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 top down

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.

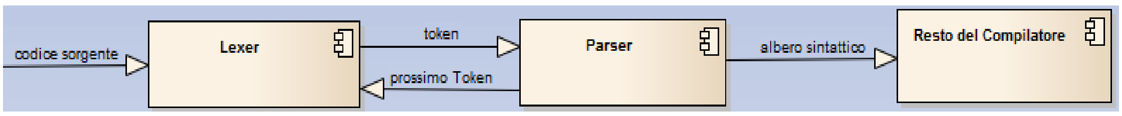

Figura 2.1: *Posizione del parser all'interno del compilatore.*

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 $\alpha_1, \alpha_2, \ldots, \alpha_k$ 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 → 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. ≪ 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 ⇒ 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**.≫ 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

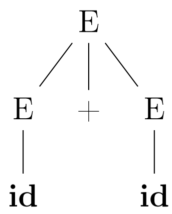

Figura 2.2: *Albero di parsing relativo alla stringa* **id** + **id**

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.

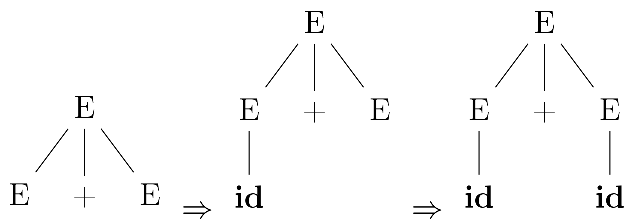

Figura 2.3: *Sequenza di alberi di parsing relativi alla derivazione* 2.3

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 $E \Rightarrow \textbf{id} + E$ 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:

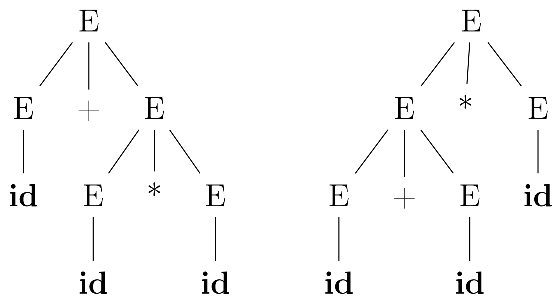

Figura 2.4: *Alberi di parsing relativi alla stringa* **id**+**id*****id**

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 $A \stackrel{+}{\Rightarrow} A\alpha$ 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.

void $A()\{$
  Scegli, per A, una produzione $A \rightarrow X_1,\ X_2 \ldots X_k$;
  for(i da 1 fino a k){
    if(X_i è non-terminale){
      richiama la procedura $X_i()$;
    }
    else{
      if(X_i è uguale al simbolo d'ingresso corrente a){
        procedi al simbolo successivo nella sequenza d'ingresso;
      }
      else{/* si è verificato un errore */;}
  }
}

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}. Per capire il motivo di ciò, si noti che le due produzioni per C hanno i corpi che iniziano con i due simboli terminali c e d. Poichè S, ha solo una produzione che inizia per C e non deriva ϵ, il FIRST(S) coincide con FIRST(C). Lo stesso ragionamento lo si può applicare per FIRST(S) e FIRST(A)

2. FOLLOW(I)=FOLLOW(A)=FOLLOW(S)={\$}. Dato che I è il simbolo iniziale, il FOLLOW(I) deve contenere il carattere speciale \$. Poichè S e A appaiono da sole nel corpo di altre produzioni nè consegue che sono seguite dal simbolo di fine stringa \$. Pertanto il FOLLOW(A) e FOLLOW(S) coincide con il FOLLOW(I).

3. FOLLOW(C)={c,d,\$}. All'interno di una produzione il simbolo C è seguito da un altro simbolo C. Pertanto il FOLLOW(C) include i simboli del FIRST(C). Inoltre essendo che il simbolo C risulta essere l'ultimo simbolo all'interno di una produzione allora il simbolo \$ rientra nel FOLLOW(C).

2.3.3 Grammatiche LL(1)

Un parser predittivo viene sempre costruito a partire da una grammatica della classe LL(1). La prima L [1] indica che la stringa in input che si sta analizzando viene letta da sinistra verso destra. La seconda L specifica che viene fatta una derivazione a sinistra; infine l'1 fra le parentesi indica che le decisioni del parser vengono fatte analizzando un solo simbolo di lookhead. Data una grammatica G con due produzioni $A \to \alpha \mid \beta$ è definita LL(1) se sono verificate le seguenti condizioni: $\ll$

1. non esiste alcun terminale a tale che sia α sia β derivano stringhe che iniziano con a;

2. al più una tra α e β deriva la stringa nulla ϵ;

3. se $\beta \stackrel{*}{\Rightarrow} \epsilon$ non deriva alcuna stringa che inizia con un non terminale appartenente all'inisieme FOLLOW(A); allo stesso modo, se $\alpha \stackrel{*}{\Rightarrow} \epsilon$, allora β non deriva alcuna stringa che inizia con un terminale in FOLLOW(A). $\gg$

Le prime due condizioni verificano che FIRST(α) e FIRST(β) sono insiemi disgiunti. La terza condizione verifica che se ϵ appartiene a FIRST(β) allora FIRST(α) e FOLLOW(A) sono insiemi disgiunti; la stessa cosa vale se ϵ appartiene a FIRST(α). Quindi in base a ciò un parser predittivo per una grammatica LL(1) può essere sempre costruito se ad ogni passo di computazione posso sostituire un non-terminale con una sola produzione che viene scelta in base al simbolo di input corrente. Ovviamente nessuna grammatica ambigua e ricorsiva a sinistra può essere una grammatica LL(1). L'algoritmo seguente raccoglie le informazioni di FIRST e FOLLOW in una **tabella di parsing predittivo**, cioè una matrice bidimensionale $M[A,a]$ dove A è un non-terminale e a è un terminale oppure il simbolo \$. L'algoritmo sceglie la produzione $A \to \alpha$ solo se il simbolo in input a appartiene a FIRST(α). Quando invece abbiamo a che fare con derivazioni $a \stackrel{*}{\Rightarrow} \epsilon$, scegliamo sempre la produzione $A \to \alpha$ se il simbolo corrente appartiene al FOLLOW(A) o si è raggiunto il simbolo \$ nella stringa in input e se tale simbolo appartiene a FOLLOW(A).

Algoritmo 1 Cotruzione di una tabella di parsing predittivo.

INPUT Una grammatica G.
OUTPUT Una tabella di parsing M.
METODO Per ogni produzione $A \to \alpha$ della grammatica G si svolgano i seguenti passi.

1. Per ogni terminale a in FIRST(α) si aggiunga $A \to \alpha$ a $M[A,a]$.

2. Se ϵ appartiene a FIRST(α), allora per ogni terminale b FOLLOW(A) si aggiunga $A \rightarrow \alpha$ a $M[A,b]$. Se ϵ appartiene a FIRST(α) e \$ a FOLLOW(A), si aggiunga $A \rightarrow \alpha$ anche a $M[A,\$]$.

Se dopo questi passi non vi è alcuna produzione in $M[A,a]$, si ha una condizione di errore, che viene indicata vuota nella casella corrispondente.
Facciamo un esempio e prendiamo in riferimento grammatica 2.4. Applichiamo l'algoritmo precedente ed otteniamo la seguente tabella di parsing.

	c	d	\$
I	$I \rightarrow A$	$I \rightarrow A$	
A	$A \rightarrow S$	$A \rightarrow S$	
S	$S \rightarrow CC$	$S \rightarrow CC$	
C	$C \rightarrow cC$	$C \rightarrow d$	

Tabella 2.1: *Tabella di parsing della grammatica* 2.4

Gli spazi bianchi indicano una condizione d'errore, mentre gli altri indicano quale produzione usare per sostituire un non-terminale in base ad un determinato simbolo in input. Si consideri, per esempio la produzione $I \rightarrow A$. Dato che FIRST(A) = FIRST(S) questa produzione viene aggiunta sia $M[A,c]$ e a $M[A,d]$.

2.3.4 Parsing predittivo non ricorsivo

Un parser predittivo non ricorsivo [1] viene costruito usando uno stack, piuttosto che effettuare le chiamate ricorsive. Se w è la porzione dell'ingresso riconosciuta a un certo momento, allora lo stack contiene una sequenza di simboli grammaticali α tali che $S \overset{*}{\Rightarrow} w\alpha$. Questo parser usa un buffer d'ingresso,che contiene anche il simbolo \$ per segnare la fine della stringa, uno stack che contiene i simboli grammaticali e la tabella di parsing costrtuita mediante l'algoritmo 1. Il fondo dello stack viene segnalato con il simbolo \$.
Il parser funziona nel seguente modo:

1. riceve in input una stringa w e una tabella di parsing M relativa a una grammatica G;

2. ad ogni passo di computazione controlla il simbolo X in cima allo stack e un simbolo a della stringa w in input.

3. Se X è un non-terminale, il parser lo sostituisce con il corpo della produzione che si trova nella posizione $M[X,a]$;

4. Altrimenti, se X è un terminale, allora il parser verifica la corrispondenza di X con il simbolo a e se esiste legge il simbolo successivo della stringa w.

5. Ripetere il passo 2.

6. Il parser termina con sucesso se lo stack non contiene nessun simbolo X e ciò determina che la stringa letta fa parte della grammatica G.

Di seguito vengono riportate le mosse del parser predittivo applicate alla grammatica 2.4.

Input	**Stack**	**Azione**	**Riconosciuta**
cdd\$	$I\$$		
cdd\$	$A\$$	output $I \to A$	
cdd\$	$S\$$	output $A \to S$	
cdd\$	$CC\$$	output $S \to CC$	
cdd\$	$cCC\$$	output $C \to cC$	
dd\$	$CC\$$	consuma **c**	**c**
dd\$	$dC\$$	output $C \to d$	**c**
d\$	$C\$$	consuma **d**	**cd**
d\$	$d\$$	output $C \to d$	**cd**
\$	\$	consuma **d**	**cdd**

Tabella 2.2: *Mosse di un parser predittivo sulla stringa* ***cdd***

In questa tabella la cima dello stack è riportata a sinistra nella colonna "Stack". Tali mosse corrispondono alla derivazione sinistra; infatti abbiamo che:

$$I \Rightarrow A \Rightarrow S \Rightarrow CC \Rightarrow cCC \Rightarrow cdC \Rightarrow cdC \Rightarrow cdd \tag{2.5}$$

Si noti che le forme sentenziali in tale derivazione corrispondono alla porzione di stringa in input già analizzata (indicata nella colonna riconosciuta).

2.4 Conclusioni

In questo capitolo è stato discusso di come funziona il parsing top down ed in particolare si è discusso degli algoritmi di parsing LL(1). Questo parser, però, presenta dei limiti:

- Non sono adatti per grammatiche ambigue e ricorsive a sinistre;
- Non ammettono tabelle di parsing in cui vi sono più produzioni per un simbolo d'ingresso.

Delle possibili soluzioni a questi limiti prevedono: eliminazione dell'ambiguità e della ricorsione a sinistra dalla grammatica, l'uso della fattorizzazione a sinistra per rendere la grammatica più adatta al parsing predittivo o l'uso di parsing generalizzati top down che usano il non-determinismo per superare i conflitti che trova un parser predittivo nelle tabelle di parsing. Nel capitolo successivo discuteremo di quest'ultima soluzione.

Capitolo 3

GLL Parsing

3.1 Introduzione

Nel capitolo precedente abbiamo discusso i concetti e il funzionamento del parsing su grammatiche LL(1). In questo capitolo discuteremo di un estenzione di questo parsing, chiamato **Parsing LL Generalizzato (GLL)**. Questo parsing è un parser a discesa ricorsiva ed è adatto a gestire tutte le grammatiche ambigue e ricorsive a sinistra. In questo capitolo vedremo come questo parser supera i limiti che hanno i parser LL(1) e mostreremo i concetti base di questo parsing e del suo funzionamento.

3.2 Stack e descrittori elementari

In questo paragrafo discuteremo del funzionamento base del GLL Parsing. Data la seguente grammatica:

$$\begin{aligned} S &\to ASd \mid BS \mid \epsilon \\ A &\to a \mid c \\ B &\to a \mid b \end{aligned} \tag{3.1}$$

Un parser a discesa ricorsiva [2] è composto dalle seguenti funzioni: $p_S()$, $p_A()$, $p_B()$, la funzione principale $main()$ e la funzione per segnalare gli errori $error()$. Ogni funzione contiene codice per ogni alternativa, α, e verificano il simbolo corrente della stringa in input appartiene a FIRST(α) o al FOLLOW(α). La stringa in input viene rappresentata come un array globale I di lunghezza m+1, dove $I[m]$=\$, segnala la fine della stringa. L'implementazione del parser viene rappresentata di seguito.

main(){ $i = 0$

```
        if(I[i] ∈ {a, b, c, d,$}){ p_S(); else error();
        if(I[i] = $){ report success } else error()
}
p_S(){
      if(I[i] ∈ {a, c}){ p_A(); p_S(); if(I[i] = d){ i = i + 1; } else error(); }
      if(I[i] ∈ {a, b}){ p_B(); p_S(); } }
p_A(){
      if(I[i] = a){ i = i + 1;}
      else if(I[i] = c){ i = i + 1 } else error(); }
p_B(){
      if(I[i] = a){ i = i + 1; }
      else if(I[i] = b){ i = i + 1 } else error(); }
```

Questa è la tabella di parsing della grammatica 3.1.

	a	b	c	d	$
S	$S \rightarrow ASd \mid BS$	$S \rightarrow BS$	$S \rightarrow ASd$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
A	$A \rightarrow a$		$A \rightarrow c$		
B	$B \rightarrow a$	$B \rightarrow b$			

Tabella 3.1: *Tabella di parsing della grammatica* 3.1

Da quello che si può notare dalla tabella 3.1 questa grammatica non è LL(1) in quanto è presente un conflitto e di conseguenza l'algoritmo implementato non funziona correttamente. Affinchè l'algoritmo funzioni correttamente è necessario aggiungere il non-determinismo. Per fare ciò dobbiamo convertire le chiamate a funzioni con operazioni di **push** su uno stack e utilizzare i **goto**. Poi partizioniamo in varie parti i corpi delle funzioni il cui non-terminale non è LL(1) ed attribuiamo un etichetta ad ogni partizione. In questo caso abbiamo più partizioni per S. Per registrare le possibili scelte che il parser può fare per sostituire un non-terminale utilizziamo dei **descrittori** all'interno dell'algoritmo a discesa ricorsiva e sostituiamo il punto in cui termina l'algoritmo con l'esecuzione di un descrittore successivo. Le funzioni d'errore vengono sostituite con l'esecuzione di descrittori successivi. Il nuovo punto di termine sarà quando non esistono più descrittori da eseguire. Formalmente un **descrittore elementare** è una tripla (**L,s,j**) dove **L** è un etichetta, **s** è uno stack e **j** è la posizione nell'array I. Quessti descrittori li manteniamo in un insieme **R**. Ogni volta che si verifica la fine di una funzione di parsing e ad ogni punto in cui è presente un terminale non LL(1) (quindi siamo in presenza di non-determinismo) all'interno dell'algoritmo, creiamo un nuovo descrittore che è formato dall'etichetta in cima allo stack corrente.

Quando l'algoritmo di parsing trova un simbolo del'input $I[i]$ diciamo che l'etichetta L in cima allo stack è estratto dallo stack s=[$s^{'}$,L] e (L,$s^{'}$,i) viene aggiunta a **R**. Questa azione viene denotata con la funzione *pop(s,i,**R**)*. Dopo aver fatto ciò rimuoviamo il descrittore ($L^{'}$, t, j) da **R** e l'algoritmo riparte dall'etichetta $L^{'}$, con stack t e con il simbolo in input $I[j]$. L'algoritmo termina quando l'insieme **R** è vuoto. Useremo la notazione L^k per unire l'etichetta L e l'indice k che indica il simbolo corrente nell'input I; mentre lo stack vuoto viene denotato con [ ]. Lo stack s viene aggiornato con la funzione *push(s, L^k)*; questa funzione non fa altro che aggiungere l'elemento L^k in cima allo stack. Di seguito viene presentato l'algoritmo.

i = 0; **R** = $\emptyset$; s = [L_0^0];
L_S: **if**($I[i] \in \{a,\ c\}$) add(L_{S1}, s, i) to **R**
  if($I[i] \in \{a,\ b\}$) add(L_{S2}, s, i) to **R**
  if($I[i] \in \{d,\ \$\}$) add($L_{S3}$, s, i) to **R**
L_0: **if**(**R** $\neq \emptyset$){ remove(L, s_1, j) from **R**
  if(L = L_0 and s_1 = [ ] and j = |I|) report success
  else{ s = s_1; i = j; **goto** L }
L_{S1}: *push*(s, L_1^i); **goto** L_A
L_1: *push*(s, L_2^i); **goto** L_S
L_2: **if**($I[i]$ = a){ i = i + 1; *pop*(s, i, **R**);} **goto** L_0
L_{S2}: *push*(s, L_3^i); **goto** L_B
L_3: *push*(s, L_4^i); **goto** L_S
L_4: *pop*(s, i, **R**); **goto** L_0
L_{S3}: *pop*(s, i, **R**); **goto** L_0
L_A: **if**($I[i]$ = a){ i = i + 1; *pop*(s, i, **R**);} **goto** L_0 }
  else{ **if**($I[i]$ = c){ i = i + 1; *pop*(s, i, **R**);}
    goto L_0 }
L_B: **if**($I[i]$ = a){ i = i + 1; *pop*(s, i, **R**);} **goto** L_0 }
  else{ **if**($I[i]$ = b){ i = i + 1; *pop*(s, i, **R**);}
    goto L_0 }

Ora facciamo un esempio ed eseguiamo l'algoritmo con la stringa d'input *aad*\$. Incomiciamo la nostra computazione aggiungendo prima la tripla (L_{S1},[L_0^0],0), e poi la tripla (L_{S2},[L_0^0],0) ad **R** ed andiamo all'etichetta L_0. Rimuoviamo (L_{S1},[L_0^0],0) da **R** ed andiamo alla linea con etichetta L_{S1}. L'operazione di *push* aggiunge allo stack s [L_0^0,L_1^0] ed andiamo in L_A. In L_A abbiamo che il parser ha trovato una coincidenza con il simbolo a, incrementa l'indice i, usato per leggere i simboli dell'input, e fa un operazione di *pop* ed aggiunge (L_1,[L_0^0],1) ad **R** e ritorna in L_0. Allo stesso modo processiamo (L_{S2},[L_0^0],0) da **R** e alla fine avremo (L_3,[L_0^0],1) che viene aggiunto ad **R**. Quindi **R** risulterà avere le seguenti triple:

$$\mathbf{R} = \{\ (L_1,[L_0^0],1),\ (L_3,[L_0^0],1)\ \}$$

Successivamente estraiamo (L_1,[L_0^0],1) e lo processiamo. Alla linea L_1 facciamo un *push* sullo stack s [L_0^0,L_2^1] ed andiamo alla linea con etichetta L_S e aggiungiamo (L_{S1},[L_0^0,L_2^1],1) e (L_{S2},[L_0^0,L_2^1],1) ad **R**. Allo stesso modo processiamo (L_3,[L_0^0],1) e in **R** abbiamo:

$$\mathbf{R} = \{\ (L_{S1},[L_0^0,L_2^1],1),\ (L_{S2},[L_0^0,L_2^1],1),\ (L_{S1},[L_0^0,L_4^1],1),\ (L_{S2},[L_0^0,L_4^1],1)\ \}$$

Processando ognuno di questi elementi otteniamo:

$$\mathbf{R} = \{\ (L_1,[L_0^0,L_2^1],2),\ (L_3,[L_0^0,L_2^1],2),\ (L_1,[L_0^0,L_4^1],2)\ (L_3,[L_0^0,L_4^1],2)\ \}$$

Quando l'input risulta essere $I[i] = d$ e processiamo queste triple otteniamo che in **R** abbiamo:

$$\mathbf{R} = \{\ (L_{S3},[L_0^0,L_2^1,L_2^2],2),\ (L_{S3},[L_0^0,L_2^1,L_4^2],2),\ (L_{S3},[L_0^0,L_4^1,L_4^2],2)\ (L_{S3},[L_0^0,L_4^1,L_4^2],2)\}$$

Processando di nuovo questi elementi otteniamo:

$$\mathbf{R} = \{\ (L_2,[L_0^0,L_2^1],2),\ (L_4,[L_0^0,L_2^1],2),\ (L_2,[L_0^0,L_4^1],2)\ (L_4,[L_0^0,L_4^1],2)\}$$

Da queste triple otteniamo:

$$\mathbf{R} = \{\ (L_2,[L_0^0],3),\ (L_2,[L_0^0],2),\ (L_4,[L_0^0],3)\ (L_4,[L_0^0],2)\}$$

Quando abbiamo $I[3] = \$$ processiamo le triple mostrate precedentemente ed otteniamo la tripla (L_0,[ ],3) e la tripla (L_0,[ ],2) che vengono aggiunte ad **R** e di conseguenza l'algoritmo termina con un successo.

3.3 Graph structured stacks

Nel paragrafo precedente abbiamo visto che ogni volta si trovava un non-terminale non LL(1) applicavamo il non-determinismo duplicando lo stack usato dal parser. Per rappresentare tutti questi stack in unica struttura dati utilizzeremo il **Graph structured stacks**(**GSS**). Viene definito [4] come un grafo diretto aciclico (DAG) i cui nodi hanno etichette L^k che corrispondono agli elementi usati dallo stack, dove L indica un etichetta di una linea di codice e k la posizione su un simbolo della stringa in input. Questi nodi sono raggruppati in vari insiemi disgunti chiamati *livelli*. Il GSS viene costruito un livello alla volta. Infatti ogni qualvolta il parser trova una coincidenza tra un simbolo in input e la grammatica crea un livello. Il GSS viene disegnato da sinistra verso destra ed il nodo più a sinitra rappresenta la cima di ogni stack. Per costruire il GSS all'interno dell'algoritmo di parsing, dobbiamo usare una nuova tripla, chiamato **descrittore**. Un descrittore è formata da (L, u, i), dove L è un etichetta, u è un nodo del GSS e i che indica il simbolo della stringa in input $I[\ ]$. Queste triple vengono aggiunte ad **R**.

3.3.1 Insiemi U e P

Per costruire il GSS è necessario utilizzare un altro insieme **U** che contiene gli stessi descrittori che inseriamo nell'insieme **R**. Infatti abbiamo che $\mathrm{U}_i = \{ (L, u) \mid (L, u, i)$ è stato aggiunto ad $\mathbf{R}\}$. Un problema [2] che può sorgere sia ha quando un nodo figlio w è aggiunto ad u dopo aver eseguito le operazioni di pop sul GSS perchè l'azione di pop necessita di essere applicata a questo nodo figlio. Per risolvere ciò usiamo l'insieme **P** che contiene coppie (u, k) e verranno utilizzate per eseguire le operazioni di pop. Infatti quando un nuovo nodo figlio w è aggiunto ad u, ogni elemento (u, k) presente in **P**, se (L_u, w) non è presente in $\mathbf{U}_k$, allora (L_u, u, k) è aggiunto ad **R**, dove L_u è l'etichetta nel nodo u. L'implementazione di questa tecnica verrà mostrata nel paragrafo 3.4.1

3.4 Definizione GLL Parsing

In questo paragrafo vengono definite le basi per realizzare l'algoritmo di GLL Parsing. Mostreremo come costruire il GSS e il suo l'output. Il principio base è quello mostrato al paragrafo 3.2

3.4.1 Funzioni Fondamentali

L'algoritmo [2] utilizza quattro funzioni che sono essenziali per il suo funzionamento. Queste funzioni vengono elencate qui di seguito.

- ***add()***: La funzione $add(L, u, j)$ controlla se c'è un descrittore (L, u) in U_j e se non c'è lo aggiunge a U_j e ad **R**. La funzione è definita nel seguente modo:

  $add(L, u, j)\{$ **if**$((L, u) \notin \mathrm{U}_j)\{$ add (L, u) to U_j; add (L, u, j) to **R** } }

- ***pop()***: La funzione $pop(u, j)$ chiama la funzione $add(L_u, v, j)$ per tutti i figli v di u e aggiunge (u, j) a **P**. Viene definita nel seguente modo:

  $pop(u, j)\{$ **if**$(u \neq u_0)\{$ add (u, j);
  for each child v of u { add (L_u, v, j); } } }

- ***create()***: La funzione *create*$(L,\ u,\ j)$ crea un nodo v nel GSS etichettato L^j con figlio u se non esiste ancora e restituisce v. Se $(v,\ k)$ appartiene a $\mathbf{P}$ chiama la funzione *add*$(L,\ u,\ k)$. La definizione di questa funzione è al seguente:

  create$(L,\ u,\ j)\{$ **if** there is not a GSS node labelled L^j create one
  let v be the GSS node labelled L^j
  if there is not an edge from v to $v\{$
  create an edge from v to u
  for all $((v,\ k) \in \mathbf{P})\ \{\ add(L,\ u,\ k)\ \}\ \}$
  return $v\ \}$

- ***test()***: La funzione *test*$(x,\ A,\ \alpha)$ controlla se il simbolo d'input corrente x appartiene al FOLLOW(A), dove A è un non-terminale o appartiene al FIRST(α), dove α è un item che stiamo processando. È definita nel seguente modo:

  test$(x{,}A{,}\alpha)\{$ **if**$(x \in \mathrm{FIRST}(\alpha))$ or $(\epsilon \in \mathrm{FIRST}(\alpha)$ and $x \in \mathrm{FOLLOW}(A))\{$
  return true $\}$
  else $\{$ **return false** $\}\ \}$

3.4.2 Gestione degli item

Informalmente un **item** [1] di una grammatica G è una produzione di G con un punto in qualche posizione del corpo. Ad esempio la produzione $C \rightarrow DKL$ ammette quattro item:

$$C \rightarrow .DKL$$
$$C \rightarrow D.KL$$
$$C \rightarrow DK.L$$
$$C \rightarrow DKL.$$

La produzione $C \rightarrow \epsilon$ genera l'item $C \rightarrow .$.
Un item indica una porzione di una produzione che si sta analizzando ad un certo punto del processo di parsing. Sia α la parte di produzione dopo il punto di un item. Nel GLL Parsing gli item verrano gestiti nel seguente modo:

1. Un item del tipo $a\alpha$, dove a è un terminale, definiamo:

   code$(a\alpha$, j$)$ = **if**$(I[j] = a)\{\ j = j + 1\ \}$ **else** $\{$ **goto** $\mathrm{L}_0\ \}$

2. Un item del tipo $A\alpha$, dove A è un non-terminale, definiamo:

   $\mathit{code}(A\alpha, \text{j}, X) = \textbf{if}(\mathit{test}(I[j], X, A\alpha))\{$
   $c_u = \mathit{create}(R_A\alpha, c_u, j)$; **goto** L_A }
   else { **goto** L_0 }

3. Per la produzione $A \rightarrow \epsilon$ con item $C \rightarrow .$ definiamo:

   $\mathit{code}(A \rightarrow \epsilon, \text{j}) = \mathit{pop}(c_u, j)$; **goto** L_0;

Quindi, in base a ciò, per ogni produzione $A \rightarrow \beta$, dove $\beta = x_1 \ldots x_n$, abbiamo:

1. Se x_1 è un terminale:

   $\mathit{code}(A \rightarrow \beta, \text{j}) =$ $j = j + 1$
   $\mathit{code}(x_2 \ldots x_n, j, A)$
   $\mathit{code}(x_3 \ldots x_n, j, A)$
   ...
   $\mathit{code}(x_n, j, A)$
   $\mathit{pop}(c_u, j)$; **goto** L_0;

2. Se x_1 è un non-terminale:

   $\mathit{code}(A \rightarrow \beta, \text{j}) =$ $c_u = \mathit{create}(R_{Al}, c_u, j)$; **goto** L_A
   A_l: $\mathit{code}(x_2 \ldots x_n, j, A)$
   $\mathit{code}(x_3 \ldots x_n, j, A)$
   ...
   $\mathit{code}(x_n, j, A)$
   $\mathit{pop}(c_u, j)$; **goto** L_0;

3.4.3 Gestione delle produzioni

Siano $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ regole grammaticali. Quando dobbiamo gestire le sostituzioni dei non-terminali con gli opportuni corpi di produzione durante il parsing vengono definiti due modi:

1. Se A è un non-terminale LL(1) che non presenta conflitti abbiamo:

   $\mathit{code}(A, \text{j}) =$ $\textbf{if}(\mathit{test}(I[j], X, \alpha_1)\{$ **goto** L_{A1} }
   ...

else if$(test(I[j],X,\alpha_n)\{$**goto** $L_{An}\}$

L_{A1}: $code(A{\to}\alpha_1$, j)

. . .

L_{An}: $code(A{\to}\alpha_n$, j)

2. Se A è un non-terminale non LL(1) che presenta conflitti abbiamo:

$code(A$, j) = **if**$(test(I[j],X,\alpha_1)\{\ add(L_{A1},c_u,j)\}$

. . .

else if$(test(I[j],X,\alpha_n)\{\ add(L_{An},c_u,j)\}$

goto L_0

L_{A1}: $code(A{\to}\alpha_1$, j)

. . .

L_{An}: $code(A{\to}\alpha_n$, j)

3.4.4 Shared packed parse forests

Gli alberi sintattici prodotti da una grammatica ambigua [4] possono essere combinati in unica struttura chiamata **Shared packed parse forests** (Sppf). I nodi padri sono uniti in nuovo nodo ed un nodo involucro diventa il nodo padre di ogni sottoalbero. Per la grammatica 2.2, che può essere rappresentata con due alberi di parsing, il corrispondente sppf è mostrato nella figura 3.1. Il parsing GLL restituirà in output un sppf in quanto opera su

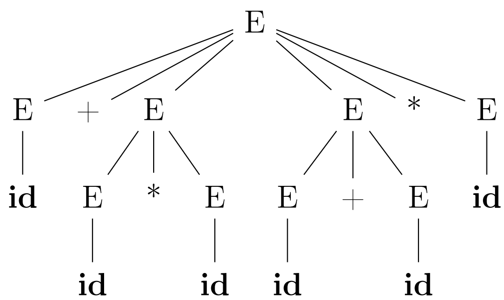

Figura 3.1: *Sppf relativo alla stringa* **id**+**id***id**

grammatiche ambigue e ricorsive a sinistra. Per poter costruire questo albero sono state definite due funzioni:

- ***getNodeT()***: la funzione $getNodeT(\alpha,c_n)$ viene usata per creare un nodo, etichettato α, all'interno del sppf e collegarlo come nodo figlio al nodo c_n. Viene chiamata quando il parser trova un terminale e quando sostituiamo un non-terminale con il corpo della produzione. Viene definita nel seguente modo:

  $getNodeT(\alpha,c_u)\{$ create a new node c_v in SPPF;
  create an adge from c_u to c_v $\}$

- ***getNodeP()***: la funzione $getNodeP(c_u)$ viene usata per ottenere un nodo c_v padre del nodo c_u all'interno del sppf. Viene chiamata quando è necessario recuperare un non-terminale ogni volta che viene trovato un simbolo della stringa in input. In questo modo recuperiamo i restanti non-terminali del corpo di una produzione. Viene definita nel seguente modo:

  $getNodeP(c_u)\{$ **for** each edge E(c_p,c_c)in Sppf {
  if(c_u=c_c){ **return** c_c } } }

3.5 Costruzione del GLL Parser

Ora, avendo definito le varie funzioni e le varie operazioni da eseguire, siamo in grado di costruire in GLL Parser. Qui di seguito riportiamo lo pseudocodice del parser e nei capitoli successivi ne vedremo l'implementazione in Java di questo parser.

Sia Γ una grammatica [2] e i suoi non-terminali sono A, ..., X. L'algoritmo di GLL parsing per la grammatica Γ risulta essere il seguente:

m è una costante che indica la lunghezza della stringa in input
I è un array che contiene la stringa in input di dimensione $m+1$
i è una variabile intera usata per accedere alle locazioni dell'array I
GSS è un DAG che contiene i nodi etichettati nella forma L^j
c_u è un nodo del GSS
P è un insieme che ha coppie formate da un nodo del GSS e di intero
R è un insieme di descrittori

read the input into I and set $I[m]$=\$, i=0;
create GSS nodes $u_1=LL_0^0$, u_0=\$ and edge (u_0,u_1)
$c_u=u_1$, $i=0$
for $0 \leq j \leq m$ $\{$ $\mathbf{U}_j = \emptyset$ $\}$
$\mathbf{R}=\emptyset$, $\mathbf{P}=\emptyset$

goto L_S;
L_0: **if**($\mathbf{R} \neq \emptyset$){
remove a descriptor (L,u,j) from $\mathbf{R}$
c_u=u, i=j , **goto** L }
else if($(L_0,u_0,j) \in U_m$){ report success } **else**{ report failure }
L_A: *code*(A,i)
. . .
L_X: *code*(X,i)

Capitolo 4

Grammatiche Posizionali

4.1 Introduzione

uguihiu

4.2 Definizione formale

niloi

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).

[4] Giorgios R. Economopoulus, *Generalized LR parsing algorithms*. Tesi di dottorato, Royal Holloway, University of London (2006).