Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

GLL Parsing su linguaggi non lineari

Relatore
Prof. Gennaro Costagliola

Candidato
Mazzotta Fabio

Anno Accademico 2018-2019

Ai miei genitori.
Dedicato a chi ha creduto in me;
e a chi lotta ogni giorno e non si arrende.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Obiettivi

Questa tesi di laurea descrive il funzionamento e l'implementazione del parsing **Generalizzato LL (GLL)** sui linguaggi non lineari. Il parsing GLL è un algoritmo di parsing top down che viene utilizzato per gestire tutte le grammatiche context-free che sono ambigue e ricorsive a sinistra. La caratteristica principale di questo algoritmo è che risulta essere un parser a **discesa ricorsiva** e ciò permette di avere il controllo del flusso sulle strutture della grammatica e risultano semplici da implementare e semplici da testare passo dopo passo attraverso il debugger. Questo parser è stato utilizzato per riconoscere linguaggi non lineari (bidimensionali) generati da grammatiche posizionali, ossia generalizzazioni di grammatiche context-free. La tesi è divisa in tre parti. Nella prima parte si cerca di illustrare come funziona il parsing LL, che rappresenta la base del parsing GLL, e i suoi limiti. Successivamente si discuterà come estendere il parsing LL attraverso il parsing GLL, illustrandone i principi e le strutture dati che utilizza. Ciò viene descritto rispettivamente nel secondo e terzo capitolo. Nella seconda parte si analizzaranno le grammatiche posizionali. Questo argomento sarà trattato nel quarto capitolo. Nell'ultima parte si parlerà l'implementazione del parsing GLL applicato ad una grammatica posizionale. In particolare nel quinto capitolo si descriverà le varie componenti software del parsing GLL, nel sesto capitolo si illustrerà come viene applicato il software del parsing GLL ad una grammatica posizionale e nel settimo capitolo si parlerà del tool utilizzato per testare il software del parsing GLL. Infine nell'ottavo capitolo si discuteranno i risultati ottenuti e gli sviluppi futuri.

Capitolo 2

Parsing LL(1)

2.1 Introduzione

Il parsing, o analisi sintattica, è una fase di compilazione che viene utilizzata per definire la sintassi di un linguaggio di programmazione. In altre parole definisce la forma di un programma corretto. Utilizza i token [1], ossia sequenze di caratteri dotate di significato restituite da un analizzatore lessicale (Lexer); per produrre una rappresentazione intermedia ad albero che rappresenta la struttura grammaticale dei token. Una tipica rappresentazione è l'*albero sintattico*, o *syntax tree* in cui un nodo interno rappresenta un'operazione mentre i figli rappresentano gli argomenti dell'operazione; infine, questo albero prodotto, viene passato alle restanti fasi del processo di compilazione. Chiaramente, ci si aspetta che il parser sia in grado segnalare gli errori delle forme sintattiche sbagliate. In figura 2.1 viene mostrato il funzionamento del parser.

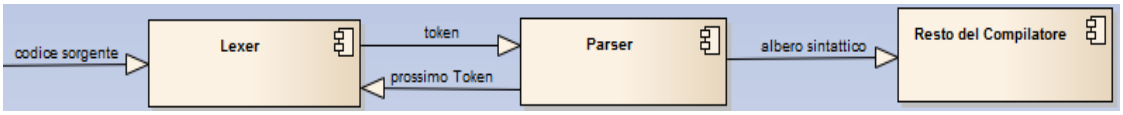

Figura 2.1: *Posizione del parser all'interno del compilatore.*

I metodi di parsing più comunemente utilizzate dai compilatori sono:

- **Parsing top down:** la costruzione dell'albero sintattico avviene partendo dalla radice dell'albero fino ad arrivare alle foglie dell'albero;
- **Parsing bottom up:** la costruzione dell'albero sintattico avviene partendo dalle foglie dell'albero fino ad arrivare alla sua radice.

In questa tesi tratteremo il parsing top down in quanto il GLL parsing usa questa metodologia

2.2 Grammatiche context-free

In questo paragrafo introduciamo una notazione - *la grammatica context-free* - utilizzata per specificare la sintassi dei linguaggi di programmazione. Le grammatiche sono usate per descrivere i costrutti dei linguaggi di programmazione. Ad esempio in C, il while può avere la seguente forma:

while (espressione) statement

Questa notazione indica che il costrutto è composto dalla parola chiave **while**, una parentesi tonda aperta, un'espressione, una parentesi tonda chiusa e uno statement. Usando la variabile *expr* che indica una generica espressione e la variabile *stmt* per indicare lo statement, la regola di questo costrutto può essere definita nel seguente modo:

$$stmt \rightarrow \textbf{while}\ (exp) stmt \tag{2.1}$$

in cui la freccia può essere letta come "può avere la forma". Questa regola prende il nome di **produzione.** All'interno della produzione la parola while, la parentesi aperta e tonda prendono il nome di **terminali**, mentre le variabili expr e stmt prendono il nome di **non terminali**.

2.2.1 Definizione di grammatica

Una grammatica context-free è una quadrupla i cui elementi sono [1]:

1. **Terminali.** I terminali sono simboli di base con cui la grammatica definisce il linguaggio. Il termine "*token*" è un sinonimo di terminale.

2. **Non-Terminali.** I non-terminali sono variabili sintattiche che denotano un insieme di stringhe. Nella produzione 2.1 *stmt* e *expr* sono non-terminali. Gli insiemi di stringhe rappresentati dai non-terminali concorrono a definire il linguaggio generato dalla grammatica.

3. **Simbolo Iniziale.** In una grammatica uno dei non-terminali costituisce il simbolo iniziale e l'insieme di stringhe che esso denota coincide con l'intero linguaggio generato dalla grammatica.

4. **Produzione.** Le produzioni di una grammatica definiscono come i terminali e i non-terminali possono essere combinate a formare stringhe. Ogni produzione è formata da:

(a) un non-terminale chiamato **testa**; la produzione definisce alcune delle stringhe denotate alla sua testa;

(b) il simbolo $\rightarrow$; a volte il simbolo $::=$ è utilizzato al posto della freccia;

(c) un **corpo** o **lato destro** costituito da zero o più non-terminali o terminali; i componenti descrivono un modo in cui le stringhe denotate dal non-terminale della testa possono essere costruite.

2.2.2 Convenzioni notazionali

In questo paragrafo vengono definite le convenzioni notazionali delle grammatiche che verranno usate nel resto della tesi.

1. I seguenti simboli rappresentano i terminali:
   (a) le singole lettere minuscole dell'alfabeto;
   (b) i simboli degli operatori matematici e di punteggiatura;
   (c) le stringhe minuscole in grassetto;
   (d) le cifre numeriche.
2. I seguenti simboli sono non-terminali:
   (a) le singole lettere maiuscole dell'alfabeto;
   (b) se usate per descrivere i singoli costrutti della programmazione, le lettere maiuscole possono indicare i non-terminali del linguaggio.
3. La testa della prima produzione è il simbolo iniziale.
4. Un insieme di produzioni del tipo $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_k$, con una testa comune A (che chiamiamo *A-produzioni*), possono essere scritte nel seguente modo: $A \rightarrow \alpha_1 \mid \alpha_1 \mid \alpha_2 \ldots \alpha_k$. Chiamiamo α_1, α_2, ..., α_k le *alternative per A*.

2.2.3 Derivazioni

Un albero di parsing [1] può essere costruito mediante varie fasi di derivazioni dove, partendo dal simbolo iniziale, ad ogni passo di riscrittura un simbolo non-terminale viene sostituito con il corpo della sua produzione. Tale visione *derivazionale* corrisponde al metodo di costruzione top-down degli alberi di parsing. Facciamo un esempio. Consideriamo la seguente grammatica:

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id} \tag{2.2}$$

La produzione E $\rightarrow$ E + E significa che se E indica un'espressione alla anche E + E è un'espressione. La sostituzione di una singola E con E + E si indica con la seguente notazione:

$$E \Rightarrow E + E \Rightarrow \mathbf{id} + E \Rightarrow \mathbf{id} + \mathbf{id} \tag{2.3}$$

che si legge "E deriva E + E. La produzione $E \rightarrow E + E$ può essere utilizzata per sostituire qualsiasi occorrenza di E con E + E in una qualsiasi stringa di simboli della grammatica. La sequenza 2.3 viene definita come una derivazione della stringa **id** + **id** a partire da E. Questa derivazione dimostra che la stringa **id** + **id** è una particolare istanza di un'espressione. Ora diamo una definizione formale di concetto di derivazione. $\ll$ Consideriamo un non-terminale A posizionata in mezzo ad una sequenza di simboli grammaticali $\alpha A \beta$ dove α e β sono stringhe arbitrarie di simboli grammaticali. Supponiamo che $A \rightarrow \gamma$ sia una produzione. In tal caso possiamo scrivere $\alpha A \beta \Rightarrow \alpha\gamma\beta$, in cui il simbolo $\Rightarrow$ significa "deriva in un solo passo". Quando abbiamo una sequenza di passi di derivazione del tipo $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ in cui possiamo riscrivere α_1 come α_n diremo α_1 *deriva* α_n. Per esprimere che una stringa "deriva in zero o più passi" una nuova stringa utilizziamo il simbolo $\overset{*}{\Rightarrow}$. Quindi,

1. $\alpha \overset{*}{\Rightarrow} \alpha$, per qualsiasi stringa α;
2. se $\alpha \overset{*}{\Rightarrow} \beta$ e $\beta \Rightarrow \gamma$, allora $\alpha \overset{*}{\Rightarrow} \gamma$.

Inoltre il simbolo $\overset{+}{\Rightarrow}$ significa "deriva in uno o più passi.

Se $S \overset{+}{\Rightarrow} \alpha$, dove S è il simbolo iniziale della grammatica G, diciamo che α è una **forma sentenziale** di G. Una forma sentenziale può contenere sia terminali che non terminali e può essere vuota. Una **sentenza** o **frase** di G è una forma sentenziale che non contiene nessun non-terminale. Il **linguaggio generato** da una grammatica G è l'insieme di tutte le sue frasi. Quindi una stringa di terminali w appartiene a *L(G)*, il linguaggio generato da G, se e solo se w è una frase di G, cioè se $S \overset{*}{\Rightarrow} w$. Un linguaggio che può essere generato da una grammatica è detto un **linguaggio libero dal contesto**. Se due grammatiche generano lo stesso linguaggio sono dette **equivalenti**.$\gg$ La stringa **id**+**id** è una frase della grammatica 2.2 poichè esiste la derivazione 2.3. Le sequenze di derivazioni prevedono che ad ogni vengano fatte due scelte: la prima scelta consiste nello scegliere il non-terinale da sostituire; la seconda scelta consiste nello scegliere una delle produzioni in cui il non-terminale scelto risulta essere la testa della produzione. Infatti nella derivazione 2.3 ogni non-terminale è sostituito con il corpo della produzione corrispondente. Ogni non-terminale da sostituire viene selezionato in questo modo:

1. nelle *derivazioni sinistre* si sceglie sempre il non-terminale più a sinistra. La derivazione 2.3 è una derivazione a sinistra.

2. nelle *derivazioni destre* si sceglie sempre il non-terminale più a destra.

2.2.4 Alberi di parsing

Un **albero di parsing** è [1] una rappresentazione grafica di una derivazione che non dipende dall'ordine in cui le produzioni sono utilizzate per rimpiazzare i non-terminali. Ogni nodo interno rappresenta l'applicazione di una produzione ed è etichettato con il non-terminale che indica la testa della produzione. I figli di questo nodo sono etichettati con i simboli che appaiono nel corpo della produzione utilizzata per sostituire il non-terminale. Un esempio di albero di parsing relativo alla stringa **id** + **id** è mostrato nella figura 2.2. Le foglie dell'albero di parsing sono etichettate con terminali o

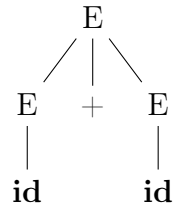

Figura 2.2: *Albero di parsing relativo alla stringa* **id** + **id**

non-terminali che, letti da sinistra verso destra formano una forma sentenziale chiamata **frontiera** dell'albero. Ora tramite un esempio mostreremo come viene costruito un albero sintattico.

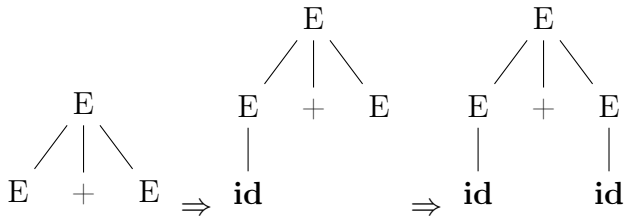

Figura 2.3: *Sequenza di alberi di parsing relativi alla derivazione* 2.3

In figura 2.3 viene rappresentata la sequenza di alberi sintattici costruiti dalla derivazione 2.3. Il primo passo della derivazione $E \Rightarrow E + E$ prevede di aggiungere come radice dell'albero sintattico il simbolo iniziale E e come figli E, $+$, ed E che corrisponde al corpo della produzione $E + E$. Al secondo

passo della derivazione $E \Rightarrow \mathbf{id} + E$ aggiungiamo al nodo più a sinistra E il nodo figlio **id**. Così facendo otteniamo al terzo passo il corrispondente albero sintattico per la stringa **id** + **id**.

2.2.5 Ambiguità

Una grammatica viene definita **ambigua** se produce più di un albero sintattico. In altre parole una grammatica ambigua presenta [1] più di una derivazione destra o sinistra per una frase. Facciamo un esempio. Prendiamo in considerazione la grammatica 2.2 e la frase **id**+**id***__id__; questa frase presenta due alberi di parsing che sono:

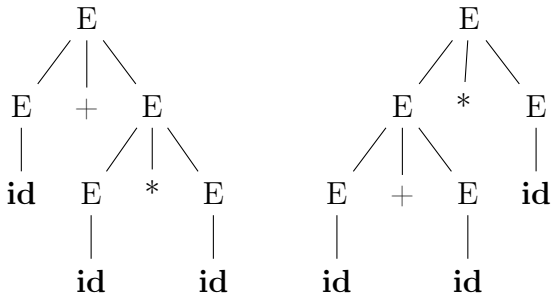

Figura 2.4: *Alberi di parsing relativi alla stringa* **id**+**id***__id__

Di conseguenza ciò dimostra che la grammatica 2.2 risulta essere ambigua.

2.2.6 Ricorsione a sinistra

Una grammatica viene definita **ricorsiva a sinistra** [1] se ha un non-terminale A per cui esiste una derivazione $A \overset{+}{\Rightarrow} A\alpha$ della stringa α. Un esempio di ricorsione a sinistra è la seguente produzione:

$$term \rightarrow term + fact$$

Le grammatiche ricorsive a sinistre risultano essere problematiche da gestire da parser a discesa ricorsiva perchè entrano in un ciclo infinito. Supponiamo che la procedura per il simbolo *expr* decide di applicare questa produzione. Il corpo inizia con *expr* per cui la procedura per *expr* viene invocata ricorsivamente. Poichè il simbolo di lookahead cambia solo quando si verifica una corrispondenza con un terminale del corpo della produzione, nulla cambia sulla stringa in ingresso che si sta analizzando. Di conseguenza la procedura *expr()* viene chiamata di nuovo e così fino all'infinito.

2.3 Parsing top down

Il parsing top down è una tecnica che prevede di costruire l'albero di parsing per una determinata stringa partendo dalla radice dell'albero fino ad arrivare alle foglie che rappresentano i simmboli della stringa. Questo parsing effettua derivazioni a sinistra sulle stringhe che analizza. Infatti ad ogni passo di computazione il parsing top down cerca di trovare un possibile corpo di produzione da sostituire ad ogni non-terminale. Una volta fatto ciò cerca di trovare una corrispondenza tra i simboli della stringa in ingresso e tra i simboli del corpo della produzione. In questo paragrafo analizzeremo i principi e gli strumenti che usa il parsing top down. Verrà presentato il parsing a discesa ricorsiva che richiede *backtracking* per trovare la produzione opportuna da applicare al non-terminale. Successivamente introdurremo le funzioni FIRST e FOLLOW utilizzate per scegliere la produzione da applicare in base al simbolo in input che si sta analizzando. Poi parleremo delle grammatiche LL(1) ed infine dei parser predittivi che usano le funzioni FIRST e FOLLOW per scegliere le produzioni da sostituire.

2.3.1 Parsing a discesa ricorsiva

Un parsing a discesa ricorsiva è un programma che contiene una procedura per ogni non-terminale della grammatica. L'esecuzione [1] inizia con la procedura relativa al simbolo iniziale e termina con successo se il suo corpo scandisce tutta la stringa d'ingresso. Una procedura per un terminale viene mostrato nella figura 2.5.

```
1) void A(){
2)    Scegli, per A, una produzione A → X1, X2 ... Xk;
3)    for(i da 1 fino a k){
4)        if(Xi è non-terminale){
5)          richiama la procedura Xi();
6)        }
7)        else{
8)         if(Xi è uguale al simbolo d'ingresso corrente a){
9)              procedi al simbolo successivo nella sequenza d'ingresso;
10)         }
11)          else{/* si è verificato un errore */;}
12)   }
13) }
```

Figura 2.5: *Procedura di un non-terminale per un parser top down*

Lo pseudocodice mostrato [1] in questa figura è non-deterministico poichè inizia con la scelta di quale produzione utilizzare per A senza indicare come deve essere fatta la scelta. Questo metodo può richiedere backtracking, cioè può richiedere di rileggere più di una volta la stringa in ingresso. Per aggiungere il backtracking al codice in figura 2.5. La linea (2) va tolta e rimpiazzata con istruzioni in cui $\ll$ è necessario provare ognuna delle possibili produzioni secondo un certo ordine. In questo caso il fallimento alla linea (11) non è un fallimento "definitivo", ma indica una necessita di tornare alla linea (2) e provare un'altra produzione. Solo se non vi sono più produzioni per A da provare si segnala che è stato identificato un errore nella stringa d'ingresso. Quindi se vogliamo provare una nuova produzione per A, a seguito di un fallimento, dobbiamo essere in grado di riportare il puntatore alla stringa d'ingresso alla posizione in cui si trovava quando abbiamo raggiunto la linea (2) per la prima volta. $\gg$ Una grammatica ricorsiva a sinistra risulta essere compromettente per questo tipo di parser in quanto può entrare in un ciclo infinito. Per maggiori dettagli si veda il paragrafo 2.2.6

2.3.2 Funzioni FIRST e FOLLOW

Per stabilire quale produzione applicare per sostituire un non-terminale basandoci sui simboli della stringa in input, i parser, sia quelli top-down e bottom-up, usano le funzione FIRST e FOLLOW.

Definiamo **FIRST**(α), [1] in cui α è una generica stringa di simboli della grammatica, come l'insieme dei terminali che costituiscono l'inizio delle stringhe derivabili da α. Se $\alpha \stackrel{*}{\Rightarrow} \epsilon$, allora anche ϵ appartiene all'insieme FIRST.

Definiamo **FOLLOW**(A), in cui A è un non-terminale, come l'insieme dei simboli terminali che possono apparire immediatamente alla destra di Ain qualche forma sentenziale, cioè l'insieme dei terminali a per cui esiste una derivazione nella forma $A \stackrel{*}{\Rightarrow} \alpha A a \beta$, dove α e β sono generiche forme sentenziali. Se A appare come simbolo più a destra di una forma sentenziale, allora \$ appartiene al FOLLOW(A).

1. Se X è non terminale, FIRST(X) = {.
2. Se X è un non-terminale ed esiste una produzione del tipo $X \rightarrow Y_1 Y_2 \ldots Y_k$ con $k \geq 1$, allora si aggiunga a a FIRST(X) se per qualche valore di i, a appartiene a FIRST(Y_i) e ϵ appartiene a tutti gli insiemi FIRST(Y_1), ..., FIRST(Y_{i-1}), cioè se $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Se ϵ appartiene a FIRST(Y_j) per j = 1,2,...,k, allora si aggiunga ϵ all'insieme FIRST(X); se invece $Y_1 \stackrel{*}{\Rightarrow} \epsilon$si aggiunga FIRST($Y_2$) a FIRST($X$), e così via.

3. Se esiste una produzione $X \to \epsilon$, si aggiunga ϵ a FIRST(X).

Per calcolare FOLLOW(A) per tutti i non-terminali A si usano le seguenti regole:

1. Si aggiunga \$ a FOLLOW(S), ricordando che S è il simbolo iniziale e \$ è il marcatore di fine della stringa d'ingresso;

2. Se esiste una produzione del tipo A $\to \alpha B \beta$, allora si aggiunga a FOLLOW(B) ogni elemento di FIRST(β) eccetto ϵ.

3. Se esiste una produzione del tipo $A \to \alpha B$ oppure del tipo $A \to \alpha B \beta$ per cui FIRST(β) contiene ϵ, allora tutti i simboli in FOLLOW(A) appartengono a FOLLOW(B) $\gg$

Facciamo un esempio di come si calcolano FIRST e FOLLOW su una grammatica. Consideriamo la seguente grammatica:

$$\begin{aligned} I &\to A \\ A &\to S \\ S &\to CC \\ C &\to cC \mid d \end{aligned} \tag{2.4}$$

I FIRST e FOLLOW di questa grammatica sono:

1. FIRST(I)=FIRST(A)=FIRST(S)=FIRST(C)={c,d}.

2. FOLLOW(I)=FOLLOW(A)=FOLLOW(S)={\$}.

3. FOLLOW(C)={c,d,\$}.

2.3.3 Gramatiche LL(1)

ceiocmeo

2.3.4 Parsing predittivo non ricorsivo

klmlm

2.4 Conclusioni

nknnin

Bibliografia

[1] Alfred V.Aho, Monica S. Lam, Ravi Sethi,Jeffrey D. Ullman, *Compilatori. Principi, Tecniche e Strumenti. Seconda Edizione*. Pearson, Addison Wesley (2009).

[2] Elizabeth Scott, Adrian Johnstone, *GLL Parsing*. Electronic Notes in Theorical Computer Science 253 (2010) (pp.177-189).

[3] Gennaro Costagliola, Masaru Tomita, Shi-Kuo Chang, *A Generalized Parser for 2-D Languages*, IEEE (1991).