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 2019-2020

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 Introduzione

Questa tesi di laurea descrive il funzionamento e l'implementazione del parsing **Generalizzato LL (GLL)**. Il parsing GLL è un algoritmo di parsing top down che viene utilizzato per gestire tutte le grammatiche context-free comprese quelle che sono ambigue e ricorsive. La caratteristica principale di questo algoritmo è che risulta essere un parser a **discesa ricorsiva** e ciò permette di avere il controllo del flusso sulla struttura della grammatica e di conseguenza risulta semplice da implementare e da testare. L'obiettivo da raggiungere sarà quello di far riconoscere al GLL parsing i linguaggi non lineari (bidimensionali) prodotti dalle grammatiche posizionali. La tesi è divisa in tre parti. Nella prima parte si illustreranno i principi su cui si basa il funzionamento del GLL parsing e verrà descritto il parsing top down, che rappresenta la base di funzionamento del GLL parsing, e i suoi limiti. Successivamente si introdurrà il GLL parsing, illustrandone i principi e i meccanismi che usa per superare i limiti dei parser top down tradizionali, le strutture dati che utilizza e il risultato ottenuto dalla sua computazione. Ciò viene descritto rispettivamente nel secondo e terzo capitolo. Nella seconda parte verrà descritto il funzionamento del GLL parsing che opera su grammatiche posizionali per riconoscere i linguaggi non lineari. Questo argomento sarà trattato nel quarto capitolo. Infine, nell'ultima parte si parlerà dell'implementazione del GLL parsing applicato ad una grammatica context-free e ad una grammatica posizionale. Ciò verrà descritto nel quinto e sesto capitolo. La tesi si conclude al settimo capitolo in cui vengono descritti i risultati e gli obiettivi raggiunti.

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. Ovviamente, il parser è in grado segnalare gli errori delle forme sintattiche sbagliate. In figura 2.1 viene mostrato il funzionamento del parser.

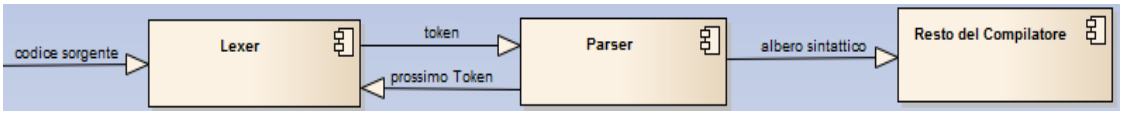

Figura 2.1: *Posizione del parser all'interno del compilatore.*

I metodi di parsing più comunemente utilizzati 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:

$$\textbf{while} \quad (espressione) \quad statement$$

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:

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

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: $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$. 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 di una delle sue 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 allora 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 posizionato 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**.≫ 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 passo vengano fatte due scelte: la prima scelta consiste nello scegliere il non-terminale 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.

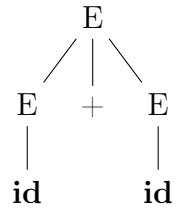

Figura 2.2: *Albero di parsing relativo alla stringa* **id** + **id**

Le foglie dell'albero di parsing sono etichettate con terminali o 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.

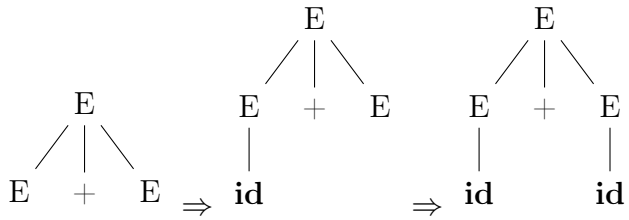

Figura 2.3: *Sequenza di alberi di parsing relativi alla derivazione* 2.3

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 suo corpo di 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:

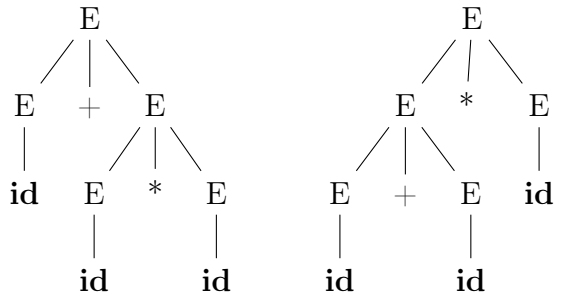

Figura 2.4: *Alberi di parsing relativi alla stringa* **id**+**id***id

Di conseguenza risulta che la grammatica 2.2 risulta essere ambigua.

2.2.6 Ricorsione

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

$$term \rightarrow term + fact$$

Le grammatiche ricorsive risultano essere problematiche da gestire dai 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 che vengono utilizzate per scegliere le produzioni da sostituire basandosi sul simbolo in input che si sta analizzando. Poi parleremo delle grammatiche LL(1) ed infine dei parser predittivi non ricorsivi.

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 non-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 è necessario che alla 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 la necessità 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 funzioni 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).

$\ll$ Per calcolare FIRST$\{X\}$ si applicano le seguenti regole finchè non sia più possibile aggiungere all'insieme FIRST nè nuovi terminali, nè ϵ.

1. Se X è non terminale, FIRST(X) = $\{X\}$.

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 $X \rightarrow \epsilon$, 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 $\rightarrow \alpha B \beta$, allora si aggiunga a FOLLOW(B) ogni elemento di FIRST(β) eccetto ϵ.

3. Se esiste una produzione del tipo $A \rightarrow \alpha B$ oppure del tipo $A \rightarrow \alpha B \beta$ 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:

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

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 $A \rightarrow \alpha \mid \beta$ è definita LL(1) se sono verificate le seguenti condizioni: ≪

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). ≫

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 \$, cioè il marcatore di fine sequenza. L'algoritmo sceglie la produzione $A \rightarrow \alpha$ 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 $A \rightarrow \alpha$ 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 Costruzione di una tabella di parsing predittivo.

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

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

2. Se ϵ appartiene a FIRST(α), allora per ogni terminale b FOLLOW(A) si aggiunga $A \to \alpha$ a $M[A,b]$. Se ϵ appartiene a FIRST(α) e \$ a FOLLOW(A), si aggiunga $A \to \alpha$ 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 la grammatica 2.4. Applichiamo l'algoritmo precedente ed otteniamo la seguente tabella di parsing.

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

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 $I \to A$. 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 $S \stackrel{*}{\Rightarrow} w\alpha$. Questo parser usa un buffer d'ingresso, che contiene la stringa da analizzare compreso anche il simbolo \$ per segnare la fine della stringa, uno stack che contiene i simboli grammaticali e la tabella di parsing costruita 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 dal passo 2.
6. Il parser termina con successo 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:

$$I \Rightarrow A \Rightarrow S \Rightarrow CC \Rightarrow cCC \Rightarrow cdC \Rightarrow cdd$$

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;
- Non ammettono tabelle di parsing in cui vi sono più produzioni per un simbolo d'ingresso.

Delle possibili soluzioni a questi limiti prevedono: l'eliminazione dell'ambiguità e della ricorsione 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 estensione di questo parsing, chiamato **Parsing LL Generalizzato (GLL)**. Questo parsing è un parser a discesa ricorsiva ed è adatto a gestire tutte le grammatiche comprese quelle che risultano essere ambigue e ricorsive a sinistra. In questo capitolo vedremo come questo parser supera i limiti che hanno i parser LL(1) e ne mostreremo i concetti base di questo parsing e il suo funzionamento.

3.2 Stack e descrittori elementari

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

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

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

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 utilizzare il non-determinismo. Per fare ciò dobbiamo convertire le chiamate a funzioni con operazioni di **push** su uno stack e utilizzare i **goto** per spostarci sulle varie partizioni delle funzioni. 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 avremo 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 di una partizione, **s** è uno stack e **j** è la posizione nell'array *I*. Questi 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 non-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 dell'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.

$i = 0$; $\mathbf{R} = \emptyset$; $s = [L_0^0]$;
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**($\mathbf{R} \neq \emptyset$){ 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**($I[i] = a$){ $i = i + 1$; $pop(s, i, \mathbf{R})$;} **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: $pop(s, i, \mathbf{R})$; **goto** L_0
L_{S3}: $pop(s, i, \mathbf{R})$; **goto** L_0
L_A: **if**($I[i] = a$){ $i = i + 1$; $pop(s, i, \mathbf{R})$;} **goto** L_0 }
else{ **if**($I[i] = c$){ $i = i + 1$; $pop(s, i, \mathbf{R})$;}
goto L_0 }
L_B: **if**($I[i] = a$){ $i = i + 1$; $pop(s, i, \mathbf{R})$;} **goto** L_0 }
else{ **if**($I[i] = b$){ $i = i + 1$; $pop(s, i, \mathbf{R})$;}
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 per leggere il simbolo successivo, fa un operazione di *pop* ed aggiunge (L_1,[L_0^0],1) ad **R** e ritorna in L_0. Allo stesso modo processia-

mo (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:

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

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 Definizione GLL Parsing

L'algoritmo descritto al paragrafo 3.2 potrebbe creare un numero esponenziale, della taglia dell'input, di descrittori per alcune grammatiche e non lavora correttamente su grammatiche ricorsive a sinistra. In questo paragrafo vedremo come correggere l'algoritmo e introdurremo le nuove strutture dati utilizzate dall'nuovo algoritmo.

3.3.1 Graph structured stacks

Nell'algoritmo presentato precedentemente, 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 partizione di una funzione 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 la posizione di un simbolo della stringa in input $I[\,]$. Queste triple vengono aggiunte ad **R**.

3.3.2 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 $U_i = \{$ (L,u) | (L, u, i) è stato aggiunto ad **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 3.3.3

3.3.3 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**$((L, u) \notin U_j)\{$ 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 $\mathbf{P}$. 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**$(x \in \text{FIRST}(\alpha))$ or $(\epsilon \in \text{FIRST}(\alpha)$ and $x \in \text{FOLLOW}(A))\{$
  return true $\}$
  else $\{$ **return false** $\}$ $\}$

3.3.4 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 dell'item. Nel GLL Parsing gli item verrano gestiti nel seguente modo:

1. Un item del tipo $a\alpha$, dove a è un terminale, definiamo:
   $code(a\alpha, \text{j}) = \textbf{if}(I[j] = a)\{\ j = j + 1\ \}\ \textbf{else}\ \{\ \textbf{goto}\ \text{L}_0\ \}$
2. Un item del tipo $A\alpha$, dove A è un non-terminale, definiamo:
   $code(A\alpha, \text{j},X) = \textbf{if}(test(I[j],X,A\alpha))\{$
   $c_u = create(R_A\alpha,c_u,j);\ \textbf{goto}\ L_A\ \}$
   $\textbf{else}\ \{\ \textbf{goto}\ \text{L}_0\ \}$
3. Per la produzione $A{\rightarrow}\epsilon$ con item $C{\rightarrow}.$ definiamo:
   $code(A{\rightarrow}\epsilon, \text{j}) = pop(c_u,j);\ \textbf{goto}\ \text{L}_0;$

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:
   $$\begin{aligned} code(A{\rightarrow}\beta, \text{j}) = \quad & j = j + 1 \\ & code(x_2 \ldots x_n,\ j,A) \\ & code(x_3 \ldots x_n,\ j,A) \\ & \ldots \\ & code(x_n,\ j,A) \\ & pop(c_u,j);\ \textbf{goto}\ \text{L}_0; \end{aligned}$$
2. Se x_1 è un non-terminale:
   $$\begin{aligned} code(A{\rightarrow}\beta, \text{j}) = \quad & c_u{=}create(R_{Al},c_u,j);\ \textbf{goto}\ L_A \\ A_l{:}\ & code(x_2 \ldots x_n,\ j,A) \\ & code(x_3 \ldots x_n,\ j,A) \\ & \ldots \\ & code(x_n,\ j,A) \\ & pop(c_u,j);\ \textbf{goto}\ \text{L}_0; \end{aligned}$$

3.3.5 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:

$$
\begin{aligned}
code(A, \text{j}) = \quad & \textbf{if}(test(I[j],X,\alpha_1)\{\ \textbf{goto}\ L_{A1}\ \} \\
& \ldots \\
& \textbf{else if}(test(I[j],X,\alpha_n)\{\textbf{goto}\ L_{An}\} \\
& L_{A1}\colon\ code(A{\rightarrow}\alpha_1, \text{j}) \\
& \ldots \\
& L_{An}\colon\ code(A{\rightarrow}\alpha_n\ , \text{j})
\end{aligned}
$$

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

$$
\begin{aligned}
code(A, \text{j}) = \quad & \textbf{if}(test(I[j],X,\alpha_1)\{\ add(L_{A1},c_u,j)\} \\
& \ldots \\
& \textbf{else if}(test(I[j],X,\alpha_n)\{\ add(L_{An},c_u,j)\} \\
& \textbf{goto}\ L_0 \\
& L_{A1}\colon\ code(A{\rightarrow}\alpha_1, \text{j}) \\
& \ldots \\
& L_{An}\colon\ code(A{\rightarrow}\alpha_n\ , \text{j})
\end{aligned}
$$

3.3.6 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. La grammatica 2.2, che viene rappresentata con due alberi di parsing, il corrispondente sppf è mostrato nella figura 3.1. Il

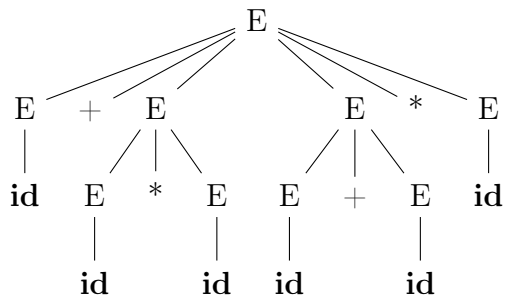

Figura 3.1: *Sppf relativo alla stringa* **id**+**id***id

parsing GLL restituirà in output un sppf in quanto combinerà tutti gli alberi

sintattici calcolati dai vari flussi di computazione creati dall'applicazione del non-determinismo. Per poter costruire questo albero sono state definite due funzioni:

- ***getNodeT()***: la funzione $getNodeT(\alpha, c_n, \mathrm{a}x_1 \dots x_n)$ viene usata per inserire all'interno del sppf un nodo etichettato α, che corrisponde ad un simbolo terminale e non, e $\mathrm{a}x_1 \dots x_n \mathrm{a}x_1 \dots x_n$, che equivale all'item a cui appartiene. Questo nodo creato viene collegato come nodo figlio al nodo c_n. Viene definita nel seguente modo:

  $getNodeT(\alpha, c_n, \mathrm{a}x_1 \dots x_n)\{$ **for** each edge E(c_p,c_c) in SPPF
  if((c_n=c_p)and($\mathrm{a}x_1 \dots x_n$=*item of* c_c))\{
  create a new node c_v=(c_a, $\mathrm{a}x_1 \dots x_n$);
  create an edge from c_n to c_v in SPPF;\}\}

- ***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 \} \} \}

3.4 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 [1] e nei capitoli successivi ne vedremo l'implementazione.

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

[1] In questo pseudocodice è stata omessa la costruzione del sppf. Verrà discusso nei capitoli successivi.

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=L_0^0, u_0=\$ and edge (u_0,u_1)
c_u=u_1, i=0
for $0 \leq j \leq m$ { $\mathbf{U}_j = \emptyset$ }
R=$\emptyset$, **P**=$\emptyset$
goto L_S;
L_0: **if**(**R** $\neq \emptyset$){
remove a descriptor (L,u,j) from **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

GLL Parsing Posizionale

4.1 Introduzione

In questo capitolo introduciamo un estensione del GLL parsing, il **GLL Parsing Posizionale**. Il GLL Parsing Posizionale permette di trattare le *grammatiche posizionali* che producono i cosidetti **linguaggi non lineari**. Queste grammatiche presentano produzioni contenenti *relazioni spaziali*, ossia relazioni che indicano la direzione di come deve essere il simbolo successivo dal testo in input. Illustreremo la definizione base di questa nuova grammatica e il funzionamento base delle relazioni spaziali.

4.2 Definizione formale

Una **grammatica posizionale context-free** è una sestupla [3] i cui elementi sono:

1. **Non-Terminali (N)**: variabili sintattiche che denotano un insieme di stringhe.
2. **Terminali (T)**: simboli di base che definiscono il linguaggio.
3. **Simbolo iniziale (S)**: è un non-terminale e l'insieme di stringhe che esso denota coincide con l'intero linguaggio generato dalla grammatica.
4. **Produzioni (P)**: regole che definiscono come possono essere combinati i terminali e i non-terminali.
5. **Relazioni Spaziali (POS)**: relazioni che danno informazioni sulle posizioni di spostamento da effettuare per la lettura del simbolo successivo.

6. **Regole di valutazione (PE)**: regole usate per determinare come deve essere effettuato lo spostamento sull'input.

Ogni produzione presenta la seguente forma:

$$A \to \alpha_1 REL_1 \alpha_2 REL_2 \dots REL_{m-1} \alpha_n \quad m \geq 1$$

dove $A \in$ N, ogni $\alpha_i \in$ P ed ogni $REL_1 \in$ POS.

Ogni relazione spaziale REL_1 dà informazioni sulla posizione relativa ad α_{i+1} rispetto ad α_i. Nelle grammatiche tradizionali abbiamo una sola relazione spaziale che è data dalla concatenazione della stringa, nelle grammatiche posizionali possiamo definire altre relazioni spaziali ed in questo modo possiamo usarli per descrivere i linguaggi bidimensionali. In questi linguaggi, le informazioni posizionali vengono usate dal lexer per leggere il prossimo simbolo.

4.2.1 Relazioni spaziali

L'insieme di relazioni spaziali [3] possono essere suddivisi in 3 sottoinsiemi: *type*-2, *type*-1, *type*-0. Ogni sottoinsieme include quattro relazioni spaziali direzionali che sono: sopra, sotto, destra e sinistra. Ogni direzione può essere descritta da una funzione che prende in input una coordinata nel piano cartesiano e restituisce in output un insieme di posizioni. Qui di seguito mostriamo le definizioni dei tre tipi di relazioni spaziali ed illustreremo solo le funzioni descritte; le altre possono essere ricavate in maniera molto simile.

Type-2
$\to$ spostamento di un movimento a destra: $(x,\ y) \to (x{+}1,\ y)$
$\leftarrow$ spostamento di un movimento a sinistra
$\uparrow$ spostamento di un movimento in alto
$\downarrow$ spostamento di un movimento in basso

Type-1
$\to_1$ spostamento a destra sulla stessa linea:
$(x,\ y) \to \{\to^{(n)} (x,y) \mid \text{n}{=}1,2,\dots\}$
$\leftarrow_1$ spostamento a sinistra sulla stessa linea
$\uparrow_1$ spostamento in alto sulla stessa linea
$\downarrow_1$ spostamento in basso sulla stessa linea

Type-0
$\to_0$ spostamento a destra:
$(x,y) \to \to_0(x,y) \cup \downarrow_1(x',y') \cup \to_1 (x',\ y')$ dove $(x',\ y')$ cambia in$\to_0(x,y)$
$\leftarrow_0$ spostamento a sinistra

$\uparrow_0$ spostamento in alto
$\downarrow_0$ spostamento in basso

C'è da precisare, quindi, che ogni simbolo presente in uno spazio bidimensionale viene identificato conoscendo sempre la sua posizone. Infatti, se diamo in input una posizione p nel piano, la relazione spaziale $\rightarrow$ in type-2 applicata a p restituisce le prime posizioni a destra di p; la relazione spaziale $\rightarrow_1$ in type-1 restituisce un insieme di posizioni a destra di p e nella stessa linea di p; la relazione spaziale $\rightarrow_0$ in type-0 restituisce un insieme di posizioni a destra di p anche se non sono nella stessa posizione di p. Nei nostri esempi utilizzeremo le relazioni spaziali HOR e VER che corrispondono alle relazioni spaziali $\rightarrow_0$ e $\downarrow_0$ - $\rightarrow_0$, rispettivamente, dove $\downarrow_0$ - $\rightarrow_0$ è un insieme che risulta essere la differenza tra l'insieme di posizioni generate da $\downarrow_0$ e $\rightarrow_0$ quando sono applicate alla stessa posizione.

4.2.2 Regole di valutazioni

Una regola di valutazione PE [3] è una funzione che prende in input una stringa del tipo

$$p_1 REL_1 p_2 REL_2 \dots REL_{m-1} \quad m \geq 1$$

dove ogni p_i è una posizione ogni REL_i è una relazione spaziale, il suo output è una **immagine** dove gli elementi p_1, p_2,..., p_n, sono disposti nello spazio in questo modo:

$$p_i + 1 \in REL_i(p_i) \quad 1 \leq i \leq m-1$$

Le regole di valutazione delle relazioni spaziali sono state pensate per essere una sequenza da sinistra a destra. Diciamo che una regola di valutazione è **semplice** se non presenta effetti collaterali.
Un esempio di applicazione di semplici regole di valutazione sono i seguenti esempi:

$$\begin{aligned} PE(a \rightarrow b \rightarrow c \rightarrow d) &= abcd \\ PE(a \quad VER \quad b \quad HOR \quad c) &= \begin{matrix} a \\ b \quad c \end{matrix} \end{aligned}$$

Derivazioni
Denotiano $\alpha \stackrel{*}{\Rightarrow} \beta$ (si legge α deriva in zero o più passi β) se esiste una stringa $\alpha_0 \alpha_1 \dots \alpha_m$ (m$\geq$0) tale che

$$\alpha \Rightarrow \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_m \quad = \quad \beta$$

La sequenza $\alpha_0\alpha_1 \ldots \alpha_m$ è chiamata **derivazione** di β da α. Una **forma sentenziale posizionale** è una stringa α tale che S $\stackrel{*}{\Rightarrow} \alpha$. Una **frase posizionale** è una forma sentenziale posizionale con soli simboli terminali. Una **forma pittorica** è la valutazione di una forma sentenziale posizionale. Un **immagine** è una forma pittorica che non contiene non-terminali. Il **linguaggio pittorico L(G)** definito da una grammatica posizionale G è l'insieme delle sue immagini. Un esempio di grammatica posizionale è mostrato di seguito.

$$\begin{gathered}\mathbf{N} = \{\text{E,T,F}\} \\ \mathbf{S} = \text{E} \\ \mathbf{T} = \{+,\ \mathit{hbar},\ (,\ ),\ \mathit{id}\} \\ \mathbf{POS} = \{\text{HOR, VER}\} \\ \mathbf{PE} \text{ non è una semplice regola di valutazione} \\ \mathbf{P} = \{\ \text{E} \rightarrow \text{E}\ \mathit{HOR}\ {+}\mathit{HOR}\ \text{T} \mid \text{T} \\ \text{T} \rightarrow \mathit{VER}\ \mathit{hbar}\ \mathit{VER}\ \text{F} \mid \text{F} \\ \text{F} \rightarrow (\mathit{HOR}\ \text{E}\ \mathit{HOR}) \mid \mathit{id}\ \}\end{gathered}$$

Una frase posizionale di questa grammatica è:

$$\begin{gathered}\mathit{id}\ \mathit{HOR}\ +\ \mathit{HOR}\ (\ \mathit{HOR}\ \mathit{id}\ \mathit{HOR}\ +\ \mathit{HOR}\ \mathit{id}\ \mathit{HOR})\ \mathit{VER}\ \mathit{hbar}\ \mathit{VER}\ \mathit{id} \\ \mathit{HOR}\ +\ \mathit{HOR}\ +\ \mathit{id}\end{gathered}$$

Rimpiazzando *hbar* con una linea orizzontale, e seguendo le definizioni di *HOR* e *VER*, da questa frase possiamo ottenere una delle possibili immagini:

$$\mathit{id}\ +\ \frac{(\mathit{id}+\mathit{id})}{\mathit{id}}\ +\mathit{id}$$

4.3 Definizione GLL Posizionale

In questo paragrafo descriveremo la gestione dell'input da parte del GLL parsing posizionale. Il funzionamento dell'algoritmo viene omesso in quanto la gestione dei terminali, non terminali e item della produzione avviene allo stesso modo delle grammatiche context-free

4.3.1 Gestione dell'input

L'input che viene dato in pasto al parser è un immagine simbolica e ogni simbolo contenuto in esso è un token. L'input viene rappresentato da un array X dove vengono memorizzati i token e da una matrice M che rappresenta l'input in base alla loro posizione spaziale e sono memorizzati con le posizioni *i-esime* dell'array X. Per segnalare la fine della stringa si aggiunge il simbolo \$ sia alla fine dell'array X e sia nella matrice M.

4.3.2 Gestione degli operatori spaziali

Per ogni relazione spaziale definiamo un operatore spaziale con lo stesso nome. Ogni qualvolta il parser lo trova viene chiamata la funzione *getNext-Token()* che prende in input l'indice dell'array X dell'ultimo token visto, l'operatore spaziale e un array Y che contiene i token già visti. Restituisce il token successivo, dopo aver consultato la matrice M. La ricerca sulla matrice avviene in questo modo:

- Se l'operatore spaziale è *HOR* ($>$) il calcolo del token successivo avviene in questo modo: si cerca un token non visto nelle colonne a destra della matrice M dell'ultimo token visto.
- Se invece l'operatore spaziale è *VER* ($<$) il calcolo del token successivo avviene cercando un token non ancora visto nella righe successive dell'ultimo token visto all'interno della matrice M.

La ricerca avviene usando l'array dei token visti e si restituisce il primo token che non è stato ancora visto.

Capitolo 5

Implementazione del GLL parsing

5.1 Introduzione

In questo capitolo illustreremo come è stato implementato il GLL Parsing. Parleremo delle componenti software che lo compongono ed illustremo come è stato utilizzato per un linguaggio lineare. Inoltre tratteremo anche della costruzione del SPPF.

5.2 Le componenti del sistema

Per realizzare il GLL parsing si è cercato di creare tutte le componenti usate dall'algoritmo che sono state accennate al capitolo 3. Queste componenti sono:

- **ElementoU**: è una componente che rappresenta un elemento memorizzato dall'insieme **U**,
- **ElementoP**: si occupa di gestire gli elementi dell'insieme **P**,
- **DescrittoreR**: questa componente rappresenta un descrittore che viene memorizzato dall'insieme **R**
- **GLLParsing**: è una componente che gestisce la computazione del GLL parsing.

Di seguito mostriamo il class diagram del sistema e nei paragrafi successivi tratteremo nei dettagli ogni componente del sistema.

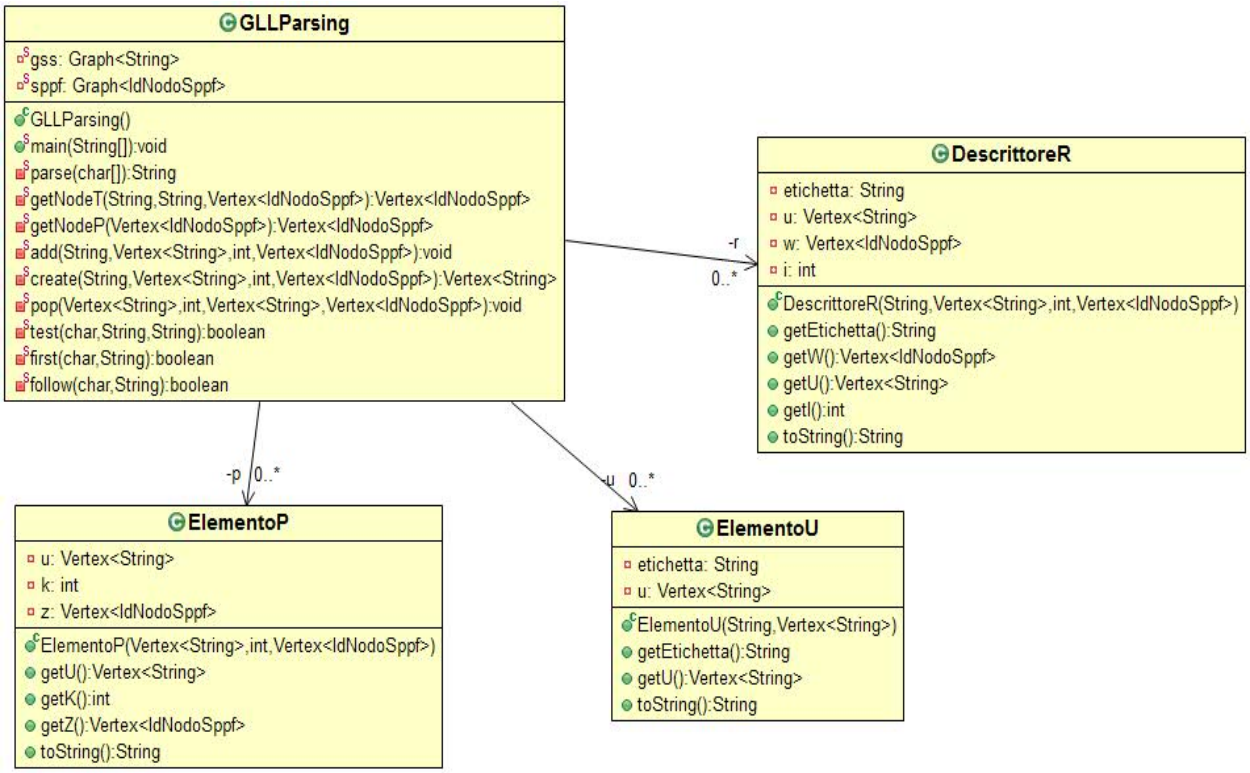

Figura 5.1: *Class diagram del software del GLL Parsing*

5.3 Le classi degli insiemi R, P e U

In questo paragrafo discutiamo delle classi che rappresentano gli elementi memorizzati dagli insiemi **P**, **R** e **U**. Qui di seguito presentiamo la classe ***ElementoU.java***

```
1 package gllparsinglineare;
2
3 import graph.*;
4
5 public class ElementoU {
6
7   private String etichetta;
8   private Vertex<String> u;
9   public ElementoU(String etichetta, Vertex<String> u) {
10     this.etichetta = etichetta;
11     this.u = u;
12  }
13
14  public String getEtichetta() {
```

```
15      return etichetta;
16   }
17
18   public Vertex<String> getU() {
19      return u;
20   }
21
22   public String toString(){
23      return "< "+ etichetta + " , " + u.element() + " >";
24   }
25 }
```

Questa classe definisce gli elementi appartenenti all'insieme **U**. Infatti questa classe ha come variabili d'istanza *etichetta*, di tipo *String*, che rappresenta l'identificativo usato per spostarsi sui vari item delle produzioni, ed un nodo *u*, di tipo *Vertex<String>*, che mantiene traccia del nodo del GSS che il parser sta processando. Presenta un costruttore per inizializzare le variabili d'istanza al momento della creazione dell'oggetto (linee 8-10), dei metodi d'accesso alle variabili d'istanza (linee 13-19) ed il metodo *toString()* per descrivere lo stato dell'oggetto (linee 21-22).
Ora descriviamo la classe ***ElementoP.java***

```
1 package gllparsinglineare;
2
3 import graph.*;
4
5 public class ElementoP {
6
7   private Vertex<String> u;
8   private int k;
9   private Vertex<IdNodoSppf> z;
10
11  public ElementoP(Vertex<String> u, int k, Vertex<IdNodoSppf>z) {
12     this.u = u;
13     this.k = k;
14     this.z = z;
15  }
16
17  public Vertex<String> getU() {
18     return u;
19  }
20
21  public int getK() {
22     return k;
23  }
24
25  public Vertex<IdNodoSppf>getZ(){
```

```
26      return z;
27   }
28
29   public String toString(){
30      return "< " + u.element() + " , " + k + " , " + z.element() + " >";
31   }
32 }
```

Questa classe rappresenta un elemento dell'insieme **P**. Ha come variabili d'istanza *u* di tipo *Vertex<String>*, che rappresenta un nodo del GSS, un intero *k*, che rappresenta la posizione di un simbolo all'interno dell'input, ed una variabile d'istanza *z*, di tipo *Vertex<IdNodoSppf>* che rappresenta un nodo del sppf (linee 7-9). Possiede un costruttore per inizializzare le varibili d'istanza al momento della creazione dell'oggetto (linee 11-15), dei metodi d'accesso alle variabili d'istanza (linee 17-27) ed il metodo *toString()* che serve per descrivere lo stato dell'oggetto (linee 35-37). La presenza del nodo del sppf è importante perchè ogni volta che viene fatto un *pop()* il parser deve riprendere la costruzione dell'sppf ripartendo dall'ultimo nodo inserito. Ora presentiamo la classe ***DescrittoreR.java***.

```
1 package gllparsinglineare;
2
3 import graph.*;
4
5 public class DescrittoreR {
6
7   private String etichetta;
8   private Vertex<String> u;
9   private Vertex<IdNodoSppf> w;
10  private int i;
11
12  public DescrittoreR(String etichetta, Vertex<String> u, int i,Vertex<
        IdNodoSppf> w) {
13     this.etichetta = etichetta;
14     this.u = u;
15     this.i = i;
16     this.w = w;
17  }
18
19  public String getEtichetta() {
20     return etichetta;
21  }
22
23  public Vertex<IdNodoSppf> getW() {
24     return w;
25  }
26
```

```
27    public Vertex<String> getU() {
28       return u;
29    }
30
31    public int getI() {
32       return i;
33    }
34
35    public String toString(){
36       return "<" + etichetta + " , " + u.element() + " , " +i + " , " + w.
             element() +">";
37    }
38
39 }
```

Questa classe rappresenta un descrittore che viene memorizzato nell'insieme **R**. Ha come varibili d'istanza un'*etichetta*, di tipo *String*, che inidica l'item che deve essere computato, un nodo u, che indica il nodo del GSS sul quale sta avvenendo la computazione, un intero i, che indica un simbolo della stringa in ingresso che si vuole trovare ed un nodo w, che indica un nodo del sppf su cui il parser deve continuare la costruzione (linee 7-10). Possiede un costruttore per inizializzare le variabili al momento della creazioe dell'oggetto (linee 12-17), dei metodi d'accesso alle variabili e il metodo *toString()* per descrivere lo stato di un oggetto

5.4 La classe GLL Parser

Inn questo paragrafo descriviamo la classe ***GLLParsing.java*** che definisce l'implementazione del GLL Parsing. Illustremo in più parti le varie operazioni che svolge. Il parsing viene eseguito sulla grammatica 3.1

```
1  package gllparsing;
2
3  import graph.*;
4  import java.io.*;
5  import java.util.*;
6
7  /*
8     * GRAMMATICA
9     * S->ASd
10    * S->BS
11    * S->epsilon
12    * A->a
13    * A->c
14    * B->a
```

```
15    * B->b
16  */
17  public class GLLParsing {
18    //gss
19    private static Graph<String> gss;
20    //insieme r e u che sono gli insiemi usati per registrare le scelte del
          non determinismo
21    private static ArrayList<ElementoU> u;
22    private static ArrayList<DescrittoreR>r;
23    //insieme p
24    private static ArrayList<ElementoP>p;
25    //sppf
26    private static Graph<IdNodoSppf> sppf;
27
28    public static void main(String []args) {
29       File f=new File("file.txt");
30       Scanner buffer;
31       if(f.exists()){
32          try{
33             FileReader in=new FileReader(f);
34             buffer=new Scanner(in);
35             char[] buf;
36             while(buffer.hasNextLine()){
37                String line=buffer.nextLine();
38                buf=line.toCharArray();
39                gss=new Graph<String>();
40                sppf=new Graph<IdNodoSppf>();
41                r=new ArrayList<DescrittoreR>();
42                u=new ArrayList<ElementoU>();
43                p=new ArrayList<ElementoP>();
44                String esito=parse(buf);
45                System.out.println(esito);
46             }
47          }
48          catch(Exception e){
49             e.printStackTrace();
50          }
51       }
52       else{
53          System.out.println("File not Found");
54       }
55    }
```

Iniziamo a commentare le parti sostanziali. Alle linee 19-26 vengono dichiarate le variabili d'istanza che sono: il GSS ed SPPF, di tipo *Graph*, che rappresentano il gss e l'sppf che viene costruito dal parser, l'insieme r un *ArrayList* che contiene gli elementi di tipo *DescrittoreR*; l'insieme u, un *ArrayList* che ha elementi di tipo *ElementoU* e l'insieme p, un *ArrayList* che

possiede gli elementi di tipo *ElementoP*. Poi viene definito il metodo *main()* (linee 28-55) che avvia il parser. L'input viene inserito in un file e ogni riga di questo file, che contiene una stringa su cui fare il parsing, viene inserita in array *buf*, di tipo *char*, vengono istanziate le variabili d'istanza e viene chiamato il metodo *parse*(), dove passiamo *buf*, per fare parsing sulla stringa. Alla fine viene stampato l'esito del parsing.
Ora analizziamo le funzioni fondamentali.

```
1  private static void add(String etichetta, Vertex<String> nu,int j,
       Vertex<IdNodoSppf>cn){
2     u.add(new ElementoU(etichetta,nu));
3     r.add(new DescrittoreR(etichetta,nu,j,cn));
4  }
```

Il metodo *add*() ha come parametri un etichetta, che inidca un item che deve essere computato, un intero *j*, che indica la posizione del simbolo della stringa che si vuole trovare, un nodo *nu* che inidca un nodo del GSS, ed un nodo cn, che inidica un nodo dell'sppf. Il metodo aggiunge gli elementi agli insiemi **R** ed **U** se questi elementi non appartengono ad **U**[*j*].

```
1  private static Vertex<String> create(String etichetta,Vertex<String> u,
       int j,Vertex<IdNodoSppf>cn){
2     //creazione del nodo
3     String nomeNodo="Ls"+j+etichetta;
4     Vertex<String> v=null;
5     Iterator<Vertex<String>> iteratorNodes=gss.vertices();
6     while(iteratorNodes.hasNext()){
7        Vertex<String> last=iteratorNodes.next();
8        if(nomeNodo.equals(last.element())){
9           v=last;
10       }
11    }
12    if(v==null){
13       v=gss.insertVertex(nomeNodo);
14    }
15    //controlliamo arco tra v ed u
16    Iterator<Edge<String>> eset=gss.edges();
17    boolean flag=true;
18    while(eset.hasNext()){
19       Edge<String> ed=eset.next();
20       Vertex<String> u1=ed.getStartVertex();
21       Vertex<String> u2=ed.getEndVertex();
22       if((u1.element().equals(v.element()))&&(u2.element().equals(u.
             element()))){
23          flag=false;
24       }
25    }
```

```
26      if((flag)&&(!(v.element().equals(u.element())))){
27         gss.insertDirectedEdge(v, u, "");
28         for(ElementoP elp:p){
29            if(elp.getU().element().equals(v.element())){
30               add(etichetta,u,elp.getK(),elp.getZ());
31            }
32         }
33      }
34      return v;
35   }
```

Il metodo create prende come parametri un *etichetta*, di tipo *String*, un nodo u, che rappresenta un nodo del GSS, un intero j che indica la posizione del simbolo della stringa in input che si vuole trovare ed un nodo cn che rappresenta un nodo dell'sppf. Alla linea 3 definiamo il nome del nuovo nodo del gss. La rappresentazione del nodo del L^j descritta al paragrafo 3.2 avviene in questo modo: $Lsjetichetta$. Si (linee 4-14) controlla se esiste un nodo v nel GSS $\boldsymbol{Lsjetichetta}$. Se non esiste lo aggiungiamo al GSS. Alle linee 16-34, si controlla se esiste un arco tra u e v nel GSS. Se non esiste lo aggiungiamo e di conseguenza per ogni nodo v presente in **P** chiama la funzione add(), passandogli *etichetta*, il nodo u, l'intero k di un elemento di **P** il nodo del sppf z dell'elemento **P**.

```
1   private static void pop(Vertex<String> u,int j,Vertex<String> u0,Vertex
        <IdNodoSppf>cn){
2      //if u diverso da u0
3      if(!(u.element().equals(u0.element()))){
4         //mettiamo elemento u,j a p
5         p.add(new ElementoP(u,j,cn));
6         Iterator<Edge<String>> eset=gss.edges();
7         //per ogni figlio v di aggiungi lu,v,j ad r e u
8         while(eset.hasNext()){
9            Edge<String> ed=eset.next();
10           Vertex<String>u1=ed.getStartVertex();
11           Vertex<String> v1=ed.getEndVertex();
12           if(u.element().equals(u1.element())){
13              add(u1.element().substring(3),v1,j,cn);
14           }
15        }
16     }
17  }
```

Il metodo *pop*() ha come parametri un nodo del GSS u, un intero j, che indica una posizione del simbolo della stringa in input da trovare, il nodo *u0*, che rapprsenta il nodo del GSS $ ed il nodo dell'sppf *cn*. Il metodo controlla se il nodo u è diverso da *u0*. Se lo è aggiunge u, j e cn nell'insieme **P**. Poi per

ogni figlio *v1* di u nel GSS chiamiamo il metodo $add()()$ e gli passiamo come parametri l'etichetta (che sarebbe una parte del nome del nodo u), il nodo v del GSS, l'intero j e il nodo cn del sppf.

```
1   //controlla il simbolo buffer corrente di un non terminale
2   private static boolean test(char x,String nonTerm,String handle){
3      if((first(x,handle))||(first('$',handle)&&(follow(x,nonTerm)))){
4         return true;
5      }
6      else{
7         return false;
8      }
9   }
10
11  private static Vertex<IdNodoSppf> getNodeT(String simbolo,String item,
        Vertex<IdNodoSppf>cn){
12     Iterator<Edge<IdNodoSppf>>it=sppf.edges();
13     String nonTerm=item.substring(0,1);
14     boolean flag=true;
15     while(it.hasNext()) {
16        Edge<IdNodoSppf>e=it.next();
17        Vertex<IdNodoSppf> v1=e.getStartVertex();
18        Vertex<IdNodoSppf> v2=e.getEndVertex();
19        if((cn.element().getId()==v1.element().getId())&&(item.equals(v2.
              element().getItem()))) {
20           flag=false;
21        }
22     }
23     if((flag)&&(cn.element().getNomeNodo().equals(nonTerm))) {
24        Vertex<IdNodoSppf> v=sppf.insertVertex(new IdNodoSppf(simbolo,item
              ));
25        v.element().setId(v.hashCode());
26        sppf.insertDirectedEdge(cn, v, null);
27        return v;
28     }
29     return cn;
30  }
31
32  private static Vertex<IdNodoSppf> getNodeP(Vertex<IdNodoSppf>cn){
33     Iterator<Edge<IdNodoSppf>>it=sppf.edges();
34     while(it.hasNext()) {
35        Edge<IdNodoSppf>e=it.next();
36        Vertex<IdNodoSppf> v1=e.getStartVertex();
37        Vertex<IdNodoSppf> v2=e.getEndVertex();
38        if(v2.element().getId()==cn.element().getId()) {
39           return v1;
40        }
41     }
42     return cn;
```

```
43  }
```

Infine abbiamo i metodi *test*(), *getNodeT*(), *getNodeP*(). Il metodo *test*() prende come parametri x, un simbolo della stringa in input, A, un non-terminale di tipo *String* ed *item*, di tipo *String*. Fa le stesse operazioni accennate al paragrafo 3.3.3. Il metodo *getNodeT*() prende come parametro *nomeNodo* che inidica il nome del nodo che deve avere il nuovo nodo del sppf e un nodo *cn*. Questo metodo inserisce l'arco tra il nodo *cn* e il nodo v di nome *nomeNodo*. Per identificare univocamente i nodi dell'sppf settiamo un id con l'*hashcode* dell'oggetto. Il metodo *getNodeP*() prende come parametro un intero i, un identificativo del nodo dell'sppf. Questo metodo restituisce il nodo padre *v1* che ha per figlio il nodo con l'identificativo i.
Ora analizziamo il metodo *parse*();

```
1   private static String parse(char []buf){
2     //dichirazione indici
3     int i=0;
4     //inizializzo etichette
5     String etichetta ="LS";
6     //creiamo i nodi del gss inserendo un arco tra u0 e u1
7     Vertex<String> u0=gss.insertVertex("$");
8     gss.insertVertex("Ls0L0");
9     gss.insertDirectedEdge(gss.getLastNode(),u0, "");
10    Vertex<String> cu=gss.getLastNode();
11    //creazione nodo sppf
12    Vertex<IdNodoSppf> cn=sppf.insertVertex(new IdNodoSppf("S","S"));
13    cn.element().setId(cn.hashCode());
14    while(true){
15      switch(etichetta){
16      //non terminale LL1
17      //S
18      case "LS":
19        if(test(buf[i],"S","ASd")){add("LS1",cu,i,cn);}
20        if(test(buf[i],"S","BS")){add("LS2",cu,i,cn);}
21        if(test(buf[i],"S","epsilon")) {add("LS3",cu,i,cn);}
22        etichetta="L0";
23        break;
24      //terminale LL1 A
25      case "LA":
26        if(test(buf[i],"A","a")){etichetta="La";}
27        if(test(buf[i],"A","c")){etichetta="Lc";}
28        break;
29      //terminale LL1 B
30      case "LB":
31        if(test(buf[i],"B","a")){etichetta="Lab";}
32        if(test(buf[i],"B","b")){etichetta="Lb";}
33        break;
```

```
34          //.ASd
35          case "LS1":
36             if(test(buf[i],"S","ASd")){
37                cu=create("L1",cu,i,cn);
38                cn=getNodeT("A","S->.ASd",cn);
39                etichetta="LA";
40             }
41             else{
42                etichetta="L0";
43             }
44             break;
45          //A.Sd
46          case "L1":
47             if(test(buf[i],"S","Sd")){
48                cu=create("L2",cu,i,cn);
49                cn=getNodeP(cn);
50                cn=getNodeT("S","S->A.Sd",cn);
51                etichetta="LS";
52             }
53             else{
54                etichetta="L0";
55             }
56             break;
57          //AS.d
58          case "L2":
59             if(buf[i]=='d'){
60                cn=getNodeP(cn);
61                cn=getNodeT("d","S->AS.d",cn);
62                i++;
63                etichetta="Ld";
64             }
65             else{
66                etichetta="L0";
67             }
68             break;
69          //ASd.
70          case "Ld":
71             cn=getNodeP(cn);pop(cu,i,u0,cn);etichetta="L0";
72             break;
73          //.BS
74          case "LS2":
75             if(test(buf[i],"S","BS")){
76                cu=create("L3",cu,i,cn);
77                cn=getNodeT("B","S->.BS",cn);
78                etichetta="LB";
79             }
80             else{
81                etichetta="L0";
82             }
```

```
83            break;
84          //B.S
85          case "L3":
86            if(test(buf[i],"S","S")){
87              cu=create("L4",cu,i,cn);
88              cn=getNodeP(cn);
89              cn=getNodeT("S","S->B.S",cn);
90              etichetta="LS";
91            }
92            else{
93              etichetta="L0";
94            }
95            break;
96          //BS.
97          case "L4":
98            cn=getNodeP(cn);pop(cu,i,u0,cn);etichetta="L0";
99            break;
100         //epsilon
101         case "LS3":
102           cn=getNodeT("e","S->e",cn);cn=getNodeP(cn);
103           pop(cu,i,u0,cn);
104           etichetta="L0";
105           break;
106         //.a
107         case "La":
108           if(buf[i]=='a'){
109             cn=getNodeT("a","A->a",cn);
110             i++;
111             etichetta="La1";
112           }
113           else{
114             etichetta="L0";
115           }
116           break;
117         //a.
118         case "La1":
119           cn=getNodeP(cn);pop(cu,i,u0,cn);etichetta="L0";
120           break;
121         //.a
122         case "Lab":
123           if(buf[i]=='a'){
124             cn=getNodeT("a","B->.a",cn);
125             i++;
126             etichetta="La1b";
127           }
128           else{
129             etichetta="L0";
130           }
131           break;
```

```
132         //a.
133         case "La1b":
134            cn=getNodeP(cn);pop(cu,i,u0,cn);etichetta="L0";
135            break;
136         //.b
137         case "Lb":
138            if(buf[i]=='b'){
139               cn=getNodeT("b","B->.b",cn);
140               i++;
141               etichetta="Lb1";
142            }
143            else{
144               etichetta="L0";
145            }
146            break;
147         //b.
148         case "Lb1":
149            cn=getNodeP(cn);pop(cu,i,u0,cn);etichetta="L0";
150            break;
151         //.c
152         case "Lc":
153            if(buf[i]=='c'){
154               cn=getNodeT("c","A->.c",cn);
155               i++;
156               etichetta="Lc1";
157            }
158            else{
159               etichetta="L0";
160            }
161            break;
162         //c.
163         case "Lc1":
164            cn=getNodeP(cn);pop(cu,i,u0,cn);etichetta="L0";
165            break;
166         case "L0":
167            if(r.size()>0){
168               etichetta=r.get(0).getEtichetta();
169               i=r.get(0).getI();
170               cu=r.get(0).getU();
171               cn=r.get(0).getW();
172               System.out.println(r.get(0));
173               r.remove(0);
174            }
175            else{
176               if(u.size()==0) {return "NON SUCCESSO";}
177               else{
178                  if((u.get(u.size()-1).getEtichetta().equals("L0"))&&(u.get
                        (u.size()-1).getU().element().equals(u0.element()))){
179                     return "SUCCESSO";
```

```
180                 }
181                 else{return "NON SUCCESSO";}
182               }
183             }
184             break;
185         }
186     }
187   }
```

Il metodo *parse*() prende come parametro un array *buf* che rappresenta la stringa in input su cui fare il parsing. Alle linee 3-13 vengono inizializzate le variabili usate dal parsing durante la computazione. Si inizializza l'indice i a 0 per posizionarlo sul primo elemento della stringa, inseriamo un arco tra i nodi \$ e *Ls0L0* nel GSS ed inseriamo il simbolo iniziale nell'sppf ed inizializza l'etichetta al simbolo iniziale. I **goto** presentati nell'algoritmo del paragrafo 3.4 sono implementati con uno ***switch*** all'interno di ciclo ***while*** infinito. I ***case*** dello ***switch*** assumono i possibili valori che può avere un etichetta. Un etichetta rappresenta un item o un non-terminale da sostituire. Le linee 18-29 abbiamo la gestione di un non-terminale non LL(1), infatti che abbiamo che se il metodo *test*() ha successo, viene chiamata il metodo *add*(), e ciò di applicare il non-determinismo usando l'insieme **R** ed **U**. Le linee 31-47 mostrano come viene trattato un non-terminale è LL(1). Infatti abbiamo che qui non viene applicato il non determinismo e di conseguenza questo ci permette direttamente di sostituire il non-terminale settando direttamente l'etichetta all'item opportuno. Ora analizziamo le linee 48-83. In queste linee abbiamo la gestione degli item per un corpo di una produzione. Alle linee 49-58 abbiamo che se il metodo *test*() ha successo facciamo varie operazioni. La prima operazione che facciamo è creare un nodo nel GSS con l'etichetta dell'item successivo (metodo *create*()), in questo modo quando verrà sostituito il simbolo puntato da questo item, sarà possibile continuare la computazione all'item successivo. Poi aggiungiamo un nodo al sppf etichettato con il corpo della produzione in questione (metodo *getNodeT*()). Infine settiamo l'etichetta con il valore che ci permette di sostituire il non-terminale dell'item che si sta analizzando. Se il metodo *test*() fallisce torniamo al caso base *L0*. Le linee 60-68 fanno le stesse operazioni però in questo caso non chiamano il metodo *getNodeT*() in quanto viene chiamato solo dal primo item del corpo della produzione. Nelle linee 70-78 si verifica se è stato trovato un simbolo della stringa in input. Se è vero incrementiamo l'indice i in maniera tale da farlo passare al simbolo successivo. Poi viene settata l'etichetta all'item successivo. Se la verifica fallisce viene settata l'etichetta al caso base *L0*. Alle linee 80-83 abbiamo completato un item, ossia il punto è alla fine del corpo della produzione, e di conseguenza viene chiamato

il metodo *pop*() e l'etichetta viene settata l'etichetta al caso base *L0*. Alle linee 111-116 viene gestita una produzione che come corpo della produzione ϵ. Viene chiamato il metodo *getNodeT()* per aggiungere un nodo all'sppf, poi viene chiamato il metodo *getNodeP*() per recuperare il nodo padre del nodo aggiunto precedentemente, chiamiamo il metodo *pop*() e settiamo l'etichetta al caso base *L0*. Osserviamo che il metodo *getNodeT*() va inserito solo quando gestiamo il primo item di un corpo di una produzione e il metodo *getNodeP*() va inserito solo nell'ultimo item il cui corpo di produzione è composto solo da un terminale. Ciò viene dimostrato nelle linee 152-167. Nel caso di produzione con ϵ vanno aggiunti entrambi, prima *getNodeT*() e poi *getNodeP*(). Alle linee 185-207 è presente la gestione del caso base *L0*. Si verifica se l'insieme **R** è pieno. Se lo è viene estratta il primo descrittore da **R** e vengono impostate le variabili *i*, *etichetta*, il nodo del GSS e il nodo dell'sppf per poterlo computare. Dopodichè viene rimosso da **R**. Se **R** non è vuoto si verifica se l'insieme **U** è vuoto. Se lo è il parsing non ha avuto successo e termina. Altrimenti se l'ultimo descrittore di **U** ha come etichetta *L0* e il nodo del GSS è \$ il parsing termina con successo altrimenti termina con un non successo.

Capitolo 6

Implementazione del GLL Posizionale

6.1 Introduzione

Nel capitolo seguente vedremo come è si implementa il GLL parsing sule grammatiche posizionali. Descriveremo come sono gestiti gli operatori spaziali e come viene letto l'input in corrispondenza dei simboli terminali. Per il resto la gestione dei non terminali e degli item delle varie produzioni sono state gestite allo stesso modo che vale per il GLL parsing lineare.

6.2 GLL Parsing su espressioni aritmetiche

In questo paragrafo discutiamo di come è stato implementato e gestito il GLL Parsing sulla grammmatica posizionale delle espressioni regolari. La grammatica è la seguente:

$$\begin{aligned} &E \rightarrow T > + > E \mid T \\ &T \rightarrow F < hbar < T \mid F \\ &F \rightarrow (> E >) \mid id \end{aligned} \tag{6.1}$$

Il simbolo $>$e $<$ sono operatori spaziali che indicano rispettivamente di spostarsi in orizzontale e in verticale per leggere l'input.

6.2.1 Gestione dell'input

La gestione dell'input è stato definito nella classe **InputDataset** presente nel file ***InputDataset.java***.

```
1 package dataset;
2
3 import java.io.*;
4 import java.util.*;
5
6 //classe che si occupa di gestire l'input
7 public class InputDataset {
8    private ArrayList<String> buf;
9    private String [][]picture;
10   private int row;
11   private int col;
12
13   public InputDataset() {
14      buf=new ArrayList<String>();
15   }
16
17   //caricamento picture e buffer
18   public void loadDataset(FileReader inMatrix) throws IOException{
19      Scanner lettore=new Scanner(inMatrix);
20      row=Integer.parseInt(lettore.nextLine());
21      col=Integer.parseInt(lettore.nextLine());
22      picture=new String[row][col];
23      int i=0,in=0;
24      while(lettore.hasNextLine()) {
25         String [] buf=lettore.nextLine().split(" ");
26         for(int j=0;j<buf.length;j++) {
27            this.buf.add(buf[j]);
28            picture[i][j]=""+in;
29            in++;
30         }
31         i++;
32      }
33      lettore.close();
34   }
35
36   //ottieni token
37   public String getToken(int i) {
38      return buf.get(i);
39   }
40
41   //metodo per settare un valore trovato
42   public void setTokenFound(int index,ArrayList<String> tokenViews) {
43      if(tokenViews.size()==0) {
44         tokenViews.add(""+index);
45      }
46      else {
47         boolean flag=true;
48         for(String t:tokenViews) {
```

```
49            if(t.equals(""+index)) {
50               flag=false;
51            }
52         }
53         if(flag) {
54            tokenViews.add(""+index);
55         }
56      }
57   }
58
59   //verifica se e' stato visto il token index
60   private boolean isViewed(String index,ArrayList<String> tokenViews) {
61      for(String s:tokenViews) {
62         if(s.equals(index)){
63            return false;
64         }
65      }
66      return true;
67   }
68
69   //metodo che calcola il simbolo da processare
70   public int getNextToken(int index,String dirSymbol,ArrayList<String>
         tokenViews){
71      int i,j,newCol=0,newRow=0;
72      for(i=0;i<row;i++){
73         for(j=0;j<col;j++){
74            if((picture[i][j]!=null)&&(picture[i][j].equals(""+index))){
75               switch(dirSymbol){
76               case "<":
77                  if(i==(row-1)) {newRow=0;}else {newRow=i+1;}
78                  newCol=0;
79                  break;
80               case ">":
81                  if(j==(col-1)) {newCol=0;}else {newCol=j+1;}
82                  newRow=0;
83                  if(picture[i][newCol]==null) {newCol=0;}
84                  break;
85               }
86               for(int k=newRow;k<row;k++) {
87                  for(int l=newCol;l<col;l++) {
88                     if((picture[k][l]!=null)&&(isViewed(picture[k][l],
                           tokenViews))) {
89                        return Integer.parseInt(picture[k][l]);
90                     }
91                  }
92               }
93            }
94         }
95      }
```

```
96      return -1;
97    }
98 }
```

La classe **InputDataset** ha come variabili d'istanza (linee 8-11) un ArrayList *buf*, che viene utilizzato per identificare univocamente i vari token e una matrice *picture* che viene utilizzata per rappresentare l'immagine di rappresentazione dell'input. Alle linee 18-34 è stato definito il metodo **loadData()** che viene utilizzato per caricare l'input presente in un file nella matrice *picture* e nell'array *buf*. L'input viene salvato anche in array in maniera tale da poter associare un identificativo ad ogni token. Nella matrice i token sono rappresentati attraverso gli identificativi che sono stati associati dall'array *buf*. Alle linee 37-39 il metodo **getToken()** restituisce il token in base all'identificativo ricevuto in input. Il metodo **setTokenFound()** (linee 42-57) viene utilizzato per inserire un token che è stato letto all'interno dell'array dei token visti. Infine abbiamo il metodo **getNextToken()** linee(75-98) che viene utilizzato per ottenere il token successivo. Questo metodo legge la matrice finchè non trova l'ultimo simbolo letto, una volta trovato seleziona le regole per leggere il token successivo (linee 77-84). Se il simbolo è $<$ allora il token successivo deve essere letto sulla riga successiva rispetto all'ultimo token visto, altrimenti se il simbolo è $>$ il token successivo viene letto sulla colonna di destra rispetto all'ultimo token visto. Nel caso in cui l'ultimo simbolo letto si trova sull'ultima riga o colonna della tabella il token successivo verrà scelto leggendo tutta la matrice. Dopodichè si legge la matrice partendo dagli indici calcolati dagli operatori spaziali (linee 86-92) e si restituisce il primo token non visto. Per stabilire se il token è stato visto o no si usa il metodo **isViewed()** (linee 60-67) che restituisce *false* se il token è stato visto o *true* se il token non è stato visto.

6.2.2 La classe GLLParsingPosizionale

Per implementare il GLL Parsing sulle espressioni aritmetiche è stata creata la classe GLLParsingPosizionale. Gli insiemi che il parsing utilizza sono sempre gli stessi ma presentano piccole differenze rispetto a quello lineare. Gli elementi dell'insieme **P** e i descrittori dell'insieme **R** presentano un ulteriore elemento ed è l'array dei token visti. Infatti l'esecuzione di ogni descrittore deve avere i propri token visti e di conseguenza ogni elemento **P** che viene richiamato dal nodo padre u del GSS che va ad aggiungersi all'insieme **R** devono avere il proprio array di token visti. Alle funzioni *add()* e *pop()* è stato aggiunto un parametro in input per richiedere l'inserimento dell'array dei token visti. Nel metodo *main* viene istanziato l'oggetto *inputDataset*,

di tipo InputDataset, e viene chiamato il metodo *loadDataset()* per caricare l'input nella matrice, poi viene chiamato il metodo *parse()* per iniziare il parsing ed a questo metodo viene passato come parametro *inputdataset*. Il metodo *parse()* è stato implementato seguendo le stesse regole usate per quello lineare ma con delle differenze che sono mostrate qui di seguito.

```
1 public static String parse(InputDataset ds) {
2   ArrayList<String> tokenViews=new ArrayList<String>();
3   //...
4   while(true){
5     switch(etichetta){
6     // (.>E>)
7     case "L90":
8       i = ds.getNextToken(i, ">", tokenViews);
9       etichetta = "L10";
10      break;
11    // F.<hbar<T
12    case "L6":
13      i = ds.getNextToken(i, "<", tokenViews);
14      etichetta = "L50";
15      break;
16    // F<.hbar<T
17    case "L50":
18      if (ds.getToken(i).equals("hbar")) {
19        ds.setTokenFound(i, tokenViews);
20
21        //...
22
23        etichetta = "L7";
24      }
25      else {
26        etichetta = "L0";
27      }
28      break;
29    }
30  }
31  //...
32 }
```

Le linee 1-10 mostrano come il modo in cui il parser gestisce gli operatori spaziali. Si può notare che vengono gestiti come item all'interno del corpo della produzione, e ogni volta che lo incontra viene chiamato il metodo *getNextToken()* che prende in input un operatore spaziale, l'utimo token letto e l'array dei token visti. Questo metodo restituisce il token successivo da leggere; dopodichè si passa all'item successivo. In corrispondenza della lettura di un input (linee 15-28) viene verificata l'uguaglianza del token che si sta leggendo, se è vera il token letto viene messo nell'array dei token visti.

6.3 GLL Parsing sui diagrammi di flusso

Ora illustreremo un altro esempio di implementazione del GLL Parsing su un'altra grammatica posizionale. La grammatica posizionale seguente definisce i diagrammi di flusso.

Program → *START link(1,1) Statements link(2,1) END*
Statements → *Statement link(2,1) Statements* {$$.*1*=$*1.1;* $$.*2*=$*2.2;*}
Statements → *Statement* {$$.*1*=$*1.1;* $$.*2*=$*1.2;*}
Statement → *INSTRUCTION* {$$.*1*=$*1.1;* $$.*2*=$*1.2;*}
Statement → *PREDICATE link(2,1)* ∧ *link(3,2) Statements* {$$.*1*=$*1.1;* $$.*2*=$*2.2;*}
Statement → *PREDICATE link(2,1)* ∧ *link(1,2) Statements* {$$.*1*=$*1.1;* $$.*2*=$*1.3;*}
Statement → *PREDICATE link(2,1)* ∧ *nolink(1,2) Statements link(3,1)(-1)* ∧ *link(2,2) Statements* {$$.*1*=$*1.1;* $$.*2*=$*2.2;*}

In questa grammatica gli operatori spaziali sono rappresentati dai *link(i,j)* dove $i, j \geq 0$. Essi indicano come leggere il token successivo e va interpretato in questo modo:

- Se *link(i,j)* si trova dopo un terminale il token successivo sarà il primo token non letto che ha in comune uno *j-esimo* collegamento con uno *i-esimo* collegamento con il token che è stato appena letto; il modo di leggere questo token è detto **Driver**.

- Se *link(i,j)* è preceduto dopo un non-terminale il token successivo sarà il primo token che ha in comune lo *i-esimo* collegamento di uno statement con il *j-esimo* collegamento di un token; il modo di leggere questo token è detto **Tester**.

- La seguente condizione *link(i,j)* ∧ *link(x,y)* va interpretato in questo modo: prima verifichiamo che esistano questi collegamenti e ciò lo facciamo attraverso l'operatore *and* ed usando il modo *Driver*, se è vera calcoliamo il token successivo usando solo *link(i,j)* secondo il modo *Driver*; poi, dopo aver calcolato un non-terminale all'interno di un item della produzione, usiamo *link(x,y)*, in modo *Tester*, per verificare se l'item appena calcolato restituisce il terminale presente all'interno della produzione. Ciò viene fatto per valutare la correttezza del parsing nel riconoscere le produzioni.

- L'operatore *nolink(i,j)* indica che non deve esistere nessun collegamento tra un token che ha il collegamento *i-esimo* e il token che ha il collegamento *j-esimo*.

- Il simbolo (-1) che si trova dopo *link(i,j)* vuol dire che il token successivo deve essere calcolato usando il token all'interno della produzione.

Le espressioni {$$.*1*=$*1.1;* $$.*2*=$*y.z;*} indicano come costruire gli **Statement** su cui andremo a calcolare i *link* in modo *Tester*. La dicitura $$.*1*=$*1.1* indica che il primo collegamento di statement corrisponde al primo collegamento del terminale appartenente alla produzione che si sta processando, mentre $$.*2*=$*y.z* può avere diverse interpretazioni:

- $$.*2*=$*2.2*, dove *y*=*z*; indica che il secondo collegamento dello statement corrisponde al secondo collegamento dell'ultimo statement creato;
- $$.*2*=$*1.z*, dove y=1; indica che il secondo collegamento dello statement corrisponde al collegamento z del terminale appartenente alla produzione che si sta processando.

6.3.1 Struttura e gestione dell'input

In questa sezione discuteremo la struttura degli statement, dei token e di come è stato implementata la gestione dell'input. Qui di seguito viene mostrata la classe ***Token.java***.

```
1 package dataset;
2
3 import java.util.ArrayList;
4
5 public class Token {
6    private boolean start;
7    private ArrayList<String> attachPoints;
8    private String type;
9
10   public Token(boolean start, ArrayList<String> attachPoints, String type
         ) {
11      this.start = start;
12      this.attachPoints = attachPoints;
13      this.type = type;
14   }
15
16   public Token(ArrayList<String> attachPoints, String type) {
17      start=false;
18      this.attachPoints = attachPoints;
19      this.type = type;
20   }
21
22   public boolean isStart() {
```

```
23      return start;
24   }
25
26   public ArrayList<String> getAttachPoints() {
27      return attachPoints;
28   }
29
30   public String getType() {
31      return type;
32   }
33
34   public String toString() {
35      if(start) {
36         return start + " " + type + " " + attachPoints;
37      }
38      else {
39         return type + " " + attachPoints;
40      }
41   }
42
43 }
```

La classe **Token** mostra la struttura di un token. Ha come variabili d'istanza (linee 6-8) una variabile booleana *start*, che serve ad indicare se è il primo token da processare; un ArrayList<String> *attachPoints* che contiene i collegamenti che un token ha con altri token, ed infine una variabile *type* di tipo String che indica il nome del token. Dalle linee 10-43 vengono dichiarati i costruttori delle classi, i metodi per accesso alle variabili e il metodo *toString()*.

Ora presentiamo la classe **Statement** descritta nel file ***Statement.java***

```
1 package dataset;
2
3 public class Statement {
4
5   private String type;
6   private String type2;
7   private String firstLink;
8   private String secondLink;
9
10
11   public Statement(String type, String firstLink, String type2, String
         secondLink) {
12      this.type = type;
13      this.firstLink = firstLink;
14      this.secondLink = secondLink;
15      this.type2 = type2;
```

```
16    }
17
18    public void setSecondLink(String type2,String secondLink) {
19       this.secondLink = secondLink;
20       this.type2 = type2;
21    }
22
23    public String getType1() {
24       return type;
25    }
26
27    public String getType2() {
28       return type2;
29    }
30
31    public String getFirstLink() {
32       return firstLink;
33    }
34
35    public String getSecondLink() {
36       return secondLink;
37    }
38
39    public String toString() {
40       return "< "+ type + ": " + firstLink + " ; " + type2 + ": " +
             secondLink + " >";
41    }
42 }
```

Questa classe viene utilizzata per definire la struttura degli statement. Ha come variabili d'istanza (linee 5-8): *type1* e *type2*, di tipo *String*, che serve ad indentificare rispettivamente a chi appartiene a quale token appartiene il primo e il secondo collegamento; poi abbiamo *firstLink* e *secondLink*, di tipo *String* che indica il primo e il secondo collegamento di uno statement. Dalle linee 11-42 abbiamo i costruttori della classe, i metodi di accesso alle variabili, il metodo per impostare il secondo collegamento allo statement, e il metodo *toString()*.

Ora presentiamo la classe **InputHandler** descritta all'interno del file ***InputHandler.java***.

```
1 package dataset;
2
3 import java.io.*;
4 import java.util.*;
5 import java.util.Map.Entry;
6 import com.google.gson.Gson;
7 import com.google.gson.internal.StringMap;
```

```
 8
 9 public class InputHandler {
10
11    private ArrayList<Token> buf;
12
13    public InputHandler() {
14       buf=new ArrayList<Token>();
15    }
16
17    public int getFirstToken() {
18       for(int i=0;i<buf.size();i++) {
19          Token t=buf.get(i);
20          if(t.isStart()) {
21             return i;
22          }
23       }
24       throw new IllegalArgumentException("ERRORE: Inserire start:true al
             token iniziale");
25    }
26
27    public Token getToken(int i) {
28       return buf.get(i);
29    }
30
31    public void loadInput(Gson gson,FileReader f1){
32       ArrayList<String> attacc = null;
33       boolean start=false;
34       String type = null;
35       Map<?, ?> map = gson.fromJson(f1, Map.class);
36       for (Map.Entry<?, ?> entry : map.entrySet()) {
37          ArrayList<StringMap> tokens=(ArrayList<StringMap>)entry.getValue()
               ;
38           for(StringMap<?> b: tokens) {
39             Set<?> set= b.entrySet();
40             Iterator<?> it=set.iterator();
41             while(it.hasNext()) {
42                Entry<String,?> en=(Entry<String, ?>) it.next();
43                switch(en.getValue().getClass().getName()) {
44                case "java.lang.Boolean":
45                   start=(boolean) en.getValue();
46                   break;
47                case "java.lang.String":
48                   type=(String) en.getValue();
49                   break;
50                case "java.util.ArrayList":
51                   attacc=(ArrayList<String>)en.getValue();
52                }
53             }
54             if(start) {
```

```
55              buf.add(new Token(start,attacc,type));
56              start=false;
57           }
58           else {
59              buf.add(new Token(attacc,type));
60           }
61         }
62      }
63   }
64
65   public int getTokenDriver(int pos, int attaccoSinistro, int
         attaccoDestro,ArrayList<Integer>tokenViews) {
66      ArrayList<String> attacchi=buf.get(pos).getAttachPoints();
67      for(int i=0;i<buf.size();i++) {
68         Token t1=buf.get(i);
69         ArrayList<String> attacchiEsterni=t1.getAttachPoints();
70         if(isViewed(i,tokenViews)) {
71            if(attacchiEsterni.size()>=attaccoDestro) {
72               if(attacchi.get(attaccoSinistro-1).equals(attacchiEsterni.
                     get(attaccoDestro-1))) {
73                  return i;
74               }
75            }
76         }
77      }
78      return -1;
79   }
80
81   //verifica se e' stato visto il token index
82   private boolean isViewed(int index,ArrayList<Integer> tokenViews) {
83      if(tokenViews==null) {
84         return true;
85      }
86      else {
87         for(Integer s:tokenViews) {
88            if(s==index){
89               return false;
90            }
91         }
92         return true;
93      }
94   }
95
96   public int getTokenTester(Statement s,int attaccoStatement,int
         attaccoToken,ArrayList<Integer> tokenViews) {
97      String at=null;
98      if(attaccoStatement==2) {
99         at=s.getSecondoAttacco();
100     }
```

```
101      if(attaccoStatement==1) {
102         at=s.getPrimoAttacco();
103      }
104      for(int i=0;i<buf.size();i++) {
105         Token ex=buf.get(i);
106         if(isViewed(i,tokenViews)) {
107            if(ex.getAttachPoints().size()>=attaccoToken){
108               if(ex.getAttachPoints().get(attaccoToken-1).equals(at)) {
109                  return i;
110               }
111            }
112         }
113      }
114      return -1;
115   }
116
117   //metodo per settare un valore trovato
118   public void setTokenFound(int index,ArrayList<Integer> tokenViews) {
119      boolean flag=true;
120      for(Integer t:tokenViews) {
121         if(t==index) {
122            flag=false;
123         }
124      }
125      if(flag) {
126         tokenViews.add(index);
127      }
128   }
129 }
```

La classe InputHandler si occupa di gestire l'input e di calcolare i token successivi. Ha come variabile d'istanza un'array *buf*, che rappresenta l'input da processare. Alle linee 17-29 vi sono i metodi *getFirstToken* e *getToken* che vengono utilizzati rispettivamente per ottenere il primo token da far processare al parser e per ottenere il token corrente che il parser deve calcolare. Il metodo *loadInput()* (linee 31-63) viene utilizzato per caricare l'input, contenuto all'interno di file di tipo json, all'interno dell'array buf. Infine (linee 65-129) sono stati implementati i metodi che permettono di calcolare i token successivi da processare. Il metodo *getTokenDriver()* viene utilizzato per leggere il token successivo secondo il modo *Driver*. Infatti ottiene i collegamenti del token appena letto e poi vengono iterati tutti i token dell'input e alla fine viene restituito l'indice del primo token non visto che ha in comune un collegamento con il token appena letto. Per verificare se il token è stato visto o meno viene utilizzato il metodo *isViewed()*. Il metodo *getTokenTester()* viene utilizzato per calcolare il token successivo secondo il modo *Tester*. Infatti

abbiamo che viene selezionato il collegamento dello statement che ci occorre per calcolare il token successivo, poi vengono iterati tutti i token dell'input e viene restituito il primo token non visto che il collegamento in comune con lo statement. Anche qui usiamo il metodo *isViewed()* per verificare se un token è stato visto o no. In alcuni casi abbiamo la necessità di non restituire il primo token non visto e per questo il metodo restituirà sempre *true* se l'array dei token visti è *null*. Infine il metodo *setTokenFound()* viene usato per aggiungere un token visto nel'array dei token visti.

6.3.2 La classe GLLParsingFlowChart

La classe GLLParsingFlowChart implementa il funzionamento del GLL parsing posizionale sulla grammatica dei diagrammi di flusso. Questo parser usa le stesse strutture dati del parsing posizionale descritto precedentemente ma con delle differenze: gli elementi dell'insieme **R** e **P** hanno un ulteriore elemento che è rappresentato dallo stack in cui sono memorizzati gli statement calcolati. Infatti ogni descrittore ed ogni elemento p che viene calcolato deve avere il proprio stack. Di conseguenza ai metodi *add()* e *pop()* è stato aggiunto un parametro che permette di passare lo stack degli statement. Di seguito descriveremo il metodo *parse()* di questa classe tralasciando i dettagli inerenti alla gestione dei non-terminali, terminali e item e sarà trattata soltanto la gestione del calcolo dei token successivi e degli statement.

```
1   public static String parse(InputHandler buf){
2     //token visti
3     ArrayList<Integer> tokenViews=new ArrayList<Integer>();
4     //dichirazione indici
5     int i=buf.getFirstToken();
6     //inizializzo link calcolati
7     Stack<Statement> s=new ArrayStack<Statement>();
8     while(true){
9       switch(etichetta){
10      //...
11      //Statement -> INSTRUCTION *{ $$.1 = $1.1; $$.2 = $1.2; }
12      case "L11":
13        s.push(new Statement(buf.getToken(i).getType(),buf.getToken(i).
              getAttachPoints().get(0),buf.getToken(i).getType(),buf.
              getToken(i).getAttachPoints().get(1)));
14        etichetta="L12";
15        break;
16      //...
17
18      //Statements *link(2,1) Statements { $$.1 = $1.1; $$.2 = $2.2; }
19      case "L5":
20        i=buf.getTokenTester(s.top(),2,1,tokenViews);
```

```
21            if(i>0) {
22               etichetta="L6";
23            }
24            else {
25               etichetta="L0";
26            }
27            break;
28         //..
29         //Statements link(2,1) Statements { $$.1 = $1.1; $$.2 = $2.2; }*
30         case "L8":
31            if(s.size()>1) {
32               Statement cv=s.pop();
33               s.top().setSecondoAttacco(cv.getType2(), cv.
                     getSecondoAttacco());
34            }
35            etichetta="L0";
36            break;
37         //...
38         //*PREDICATE link(2, 1) ^ nolink(1,2) Statements link(3,1)(-1) ^
               link(2,2) Statements { $$.1 = $1.1; $$.2 = $2.2; }
39         case "LSTAT3":
40            if(buf.getToken(i).getType().equals("PREDICATE")) {
41               buf.setTokenFound(i, tokenViews);
42               //...
43               etichetta="L17";
44            }
45            else {
46               etichetta="L0";
47            }
48            break;
49         //PREDICATE *link(2, 1) ^ nolink(1,2) Statements link(3,1)(-1) ^
               link(2,2) Statements { $$.1 = $1.1; $$.2 = $2.2; }
50         case "L17":
51         if((buf.getTokenDriver(i,2,1,tokenViews)>0)&&(!(buf.getTokenDriver
               (i, 1, 2,tokenViews)>0))) {
52            i=buf.getTokenDriver(i,2,1,null);
53            etichetta="L18";
54         }
55         else {
56            etichetta="L0";
57         }
58         break;
59         //...
60         case "L19":
61         i=buf.getTokenTester(s.top(),1,2,null);
62         if(i>0) {
63            if((buf.getToken(i).getType().equals("PREDICATE"))&&(!(buf.
                  getToken(i).getAttachPoints().get(0).equals(s.top().
                  getSecondoAttacco())))) {
```

```
64              i=buf.getTokenDriver(i,3,1,null);
65              etichetta="L55";
66           }
67           else {
68              s.pop();
69              etichetta="L0";
70           }
71        }
72        else {
73           s.pop();
74           etichetta="L0";
75        }
76        break;
77        case "L55":
78           if(i>0) {
79              add("L20",cu,i,cn,s.duplica(),duplicaTokenViews(tokenViews))
                    ;
80              add("L5",cu,i,cn,s.duplica(),duplicaTokenViews(tokenViews));
81           }
82           etichetta="L0";
83           break;
84        //...
85        //PREDICATE link(2, 1) ^ nolink(1,2) Statements link(3,1)(-1) ^
              link(2,2) Statements *{ $$.1 = $1.1; $$.2 = $2.2; }
86        case "L21":
87           Statement s1=s.pop();
88           Statement s2=s.pop();
89           i=buf.getTokenTester(s1, 1, 3,null);
90           if(i>0) {
91              if((s1.getSecondoAttacco().equals(s2.getSecondoAttacco()))
                     &&(buf.getToken(i).getType().equals("PREDICATE"))) {
92                 s.push(new Statement(buf.getToken(i).getType(),buf.
                       getToken(i).getAttachPoints().get(0),s1.getType2(),s1.
                       getSecondoAttacco()));
93                 etichetta="L22";
94              }
95              else {
96                 etichetta="L0";
97              }
98           }
99           else {
100             etichetta="L0";
101          }
102          break;
103          //...
104       }
105    }
106 //...
107 }
```

Alle linee 2-6 vengono dichiarati l'array dei token visti, viene inizializzato il primo token da calcolare e lo stack dove memorizzare gli statement. Alle linee 12-15 notiamo la costruzione di uno statement. Ciò avviene usando il primo e il secondo collegamento del token letto e viene inserito nello stack tramite un operazione di *push*. Nelle linee 18-36 abbiamo che la lettura del token successivo attraverso il modo *Driver*. In queste linee la costruzione dello statement avviene in maniera diversa: abbiamo che viene fatto un *pop()* sullo stack se ce ne sono più di uno, e viene impostato al secondo collegamento dello statement in cima allo stack il secondo collegamento dello statement appena estratto. In questo modo vengono rimossi tutti gli statement calcolati precedentemente in maniera tale che alla fine ne rimanga uno soltanto che indica l'inizio e la fine del diagramma di flusso. Alle linee 39-107 discutiamo la gestione degli item delle produzioni. Ne prendiamo di riferimento una soltanto in quanto le altre si gestiscono allo stesso modo. Viene letto il token corrente, se risulta trovato viene inserito nell'array dei token visti. Poi viene letto il token successivo facendo un *and* tra due link usando il modo Driver. Da notare in questo caso l'uso di *null* per segnalare che non è necessario trovare il primo token non visto. Se è vero calcoliamo il token successivo usando il primo *link* e il modo driver. Poi calcoliamo l'item successivo (parte omessa poichè si tratta di gestire un non-terminale e si fa usando le stesse regole di un non terminale). Successivamente verifichiamo se lo statement calcolato sia corretto e lo facciamo tramite il secondo *link* usato nell'and precedente e usando il modo Tester. Poi verifichiamo se la lettura del token successivo ha trovato un token, se ha successo, verifichiamo che il token letto dal link Tester sia quello all'interno della produzione e che siano diversi i collegamenti tra l'ultimo statement calcolato, che si trova in cima allo stack, e il primo collegamento del token della produzione. Se ciò è vero calcoliamo il *link(3,1)(-1)* tramite il modo Driver. Fondamentale risulta essere (-1) poichè ci indica di rileggere il token successivo dal token della produzione. Nel caso tutto ciò non fosse vero rimuoviamo lo statement calcolato dallo stack in quanto non risulta essere corretto a rappresentare la produzione che si sta calcolando. Poi notiamo (linee 78-83)che il parser si sdoppia nuovamente dove un parser va in avanti a calcolare lo statement successivo della produzione, mentre l'altro va a calcolare se esistono nuovi statement allo statement calcolato precedentemente. Dopo aver calcolato tutti gli statement calcolati dalla produzione gli estraiamo dallo stack e tramite un *link()* in modo Tester otteniamo il token della produzione. Verifichiamo prima se abbiamo trovato un un token, poi verifichiamo se il secondo collegamento del primo statement calcolato sia uguale al secondo collegamento del secondo statement calcolato ed infine verifichiamo che il token calcolato appartenga al token della produzione. Se è vero tramite un operazione di push inseriamo lo statement finale

della produzione e si va all'item successivo. Se tutto ciò non risulta vero si va al caso base L0. In maniera simile vengono gestite le altre produzioni ma avendo un solo statement da calcolare non effettuiamo nessun nuovo sdoppiamento ma semplicemente usiamo il secondo *link()* dell'and in modo Tester e poi si verifica che il token sia token della produzione. Ciò serve per verificare che lo statement calcolato si collega con il token della produzione. Se ciò è vero lo estraiamo dallo stack, tramite operazione di *pop()*, e inseriamo lo statement finale della produzione nello stack

Capitolo 7

Conclusioni

7.1 Obiettivi raggiunti

Gli obiettivi della prima parte della tesi sono stati quelli di introdurre il parsing top down, descrivendone il funzionamento del parsing LL(1) e delineandone i suoi limiti, poichè non riesce a gestire grammatiche ambigue e ricorsive. Per superare questi limiti abbiamo introdotto il parsing GLL che utilizza i principi del non determinismo per gestire tutte le grammatiche comprese quelle ambigue e ricorsive. Il suo funzionamento permette di gestire tutti i conflitti presenti nella tabella di parsing LL(1); ciò viene fatto creando dei nuovi flussi di computazione per ogni conflitto di sostituzione presente per un non-terminale nella tabella di parsing. Questo parsing usa una struttura dati, chiamata GSS, che combina i vari stack usati dai vari flussi di computazione. Il risultato prodotto da questo parsing è l'SPPF, un albero che combina in un unica struttura tutti gli alberi sintattici creati dai flussi di computazione del GLL. L'obiettivo finale di questo lavoro ha portato a creare un estensione del GLL Parsing per grammatiche posizionali. Viene utilizzato sempre lo stesso algoritmo per processare le grammatiche, però ne è stata modificata la gestione della lettura dei simboli successivi in quanto non avviene più in maniera lineare, dove i simboli vengono letti in successione da sinistra verso destra, ma avviene in base alle direzioni definite dalle relazioni spaziali che possono diversi per ogni grammatica posizionale.

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).