è terminato, segnala un errore

Considerazioni su come trasformare una grammatica non-LL(1) in una grammatica LL(1)

Spesso non è possibile trasformare una grammatica non-LL(1) in una grammatica LL(1). Tuttavia, in alcuni casi, è possibile effettuare delle trasformazioni che rendano la grammatica LL(1).

Consideriamo una grammatica G con le produzioni $A \to \alpha\beta_1$ e $A \to \alpha\beta_2$. Se α deriva una stringa che non sia ϵ, allora i seguenti insiemi avranno un'intersezione non vuota

$$\text{First}^+(A \to \alpha\beta_1) \cap \text{First}^+(A \to \alpha\beta_2) \neq \emptyset$$

Questo significa che la grammatica non è LL(1). In casi come questo, possiamo provare a risolvere il problema estraendo il prefisso comune α in una produzione separata. La grammatica diventa:

$$A \to \alpha A', \quad A' \to \beta_1 \quad \text{e} \quad A' \to \beta_2$$

Se successivamente si verifica che:

$$\text{First}^+(A' \to \beta_1) \cap \text{First}^+(A' \to \beta_2) = \emptyset$$

allora la grammatica G può diventare LL(1) dopo questa trasformazione.

Fattorizzazione sinistra

Di seguito è mostrato in pseudocodice l'algoritmo di **fattorizzazione sinistra**, che permette di trasformare alcune grammatiche non $LL(1)$ in grammatiche $LL(1)$

Per ogni non terminale A:
Trova il prefisso più lungo α comune a 2 o più alternative per A
Se $\alpha \neq \epsilon$, allora
Sostituisci tutte le produzioni $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ \ldots \ \mid \alpha\beta_n \mid \gamma$
con
$A \to \alpha A' \mid \gamma$
$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ \ldots \ \mid \beta_n$

Ripeti fino a quando nessun non terminale ha alternative con lo stesso prefisso a destra.
