

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

1	Introduzione	1
2	Parsing top down	2
2.1	Introduzione	2
2.2	Grammatiche context-free	3
2.2.1	Definizione di grammatica	3
2.2.2	Convenzioni notazionali	4
2.2.3	Derivazioni	4
2.2.4	Alberi di parsing	6
2.2.5	Ambiguità	7
2.2.6	Grammatiche ricorsive	7
2.3	Parsing top down	8
2.3.1	Parsing a discesa ricorsiva	8
2.3.2	Funzioni FIRST e FOLLOW	9
2.3.3	Grammatiche LL(1)	10
2.3.4	Parsing predittivo non ricorsivo	11
2.4	Conclusioni	12
3	GLL Parsing	13
3.1	Introduzione	13
3.2	Stack e descrittori elementari	13
3.3	Definizione GLL Parsing	16
3.3.1	Graph structured stacks	17
3.3.2	Insiemi U e P	17
3.3.3	Funzioni fondamentali	17
3.3.4	Gestione degli item	18
3.3.5	Gestione delle sostituzioni	19
3.3.6	Shared packed parse forest	20
3.4	Costruzione del GLL Parser	21

4	GLL Parsing Posizionale	23
4.1	Introduzione	23
4.2	Definizione formale	23
4.2.1	Relazioni spaziali	24
4.2.2	Regole di valutazioni	25
4.3	Definizione GLL Posizionale	26
4.3.1	Gestione dell'input	26
4.3.2	Gestione degli operatori spaziali	26
5	Implementazione del GLL parsing	28
5.1	Introduzione	28
5.2	Le componenti del sistema	28
5.3	Le classi degli insiemi R, P e U	29
5.4	La classe GLL Parser	32
6	Implementazione del GLL Posizionale	43
6.1	Introduzione	43
6.2	GLL Parsing su espressioni aritmetiche	43
6.2.1	Gestione dell'input	43
6.2.2	La classe GLLParsingPosizionale	46
6.3	GLL Parsing sui diagrammi di flusso	48
6.3.1	Struttura dei simboli e gestione dell'input	49
6.3.2	La classe GLLParsingFlowChart	55
7	ParVis	60
7.1	Introduzione	60
7.2	Il prompt dei comandi	60
7.3	Le componenti grafiche del GLL Parsing	61
8	Conclusioni	67
8.1	Obiettivi raggiunti	67
	Bibliografia	68

Elenco delle figure

2.1	<i>Posizione del parser all'interno del compilatore.</i>	2
2.2	<i>Albero di parsing relativo alla stringa dcdcd</i>	6
2.3	<i>Sequenza di alberi di parsing relativi alla derivazione 2.3</i>	6
2.4	<i>Alberi di parsing relativi alla stringa dcdcd</i>	7
2.5	<i>Procedura di un non-terminale per un parser top down [1]</i>	8
3.1	<i>SPPF relativo alla stringa dcdcd</i>	20
5.1	<i>Class diagram del software del GLL Parsing</i>	29
7.1	<i>Prompt dei comandi</i>	61
7.2	<i>Insieme U</i>	62
7.3	<i>Testo in Input</i>	62
7.4	<i>Insieme P</i>	63
7.5	<i>GSS</i>	63
7.6	<i>SPPF</i>	64
7.7	<i>Stato Corrente</i>	64
7.8	<i>Informazioni sulla grammatica e sugli stati del parser</i>	65
7.9	<i>Insieme R</i>	65

Elenco delle tabelle

2.1	<i>Tabella di parsing della grammatica 2.4</i>	11
2.2	<i>Mosse di un parser predittivo sulla stringa cdd</i>	12
3.1	<i>Tabella di parsing della grammatica 3.1</i>	14

Capitolo 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 alle grammatiche context-free e posizionali e di come vengono visualizzate le operazioni del parser. Ciò verrà descritto nel quinto, sesto e settimo capitolo. La tesi si conclude all'ottavo 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 struttura corretta di un programma. Utilizza i token [1], ossia sequenze di caratteri restituite da un analizzatore lessicale (Lexer); per produrre una rappresentazione intermedia ad albero che rappresenta la struttura grammaticale dei token. Il risultato ottenuto dal parsing 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.

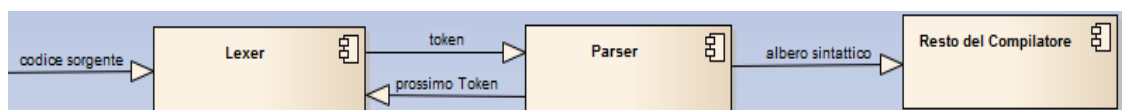


Figura 2.1: *Posizione del parser all'interno del compilatore.*

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. Sono usate per descrivere i costrutti dei linguaggi di programmazione. Ad esempio in C, il `while` può essere definito con la seguente forma:

$$\mathbf{while} \ (expression) \ statement$$

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:

$$stmt \rightarrow \mathbf{while} \ (expr) \ stmt \quad (2.1)$$

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 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$, con una testa comune A (che chiamiamo *A-produzioni*), possono essere scritte nel seguente modo: $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$. Chiamiamo $\alpha_1, \alpha_2, \dots, \alpha_k$ le *alternative per A*.

2.2.3 Derivazioni

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

$$E \rightarrow EcE \mid d \quad (2.2)$$

Un processo di derivazione, sulla stringa $dcdcd$ viene indicato con la seguente scrittura

$$E \Rightarrow EcE \Rightarrow dcE \Rightarrow dcEcE \Rightarrow dcdcE \Rightarrow dcdcd \quad (2.3)$$

che si legge " E deriva EcE ". La produzione $E \rightarrow EcE$ può essere utilizzata per sostituire qualsiasi non-terminale E con EcE per qualsiasi stringa di simboli della grammatica. La sequenza 2.3 viene definita come una *derivazione* della stringa $dcdcd$ a partire da E . Ora diamo una definizione formale di concetto di derivazione. Sia A un non-terminale e che sia posizionato in mezzo ad una sequenza di simboli grammaticali $\alpha A \beta$ dove α e β sono stringhe 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". Per esprimere che una stringa "deriva in zero o più passi" una nuova stringa utilizziamo il simbolo \Rightarrow^* . Quindi,

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

Inoltre il simbolo \Rightarrow^+ significa "deriva in uno o più passi".

Se $B \Rightarrow^+ \alpha$, dove B 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. 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 $dcdcd$ 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 *dcdcd* è mostrato nella figura 2.2.

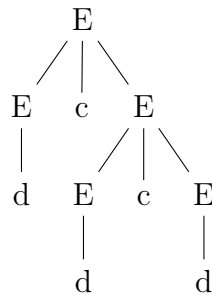


Figura 2.2: *Albero di parsing relativo alla stringa dcdcd*

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.

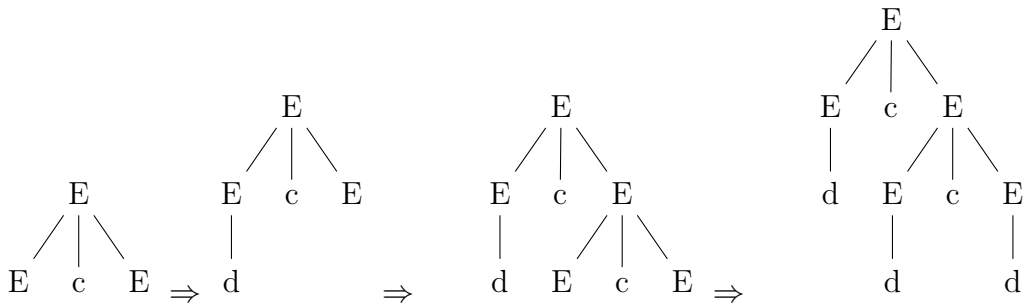


Figura 2.3: *Sequenza di alberi di parsing relativi alla derivazione 2.3*

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

d. Così facendo otteniamo all'ultimo passo il corrispondente albero sintattico per la stringa *dcdcd*.

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 *dcdcd*; questa frase presenta due alberi di parsing che sono:

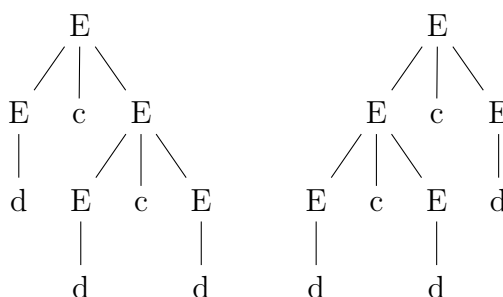


Figura 2.4: Alberi di parsing relativi alla stringa *dcdcd*

Di conseguenza risulta che la grammatica 2.2 risulta essere ambigua.

2.2.6 Grammatiche ricorsive

Una grammatica viene definita **ricorsiva** [1] se ha un non-terminale A per cui esiste una derivazione del tipo $A \xRightarrow{+} A\alpha$ o $A \xRightarrow{+} \alpha A$ 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:

$$stmt \rightarrow stmt; expr$$

Le grammatiche ricorsive risultano essere problematiche da gestire dai parser a discesa ricorsiva perchè entrano in un ciclo infinito. Supponiamo di voler applicare questa produzione; la prima cosa che facciamo è invocare la procedura *stmt()* che corrisponde al suo primo simbolo. Poichè il corpo della produzione inizia con *stmt* la procedura per *stmt* viene invocata ricorsivamente. Poichè il simbolo in input cambia solo quando si verifica una corrispondenza con un terminale del corpo della produzione, succederà che il parser continuerà a leggere sempre lo stesso simbolo. Di conseguenza la procedura *stmt()* viene chiamata sempre 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 simboli 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 il funzionamento del parsing top down. Verranno presentati i seguenti argomenti: parser a discesa ricorsiva, le funzioni FIRST e FOLLOW che vengono utilizzate per scegliere le produzioni da sostituire basandosi sul simbolo in input che si sta analizzando, le grammatiche LL(1) e i 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  $A \rightarrow X_1, X_2 \dots X_k$ ;
3)   for(i da 1 fino a k){
4)     if( $X_i$  è non-terminale){
5)       richiama la procedura  $X_i()$ ;
6)     }
7)   else{
8)     if( $X_i$  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 [1]*

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. In pratica, questa tecnica, viene utilizzata raramente per i costrutti dei linguaggi di programmazione e quindi risulta difficile trovare parser che la usano. Una grammatica ricorsiva 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 $\alpha \xRightarrow{*} \epsilon$, 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 A in qualche forma sentenziale, cioè l'insieme dei terminali a per cui esiste una derivazione nella forma $A \xRightarrow{*} \alpha A a \beta$, 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).

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 $\text{FOLLOW}(A)$ e $\text{FOLLOW}(S)$ coincide con il $\text{FOLLOW}(I)$.

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

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 lookahead. Data una grammatica G con due produzioni $A \rightarrow \alpha \mid \beta$ è definita LL(1) se sono verificate le seguenti condizioni:

- $\text{FIRST}(\alpha)$ e $\text{FIRST}(\beta)$ risultano essere insiemi disgiunti.
- Se ϵ appartiene a $\text{FIRST}(\beta)$ allora $\text{FIRST}(\alpha)$ e $\text{FOLLOW}(A)$ sono insiemi disgiunti; la stessa cosa vale se ϵ appartiene a $\text{FIRST}(\alpha)$.

In base a ciò, un parser predittivo per una grammatica LL(1) può essere sempre costruito se è possibile sostituire un non-terminale con una sola produzione che viene scelta in base al simbolo di input corrente. Ovviamente nessuna grammatica ambigua e ricorsiva può essere una grammatica LL(1). L'algoritmo seguente raccoglie le informazioni di FIRST e FOLLOW in una **tabella di parsing predittivo** $M[A, a]$, dove A è un non-terminale e a è un terminale. A volte può essere anche il marcatore di fine sequenza $\$$. L'algoritmo sceglie la produzione $A \rightarrow \alpha$ per un simbolo a solo se il simbolo appartiene a $\text{FIRST}(\alpha)$. Quando invece abbiamo a che fare con derivazioni $a \xRightarrow{*} \epsilon$, scegliamo sempre la produzione $A \rightarrow \alpha$ se il simbolo corrente appartiene al $\text{FOLLOW}(A)$, oppure se si è raggiunto il simbolo $\$$ nella stringa in input e se tale simbolo appartiene a $\text{FOLLOW}(A)$. Se dopo questi passi non vi è alcuna produzione per $M[A, a]$, 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 tabella di parsing.

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

Tabella 2.1: *Tabella di parsing della grammatica 2.4*

Si consideri, per esempio la produzione $I \rightarrow A$. Dato che $\text{FIRST}(A) = \text{FIRST}(S)$ questa produzione viene aggiunta sia $M[A, c]$ e a $M[A, d]$. Le caselle vuote in corrispondenza del simbolo $\$$ indicano gli errori.

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 $S \xRightarrow{*} w\alpha$. 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 descritta al paragrafo precedente. 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 ad una grammatica G ;
2. ad ogni passo di computazione si controlla un 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 un simbolo della stringa w e se esiste legge il simbolo successivo.
5. Ripetere le operazioni 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 .

Input	Stack	Azione	Riconosciuta
cdd \$	<i>I</i> \$		
cdd \$	<i>A</i> \$	output $I \rightarrow A$	
cdd \$	<i>S</i> \$	output $A \rightarrow S$	
cdd \$	<i>CC</i> \$	output $S \rightarrow CC$	
cdd \$	<i>cCC</i> \$	output $C \rightarrow cC$	
dd \$	<i>CC</i> \$	consuma c	c
dd \$	<i>dC</i> \$	output $C \rightarrow d$	c
d \$	<i>C</i> \$	consuma d	cd
d \$	<i>d</i> \$	output $C \rightarrow d$	cd
\$	\$	consuma d	cdd

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

Nella tabella 2.2 vengono riportate le mosse del parser predittivo applicate alla grammatica 2.4. 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. 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 ¹. 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 che il simbolo corrente della stringa in input appartiene a $FIRST(\alpha)$ o al $FOLLOW(\alpha)$. La stringa in input viene rappresentata come un array globale

¹Gli algoritmi e gli esempi presenti in questo paragrafo sono riconducibili alla bibliografia [2]

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

```

Questa è la tabella di parsing della grammatica 3.1.

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

Tabella 3.1: *Tabella di parsing della grammatica 3.1*

Da quello che si può notare dalla tabella 3.1 questa grammatica non è LL(1) in quanto è presente un conflitto e di conseguenza l'algoritmo implementato non funziona correttamente. Affinchè l'algoritmo funzioni correttamente è necessario 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. Per registrare le possibili scelte che il parser può fare per sostituire un non-terminale utilizziamo dei **descrittori** 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 \mathbf{R} . Questa azione viene denotata con la funzione $pop(s, i, \mathbf{R})$. Dopo aver fatto ciò rimuoviamo il descrittore (L', t, j) da \mathbf{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 \mathbf{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  $\mathbf{R}$ 
      if ( $I[i] \in \{a, b\}$ ) add ( $L_{S2}, s, i$ ) to  $\mathbf{R}$ 
      if ( $I[i] \in \{d, \$\}$ ) add ( $L_{S3}, s, i$ ) to  $\mathbf{R}$ 
 $L_0$ : if ( $\mathbf{R} \neq \emptyset$ ) { remove ( $L, s_1, j$ ) from  $\mathbf{R}$ 
      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 usando la stringa d'input $aad\$$. Incominciamo la nostra computazione aggiungendo prima la tripla $(L_{S1}, [L_0^0], 0)$, e poi la tripla $(L_{S2}, [L_0^0], 0)$ ad \mathbf{R} ed andiamo all'etichetta L_0 . Rimuoviamo $(L_{S1}, [L_0^0], 0)$ da \mathbf{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 \mathbf{R} e ritorna in L_0 . Allo stesso modo processiamo $(L_{S2}, [L_0^0], 0)$ da \mathbf{R} e alla fine avremo $(L_3, [L_0^0], 1)$ che viene aggiunto ad \mathbf{R} . Quindi \mathbf{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 \mathbf{R} . Allo stesso modo processiamo $(L_3, [L_0^0], 1)$ e in \mathbf{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 \mathbf{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 \mathbf{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. In questo paragrafo vedremo come correggere l'algoritmo e introdurremo le nuove strutture dati utilizzate dall'nuovo algoritmo.²

²Le funzioni che verranno descritte nei paragrafi successivi fanno riferimento alla bibliografia [2]

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 disgiunti 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 sinistra 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 formato 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 \mathbf{R} .

3.3.2 Insiemi U e P

Per stabilire l'esito del parsing è necessario usare l'insieme \mathbf{U} . Questo insieme contiene gli stessi descrittori dell'insieme \mathbf{R} . Questo insieme risulta necessario perchè non è possibile usare l'insieme \mathbf{R} , visto che vengono rimossi tutti i descrittori creati. 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 \mathbf{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 \mathbf{P} , se (L_u, w) non è presente in \mathbf{U}_k , allora (L_u, u, k) è aggiunto ad \mathbf{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 \mathbf{U}_j e se non c'è lo aggiunge a \mathbf{U}_j e ad \mathbf{R} . La funzione è definita nel seguente modo:

$$add(L, u, j) \{ \text{if}((L, u) \notin \mathbf{U}_j) \{ \text{add}(L, u) \text{ to } \mathbf{U}_j; \text{add}(L, u, j) \text{ to } \mathbf{R} \} \}$$

- **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:

$$pop(u, j) \{ \text{if}(u \neq u_0) \{ \text{add}(u, j);$$

$$\quad \text{for each child } v \text{ of } u \{ \text{add}(L_u, v, j); \} \} \}$$
- **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:

$$create(L, u, j) \{ \text{if there is not a GSS node labelled } L^j \text{ create one}$$

$$\quad \text{let } v \text{ be the GSS node labelled } L^j$$

$$\quad \text{if there is not an edge from } v \text{ to } u \{$$

$$\quad \quad \text{create an edge from } v \text{ to } u$$

$$\quad \quad \text{for all } ((v, k) \in \mathbf{P}) \{ add(L, u, k) \} \}$$

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

$$test(x, A, \alpha) \{ \text{if}(x \in \text{FIRST}(\alpha)) \text{ or } (\epsilon \in \text{FIRST}(\alpha) \text{ and } x \in \text{FOLLOW}(A)) \{$$

$$\quad \text{return true } \}$$

$$\text{else } \{ \text{return false } \} \}$$

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 verranno gestiti nel seguente modo:

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

$$code(a\alpha, j) = \mathbf{if}(I[j] = a) \{ j = j + 1 \} \mathbf{else} \{ \mathbf{goto} L_0 \}$$

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

$$code(A\alpha, j, X) = \mathbf{if}(test(I[j], X, A\alpha)) \{ \\ c_u = create(R_{A\alpha}, c_u, j); \mathbf{goto} L_A \} \\ \mathbf{else} \{ \mathbf{goto} L_0 \}$$

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

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

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

1. Se x_1 è un terminale:

$$code(A \rightarrow \beta, j) = \\ j = j + 1 \\ code(x_2 \dots x_n, j, A) \\ code(x_3 \dots x_n, j, A) \\ \dots \\ code(x_n, j, A) \\ pop(c_u, j); \mathbf{goto} L_0;$$

2. Se x_1 è un non-terminale:

$$code(A \rightarrow \beta, j) = \\ c_u = create(R_{A\beta}, c_u, j); \mathbf{goto} L_A \\ A_l: code(x_2 \dots x_n, j, A) \\ code(x_3 \dots x_n, j, A) \\ \dots \\ code(x_n, j, A) \\ pop(c_u, j); \mathbf{goto} L_0;$$

3.3.5 Gestione delle sostituzioni

Siano $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$ 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:

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

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

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

3.3.6 Shared packed parse forest

Gli alberi sintattici prodotti dai vari flussi di computazione, creati dall'applicazione del non determinismo, possono essere combinati in unica struttura chiamata **Shared packed parse forest** (SPPF). I nodi padri [4] sono uniti in nuovo nodo ed un nodo involucro diventa il nodo padre di ogni sottoalbero. La grammatica 2.2, che possiede due alberi di parsing, può essere rappresentata con un SPPF.

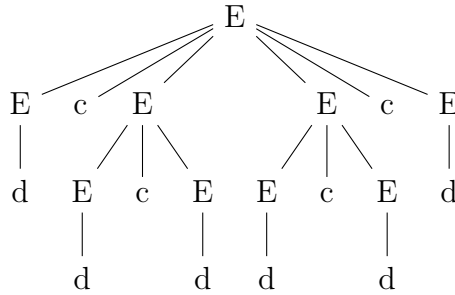


Figura 3.1: SPPF relativo alla stringa dcdcd

Il 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 $getNodeT(\alpha, c_n, ax_1 \dots x_n)$ viene usata per inserire all'interno del sppf un nodo etichettato α , che corrisponde ad un simbolo terminale e non, e $ax_1 \dots x_n ax_1 \dots x_n$, che equivale all'item a cui appartiene. Questo nodo creato viene collegato come nodo figlio al nodo c_n . Viene definita nel seguente modo:

$$getNodeT(\alpha, c_n, ax_1 \dots x_n) \{ \text{for each edge } E(c_p, c_c) \text{ in SPPF} \\ \text{if}((c_n = c_p) \text{ and } (ax_1 \dots x_n = \text{item of } c_c)) \{ \\ \text{create a new node } c_v = (c_n, ax_1 \dots x_n); \\ \text{create an edge from } c_n \text{ to } c_v \text{ in SPPF}; \} \}$$

- ***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:

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

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 ³ e nei capitoli successivi ne vedremo l'implementazione.

Sia Γ una grammatica e i suoi non-terminali sono A, \dots, 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

³Questo pseudocodice fa riferimento alla bibliografia [2]. È stata omessa la costruzione del sppf in quanto verrà discusso nei capitoli successivi.

c_u è un nodo del GSS

\mathbf{P} è un insieme che ha coppie formate da un nodo del GSS e di intero

\mathbf{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 $0 \leq j \leq m$ { $\mathbf{U}_j = \emptyset$ }

$\mathbf{R}=\emptyset$, $\mathbf{P}=\emptyset$

goto L_S ;

L_0 : **if**($\mathbf{R} \neq \emptyset$) {

remove a descriptor (L, u, j) from \mathbf{R}

$c_u=u$, $i=j$, **goto** L }

else if((L_0, u_0, j) $\in U_m$) { report success } **else** { report failure }

L_A : $code(A, i)$

...

L_X : $code(X, i)$

Capitolo 4

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 cosiddetti **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. Illusteremo la definizione base di questa nuova grammatica e il funzionamento 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:

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

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

Nelle grammatiche tradizionali abbiamo una sola relazione spaziale che è data dalla concatenazione della stringa, nelle grammatiche posizionali vengono usate più relazioni spaziali che danno le informazioni posizionali al 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

\rightarrow spostamento di un movimento a destra: $(x, y) \rightarrow (x+1, y)$

\leftarrow spostamento di un movimento a sinistra

\uparrow spostamento di un movimento in alto

\downarrow spostamento di un movimento in basso

Type-1

\rightarrow_1 spostamento a destra sulla stessa linea:

$$(x, y) \rightarrow \{\rightarrow^{(n)}(x, y) \mid n=1,2,\dots\}$$

\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

\rightarrow_0 spostamento a destra:

$$(x, y) \rightarrow \rightarrow_0(x, y) \cup \downarrow_1(x', y') \cup \rightarrow_1(x', y') \text{ dove } (x', y') \text{ cambia in } \rightarrow_0(x, y)$$

\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 posizione attuale. Ad esempio, 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 . 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 ed ogni REL_i è una relazione spaziale, il suo output è una **immagine** dove gli elementi p_1, p_2, \dots, 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.

Derivazioni

Denotiamo $\alpha \xRightarrow{*} \beta$ (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

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

La sequenza $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_m$ è chiamata **derivazione** di β da α . Una **forma sentenziale posizionale** è una stringa α tale che $S \xRightarrow{*} \alpha$. 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.

$$\begin{aligned} N &= \{E, A, B, C\} \\ S &= E \end{aligned}$$

$$\begin{aligned}
\mathbf{T} &= \{a, b, c\} \\
\mathbf{POS} &= \{HOR, VER\} \\
\mathbf{PE} &\text{ non è una semplice regola di valutazione} \\
\mathbf{P} &= \{ E \rightarrow A \text{ } VER \text{ } B \text{ } HOR \text{ } C \\
&\quad A \rightarrow a \\
&\quad B \rightarrow b \\
&\quad C \rightarrow d \}
\end{aligned}$$

La definizione delle regole di valutazione viene trascurata in quanto non è importante agli scopi di questa tesi.

Una frase posizionale di questa grammatica è:

$$a \text{ } VER \text{ } b \text{ } HOR \text{ } d$$

Da questa frase possiamo ottenere una delle possibili immagini:

$$\begin{array}{cc}
& a \\
b & c
\end{array}$$

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 tradizionali, così come descritto al capitolo 3.

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 simbolo terminale. 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 simbolo successivo avviene effettuando una ricerca nelle colonne a destra dell'ultimo token visto della matrice M .
- Se invece l'operatore spaziale è $VER (<)$ il calcolo del simbolo successivo avviene cercando nella righe successive dell'ultimo token visto all'interno della matrice M .

La ricerca avviene consultando l'array dei token visti e 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 implementare 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.

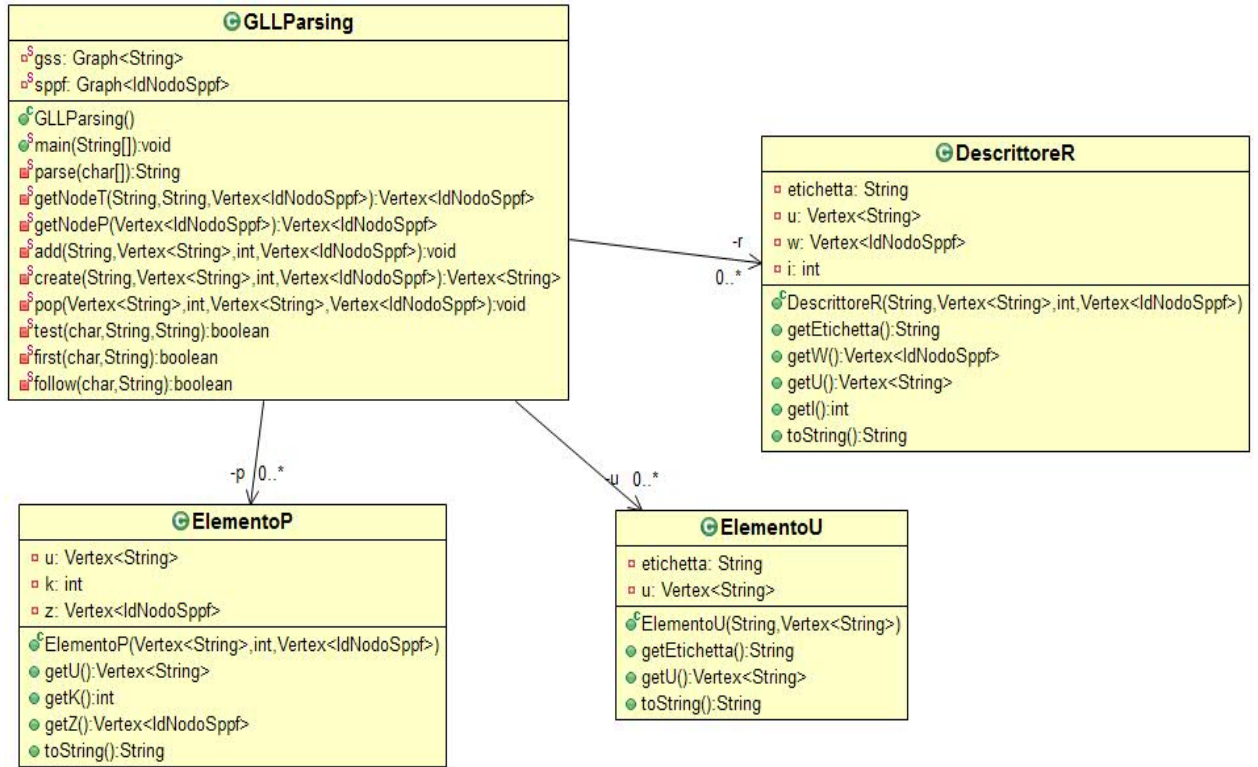


Figura 5.1: Class diagram del software del GLL Parsing

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*

```

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***

```

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 variabili 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<
13        IdNodoSppf> w) {
14        this.etichetta = etichetta;
15        this.u = u;
16        this.i = i;
17        this.w = w;
18    }
19
20    public String getEtichetta() {
21        return etichetta;
22    }
23
24    public Vertex<IdNodoSppf> getW() {
25        return w;
26    }
27 }

```

```

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 variabili d'istanza un'*etichetta*, di tipo *String*, che indica l'item che deve essere computato, un nodo *u*, che indica il nodo del GSS, 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 fare la costruzione (linee 7-10). Possiede un costruttore per inizializzare le variabili al momento della creazione 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* 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*; 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. La stringa da analizzare si trova

all'interno di un file e viene inserita nell'array *buf*, di tipo *char*. Vengono istanziate le variabili d'istanza e viene chiamato il metodo *parse()*, dove passiamo *buf*, per fare il parsing della 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 l'*etichetta*, l'identificativo di un item che deve essere computato, un intero *j*, la posizione del simbolo della stringa che si vuole trovare, un nodo *nu* che indica un nodo del GSS, ed un nodo *cn*, un nodo dell'SPPF. Il metodo aggiunge gli elementi agli insiemi **R** ed **U**.

```

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 L^j descritta al paragrafo 3.2 avviene usando questa notazione: *Lsjetichetta*. Si controlla (linee 4-14) se esiste un nodo *v* nel GSS che possieda il nome creato. 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* e il nodo dell'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 rappresenta 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 nodo *u* nel GSS chiamiamo il metodo *add()* e gli passiamo come parametri l'*etichetta* (che sarebbe parte *etichetta* del nome del nodo *u*), il nodo figlio *v* di *u*, 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 item){
3      if((first(x,item))||(first('$',item)&&(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,
12     Vertex<IdNodoSppf>cn){
13     Iterator<Edge<IdNodoSppf>>it=sppf.edges();
14     String nonTerm=item.substring(0,1);
15     boolean flag=true;
16     while(it.hasNext()) {
17         Edge<IdNodoSppf>e=it.next();
18         Vertex<IdNodoSppf> v1=e.getStartVertex();
19         Vertex<IdNodoSppf> v2=e.getEndVertex();
20         if((cn.element().getId()==v1.element().getId())&&(item.equals(v2.
21             element().getItem())) {
22             flag=false;
23         }
24     }
25     if((flag)&&(cn.element().getNomeNodo().equals(nonTerm))) {
26         Vertex<IdNodoSppf> v=sppf.insertVertex(new IdNodoSppf(simbolo,item
27             ));
28         v.element().setId(v.hashCode());
29         sppf.insertDirectedEdge(cn, v, null);
30         return v;
31     }
32     return cn;
33 }
34
35 private static Vertex<IdNodoSppf> getNodeP(Vertex<IdNodoSppf>cn){
36     Iterator<Edge<IdNodoSppf>>it=sppf.edges();
37     while(it.hasNext()) {
38         Edge<IdNodoSppf>e=it.next();
39         Vertex<IdNodoSppf> v1=e.getStartVertex();
40         Vertex<IdNodoSppf> v2=e.getEndVertex();
41         if(v2.element().getId()==cn.element().getId()) {
42             return v1;
43         }
44     }
45     return cn;
46 }

```

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 indica il nome del nodo da inserire nell' SPPF, il nodo cn , e un $item$. Questo metodo inserisce l'arco tra il nodo cn e il nodo v di nome *nomeNodo* appartenente alla produzione indicata dall' $item$. Per identificare univocamente i nodi dell'SPPF settiamo un identificativo con l'*hashCode* dell'oggetto. Il metodo *getNodeP()* prende come parametro un nodo cn dell'SPPF. Questo metodo restituisce il nodo padre u che ha per figlio il nodo cn .

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("LsOLO");
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     ControllerNode.setFirstInt(cn.element().getId());
15     while(true){
16         switch(etichetta){
17             //non terminale LL1
18             //S
19             case "LS":
20                 if(test(buf[i], "S", "ASd")){add("LS1",cu,i,cn);}
21                 if(test(buf[i], "S", "BS")){add("LS2",cu,i,cn);}
22                 if(test(buf[i], "S", "epsilon")) {add("LS3",cu,i,cn);}
23                 etichetta="L0";
24                 break;
25             //terminale LL1 A
26             case "LA":
27                 if(test(buf[i], "A", "a")){etichetta="La";}
28                 if(test(buf[i], "A", "c")){etichetta="Lc";}
29                 break;
30             //terminale LL1 B
31             case "LB":
32                 if(test(buf[i], "B", "a")){etichetta="Lab";}
33                 if(test(buf[i], "B", "b")){etichetta="Lb";}
34                 break;
35             //ASd

```

```

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

```

```

85         op.add(OperazioneLineare.creaStato(etichetta, "S->B*S"));
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="LO";}
94         break;
95     //BS.
96     case "L4":
97         cn=getNodeP(cn);
98         pop(cu, i, u0, cn);
99         etichetta="LO";
100        break;
101    //epsilon
102    case "LS3":
103        cn=getNodeT("e", "S->*e", cn);
104        cn=getNodeP(cn);
105        pop(cu, i, u0, cn);
106        etichetta="LO";
107        break;
108    //.a
109    case "La":
110        if(buf[i]=='a'){
111            cn=getNodeT("a", "A->*a", cn);
112            i++;etichetta="La1";}
113        else{
114            etichetta="LO";}
115        break;
116    //a.
117    case "La1":
118        cn=getNodeP(cn);
119        pop(cu, i, u0, cn);
120        etichetta="LO";
121        break;
122    //.a
123    case "Lab":
124        if(buf[i]=='a'){
125            cn=getNodeT("a", "B->*a", cn);
126            i++;etichetta="La1b";
127        }
128        else{
129            etichetta="LO";}
130        break;
131    //a.
132    case "La1b":
133        cn=getNodeP(cn);

```

```

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

```

```

183         (u.size()-1).getU().element().equals(u0.element()))){
184             return "SUCCESSO";
185         }
186         else{
187             return "NON SUCCESSO";
188         }
189     }
190     break;
191 }
192 }
193 }

```

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. I **goto** presentati nell'algoritmo del paragrafo 3.4 sono implementati con uno **switch** all'interno di un 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 gestire. Le linee 19-24 viene gestito un non terminale non LL(1) che presenta dei conflitti. Abbiamo che se il metodo *test()* ha successo, viene chiamato il metodo *add()*, e ciò permette applicare il non determinismo inserendo tutte le possibili sostituzioni nell'insieme **R** ed **U**. Alle linee 26-34 vengono gestiti i non-terminali LL(1), qui non viene applicato il non determinismo e di conseguenza questo ci permette di sostituire direttamente il non-terminale settando l'etichetta dell'item opportuno. Ora analizziamo le linee 35-72. In queste linee abbiamo la gestione degli item per un corpo di una produzione. Alle linee 36-56 abbiamo la gestione di un non-terminale all'interno dell'item. Viene invocato il metodo *test()*, se 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 non-terminale in cui si trova il punto nell'item e lo stesso item (metodo *getNodeT()*). Infine settiamo l'etichetta con il valore che ci permette di sostituire il non-terminale dell'item che si sta analizzando. Nel caso in cui il metodo *test()* fallisce torniamo al caso base *L0*. Le linee 47-56 fanno le stesse operazioni però prima di invocare il metodo *getNodeT()* viene invocato il metodo *getNodeP()* per richiamare l'ultimo nodo appartenente all'item che si sta processando. Nelle linee 58-66 si verifica se è stato trovato un simbolo della stringa in input. Se è vero incrementiamo l'indice *i* in ma-

niera tale da farlo passare al simbolo successivo. Poi viene settata l'etichetta all'item successivo. Se la verifica fallisce viene settata l'etichetta del caso base $L0$. Le linee 68-72 gestiscono il completamento di una produzione, ossia il punto dell'item è alla fine del corpo della produzione. Viene chiamato il metodo *getNodeP()*, il metodo *pop()* e l'etichetta viene settata l'etichetta al caso base $L0$. Alle linee 101-107 viene gestita una produzione che ha corpo 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, poi chiamiamo il metodo *pop()* e settiamo l'etichetta al caso base $L0$. Osserviamo che il metodo *getNodeP()* va inserito sempre dal secondo item della produzione che si sta processando. Nel caso in cui abbiamo una produzione con corpo ϵ vanno aggiunti entrambi, prima *getNodeT()* e poi *getNodeP()*. Alle linee 170-193 è presente la gestione del caso base $L0$. Si verifica se l'insieme \mathbf{R} è pieno. Se lo è viene estratto il primo descrittore da \mathbf{R} e vengono impostate le variabili i , *etichetta*, il nodo del GSS e il nodo dell'SPPF per calcolare una scelta applicata dal non determinismo. Dopodichè viene rimosso da \mathbf{R} . Nel caso in cui \mathbf{R} è vuoto, si verifica se l'insieme \mathbf{U} è vuoto. Se lo è il parsing non ha avuto successo e termina. Altrimenti se l'ultimo descrittore di \mathbf{U} ha come etichetta $L0$ e il nodo del GSS è \$ il parsing termina con successo, se non lo è, termina con insuccesso.

Capitolo 6

Implementazione del GLL Posizionale

6.1 Introduzione

Nel capitolo seguente vedremo come è stato implementato il GLL parsing sulle grammatiche posizionali. Descriveremo come sono state gestite le relazioni spaziali e come è stato applicato su due grammatiche posizionali. 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 grammatica posizionale delle espressioni aritmetiche. 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 relazioni spaziali che indicano rispettivamente di spostarsi in orizzontale e in verticale per leggere il simbolo successivo.

6.2.1 Gestione dell'input

La gestione dell'input è stato definito nella classe **InputDataset**.

```
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],
89                             tokenViews))) {
90                             return Integer.parseInt(picture[k][l]);
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 simboli 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*. Nella matrice i simboli sono rappresentati attraverso gli identificativi associati dall'array *buf*. Alle linee 37-39 il metodo **getToken()** restituisce il simbolo in base all'identificativo ricevuto in input. Il metodo **setTokenFound()** (linee 42-57) viene utilizzato per inserire un simbolo che è stato letto all'interno dell'array dei simboli visti. Infine abbiamo il metodo **getNextToken()** (linee 75-98) che viene utilizzato per ottenere il simbolo successivo. Questo metodo legge la matrice finché non trova l'ultimo simbolo letto, una volta trovato seleziona le regole per leggere il simbolo successivo (linee 77-84). Se il simbolo è $<$ allora il simbolo successivo deve essere letto sulle righe successive rispetto all'ultimo simbolo visto, altrimenti se il simbolo è $>$ il simbolo successivo viene letto sulle colonne a destra rispetto all'ultimo simbolo visto. Nel caso in cui l'ultimo simbolo letto si trova sull'ultima riga o colonna della tabella il simbolo successivo verrà scelto leggendo tutta la matrice. Dopodiché si legge la matrice partendo dagli indici calcolati dalle relazioni spaziali (linee 86-92) e si restituisce il primo simbolo non visto. Per stabilire se il simbolo è stato visto o no si usa il metodo **isViewed()** (linee 60-67) che restituisce *false* se il simbolo è stato visto o *true* se il simbolo 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 simboli visti e di conseguenza ogni elemento **P** che viene richiamato dal nodo padre *u* del GSS che va ad aggiungersi all'insieme **R** deve avere il proprio array dei simboli visti. Alle funzioni *add()* e *pop()* è stato aggiunto un parametro in input per richiedere l'inserimento dell'array dei simboli visti. Nel metodo *main()* viene istanziato l'oggetto *inputDataset*, di tipo `InputDataset`, viene chiamato il metodo *loadDataset()* per caricare

l'input nella matrice ed infine 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                etichetta = "L7";
23            }
24            else {
25                etichetta = "L0";
26            }
27            break;
28        }
29    }
30 }
31 //...
32 }
```

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

$$\begin{aligned}
 \text{Program} &\rightarrow \text{START link}(1,1) \text{ Statements link}(2,1) \text{ END} \\
 \text{Statements} &\rightarrow \text{Statement link}(2,1) \text{ Statements } \{\$.1=\$.1.1; \$.2=\$.2.2;\} \\
 &\quad \text{Statements} \rightarrow \text{Statement } \{\$.1=\$.1.1; \$.2=\$.1.2;\} \\
 \text{Statement} &\rightarrow \text{INSTRUCTION } \{\$.1=\$.1.1; \$.2=\$.1.2;\} \\
 \text{Statement} &\rightarrow \text{PREDICATE link}(2,1) \wedge \text{link}(3,2) \text{ Statements } \{\$.1=\$.1.1; \\
 &\quad \$.2=\$.2.2;\} \\
 \text{Statement} &\rightarrow \text{PREDICATE link}(2,1) \wedge \text{link}(1,2) \text{ Statements } \{\$.1=\$.1.1; \\
 &\quad \$.2=\$.1.3;\} \\
 \text{Statement} &\rightarrow \text{PREDICATE link}(2,1) \wedge \text{nolink}(1,2) \text{ Statements link}(3,1)(-1) \\
 &\quad \wedge \text{link}(2,2) \text{ Statements } \{\$.1=\$.1.1; \$.2=\$.2.2;\}
 \end{aligned}$$

In questa grammatica le relazioni spaziali sono rappresentati dai $\text{link}(i,j)$ dove $i, j \geq 0$. Essi indicano come leggere il simbolo successivo e va interpretato in questo modo:

- Se $\text{link}(i,j)$ si trova dopo un terminale il simbolo successivo sarà il primo simbolo non letto che ha in comune uno j -esimo collegamento con uno i -esimo collegamento con il simbolo che è stato appena letto; la lettura di questi simboli è detta **Driver**.
- Se $\text{link}(i,j)$ è preceduto da un non-terminale il simbolo successivo sarà il primo simbolo che ha in comune lo i -esimo collegamento di uno statement con il j -esimo collegamento di un simbolo; la lettura di questi simboli è detta **Tester**.
- La seguente condizione $\text{link}(i,j) \wedge \text{link}(x,y)$ va interpretato in questo modo: prima verifichiamo che esistano questi collegamenti e ciò lo facciamo attraverso l'operatore *and* ed usando il modo *Driver* per entrambi i link , se è vera calcoliamo il simbolo successivo usando solo $\text{link}(i,j)$ secondo il modo *Driver*; poi, dopo aver calcolato un non-terminale all'interno di un item della produzione, usiamo $\text{link}(x,y)$ 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 i vari simboli.
- L'operatore $\text{nolink}(i,j)$ indica che non deve esistere nessun collegamento tra un simbolo che ha il collegamento i -esimo e il simbolo che ha il collegamento j -esimo.

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

Le espressioni $\{\$.1=\$.1.1; \$.2=\$.y.z;\}$ indicano come costruire gli **Statement** utilizzati per leggere i simboli succ in modo *Tester*. La dicitura $\$.1=\$.1.1$ indica che il primo collegamento dello statement che si sta creando 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 che si sta creando corrisponde al secondo collegamento dell'ultimo statement creato;
- $\$.2=\$.1.z$, dove $y=1$; indica che il secondo collegamento dello statement che si sta creando corrisponde al collegamento z del terminale appartenente alla produzione che si sta processando.

6.3.1 Struttura dei simboli e gestione dell'input

In questa sezione discuteremo la struttura degli statement, dei simboli in input e di come è stata 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        ) {
12        this.start = start;
13        this.attachPoints = attachPoints;
14        this.type = type;
15    }
16
17    public Token(ArrayList<String> attachPoints, String type) {
18        start=false;
19        this.attachPoints = attachPoints;
20        this.type = type;
21    }

```

```

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 simbolo. Ha come variabili d'istanza (linee 6-8) una variabile booleana *start*, che serve ad indicare se il primo simbolo da processare; un `ArrayList<String>` *attachPoints* che contiene i collegamenti che un simbolo ha con altri simboli, ed infine una variabile *type* di tipo `String` che indica il nome del simbolo. Dalle linee 10-43 vengono dichiarati i costruttori delle classi, i metodi di 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 quale simbolo appartiene il primo e il secondo collegamento; poi abbiamo *firstLink* e *secondLink*, di tipo *String* che indicano 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             ;
39             for(StringMap<?> b: tokens) {
40                 Set<?> set= b.entrySet();
41                 Iterator<?> it=set.iterator();
42                 while(it.hasNext()) {
43                     Entry<String,?> en=(Entry<String, ?>) it.next();
44                     switch(en.getValue().getClass().getName()) {
45                         case "java.lang.Boolean":
46                             start=(boolean) en.getValue();
47                             break;
48                         case "java.lang.String":
49                             type=(String) en.getValue();
50                             break;
51                         case "java.util.ArrayList":
52                             attacc=(ArrayList<String>)en.getValue();
53                     }
```

```

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 simbolo da far processare al parser e per ottenere il simbolo corrente che il parser deve processare. 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 simbolo successivo secondo il modo *Driver*. Infatti ottiene i collegamenti dal simbolo appena letto e poi vengono iterati tutti i simboli dell'input e alla fine viene restituito l'indice del primo simbolo non visto che ha in comune un collegamento con il simbolo appena letto. Per verificare se il simbolo è stato visto o meno viene utilizzato il metodo *isViewed()*. Il metodo

getTokenTester() viene utilizzato per calcolare il simbolo successivo secondo il modo *Tester*. Infatti abbiamo che viene selezionato il collegamento dello statement che ci serve per calcolare il simbolo successivo, poi vengono iterati tutti i simboli dell'input e viene restituito il primo simbolo non visto che ha il collegamento in comune con lo statement. Anche qui usiamo il metodo *isViewed()* per verificare se un simbolo è stato visto o no. In alcuni casi abbiamo la necessità di non restituire il primo simbolo non visto e per questo motivo il metodo restituirà sempre *true* se l'array dei simboli visti è *null*. Infine il metodo *setTokenFound()* viene usato per aggiungere un simbolo visto nell'array dei simboli 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 simboli successivi e degli statement.

```

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

```

```

18      //...
19
20      //Statements *link(2,1) Statements { $$1 = $1.1; $$2 = $2.2; }
21      case "L5":
22          i=buf.getTokenTester(s.top(),2,1,tokenViews);
23          if(i>0) {
24              etichetta="L6";
25          }
26          else {
27              etichetta="L0";
28          }
29          break;
30      //..
31      //Statements link(2,1) Statements { $$1 = $1.1; $$2 = $2.2; }*
32      case "L8":
33          if(s.size()>1) {
34              Statement cv=s.pop();
35              s.top().setSecondoAttacco(cv.getType2(), cv.
36                  getSecondoAttacco());
37          }
38          etichetta="L0";
39          break;
40      //...
41      //PREDICATE link(2, 1) ^ nolinek(1,2) Statements link(3,1)(-1) ^
42      //link(2,2) Statements { $$1 = $1.1; $$2 = $2.2; }
43      case "LSTAT3":
44          if(buf.getToken(i).getType().equals("PREDICATE")) {
45              buf.setTokenFound(i, tokenViews);
46              //...
47              etichetta="L17";
48          }
49          else {
50              etichetta="L0";
51          }
52          break;
53      //PREDICATE *link(2, 1) ^ nolinek(1,2) Statements link(3,1)(-1) ^
54      //link(2,2) Statements { $$1 = $1.1; $$2 = $2.2; }
55      case "L17":
56          if((buf.getTokenDriver(i,2,1,tokenViews)>0)&&!(buf.getTokenDriver
57              (i, 1, 2,tokenViews)>0))) {
58              i=buf.getTokenDriver(i,2,1,null);
59              etichetta="L18";
60          }
61          else {
62              etichetta="L0";
63          }
64          break;
65      //...
66      case "L19":

```

```

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

```

```

105      //...
106    }
107  }
108  //...
109  }

```

Alle linee 2-6 viene dichiarato l'array dei simboli visti, viene calcolato il primo simbolo da calcolare e viene dichiarato 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 simbolo letto e viene inserito nello stack tramite un'operazione di *push*. Nelle linee 18-36 abbiamo che la lettura del simbolo successivo attraverso il modo *Tester*. 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 uno 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). 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 simbolo successivo ha avuto buon fine, se ha successo, verifichiamo che il simbolo calcolato 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 simbolo 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 simbolo successivo dal simbolo 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 li estraiamo dallo stack e tramite un *link()* in modo *Tester* otteniamo il simbolo della produzione. Verifichiamo prima se la ricerca del simbolo successivo ha avuto buon fine, poi verifichiamo se il secondo collegamento del primo statement calcolato sia uguale al secondo collegamento del secondo statement calcolato ed infine verifichiamo che il simbolo calcolato appartenga al simbolo 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 simbolo appartenga alla produzione. Ciò serve per verificare che lo statement calcolato sia corretto con la produzione che si sta processando. Se ciò è vero lo estraiamo dallo stack, tramite un operazione di *pop()*, e inseriamo lo statement finale nella produzione nello stack.

Capitolo 7

ParVis

7.1 Introduzione

In questo capitolo illustremo ParVis, un tool grafico che viene utilizzato per rappresentare graficamente le operazioni che vengono effettuate dai parser. Questo tool offre la possibilità di illustrare graficamente le strutture dati usate dal parser, l'albero sintattico, il testo in input su cui si sta facendo il parsing, il simbolo e lo stato corrente che si sta analizzando. Nei paragrafi successivi illustreremo le componenti basi utilizzate da ParVis, il suo funzionamento e le componenti create per rappresentare graficamente il GLL Parsing.

7.2 Il prompt dei comandi

Il **prompt dei comandi** è la componente fondamentale che viene usata per gestire le impostazioni grafiche di ParVis, l'esecuzione e il caricamento delle operazioni da illustrare. Per avviare l'esecuzione di un illustrazione grafica è necessario caricare il file *log.json*. Questo file viene generato dopo l'esecuzione del parser e contiene le operazioni di log effettuate. Nella figura 7.1 viene illustrato il prompt dei comandi. Presenta un menu "File" e "Windows" che sono utilizzati, rispettivamente, per caricare il file *log.json* e per settare le impostazioni grafiche delle finestre. Poi vi sono tre pulsanti dove i primi due, a partire da sinistra, sono utilizzati per eseguire un'operazione in avanti e all'indietro; mentre il terzo viene usato per automatizzare l'esecuzione delle operazioni. Nella parte principale del prompt dei comandi è presente una lista che contiene i vari item delle produzioni, chiamati **stati**. Questi stati sono raccolti in menu a tendina poichè contengono le varie operazioni effettuate per gestire un item. Possono essere aperti e chiusi cliccando o su

singolo stato o con i pulsanti in alto alla lista. Uno stato evidenziato indica che ParVis sta visualizzando le operazioni di quello stato, se lo si apre vedremo evidenziate le sue operazioni all'interno. In basso alla lista è presente uno slider che viene utilizzato per eseguire più velocemente le operazioni di visualizzazione.

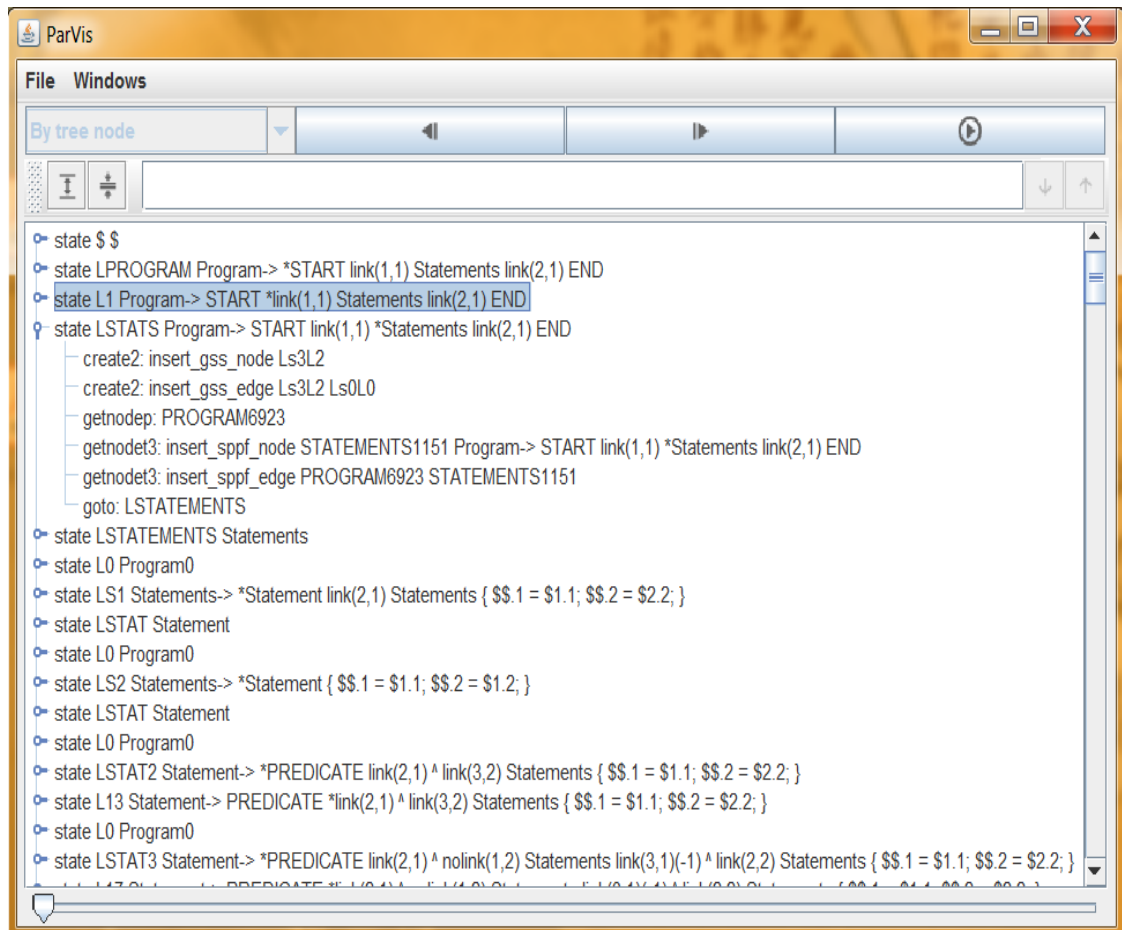


Figura 7.1: *Prompt dei comandi*

7.3 Le componenti grafiche del GLL Parsing

In questo paragrafo verranno illustrate le componenti grafiche realizzate per il GLL Parsing. Queste componenti sono state implementate nel file *json-reader.js* e ne sono state definite le impostazioni grafiche. Ogni finestra presenta un tooltip in cui vengono descritte le informazioni contenute da

ogni elemento grafico. Ogni finestra subisce continue modifiche per ogni operazione che si sta eseguendo.

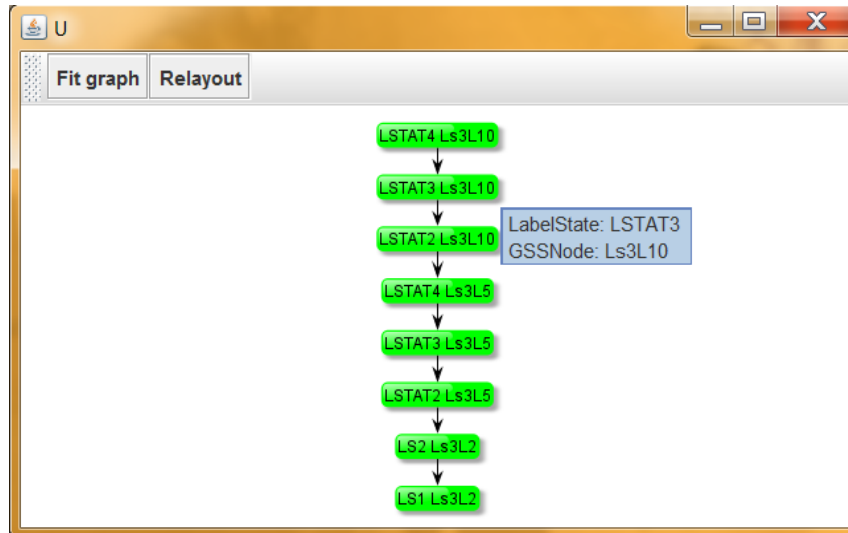


Figura 7.2: *Insieme U*

L'insieme U (fig. 7.2) viene rappresentato come una lista di nodi collegati uno dietro l'altro. Riporta le seguenti informazioni: il nome dello stato e il nome del nodo del GSS.

```

jsonTokens: [
  [ start: true
    AttachPoints: [a]
    type: START
  ],
  [ type: END
    AttachPoints: [L4]
  ],
  [ type: INSTRUCTION
    AttachPoints: [L3,L4]
  ],
  [ type: PREDICATE
    AttachPoints: [a,L2,L3]
  ],
  [ type: INSTRUCTION
    AttachPoints: [L2,L4]
  ],
]
  
```

Figura 7.3: *Testo in Input*

L'input (fig. 7.3) viene rappresentato allo stesso modo in cui è stato definito nel file d'input. Sull'input vengono utilizzati tre colori: il nero indica un

simbolo non ancora letto, il rosso indica un simbolo che il parser vuole trovare ed il blu indica un simbolo che è stato trovato dal parser.

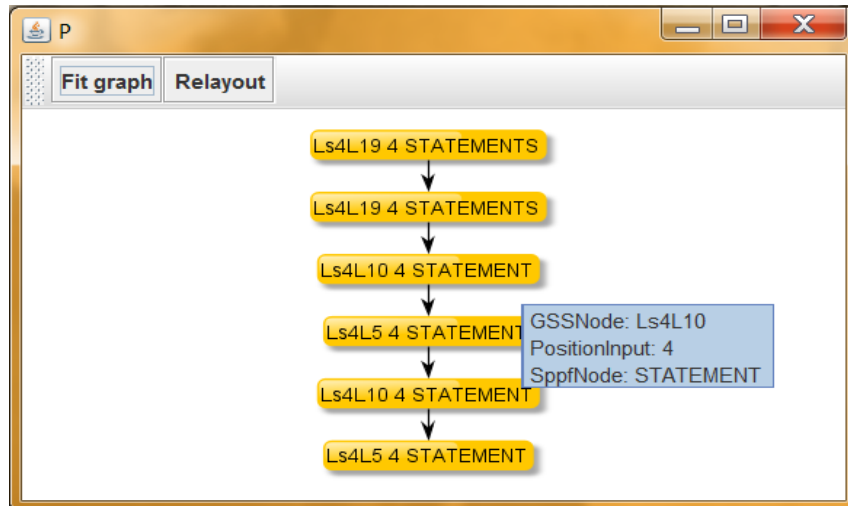


Figura 7.4: *Insieme P*

L'insieme P (fig. 7.4) è rappresentato come una lista di nodi. Sono state rappresentate le seguenti informazioni: il nodo del GSS, la posizione del simbolo da trovare e il nodo dell'SPPF.

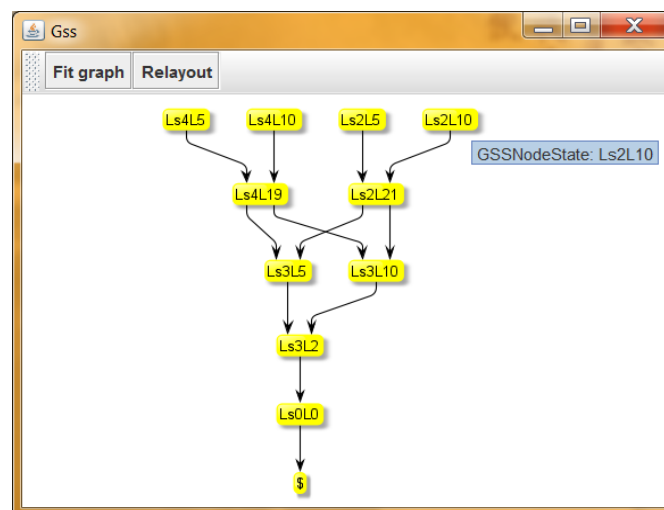


Figura 7.5: *GSS*

La figura 7.5 mostra la rappresentazione grafica del GSS. Viene rappresentato come un grafo diretto aciclico (DAG) che non contiene cicli. Ogni nodo contiene le informazioni sul simbolo che si vuole trovare e lo stato che identifica

l'item da processare. Il loro nome viene rappresentato in base alle notazioni descritte al capitolo 5.

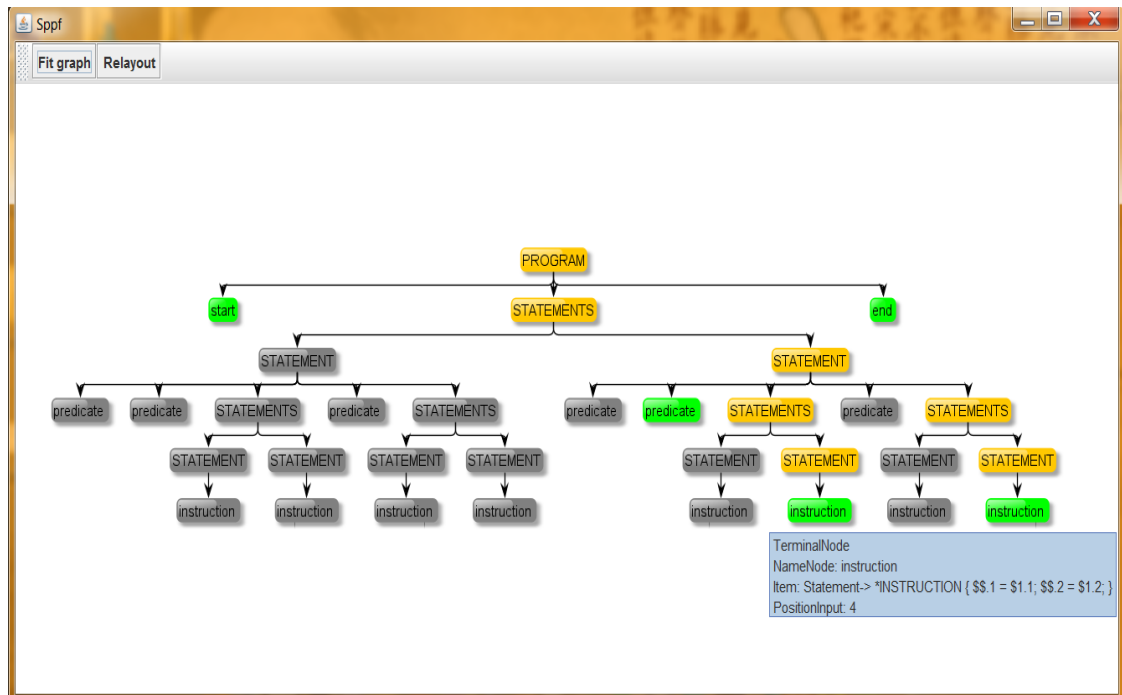


Figura 7.6: *SPPF*

L'SPPF (fig. 7.6) viene rappresentato con vari tipi di nodi: i nodi arancioni vengono usati per identificare i non-terminali, i nodi verdi vengono usati per i terminali, i nodi grigi rappresentano tutti quei nodi che appartengono a produzioni che non sono state completate per intero, nel tooltip vengono segnati come *Not Valid*. Le informazioni riportate sono: il tipo del nodo, la posizione che occupa nell'input (solo per i terminali), il nome del simbolo, l'item di produzione a cui appartiene.

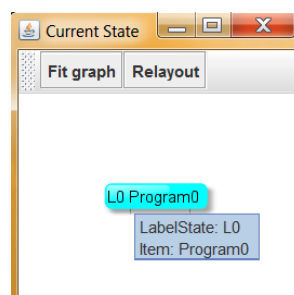
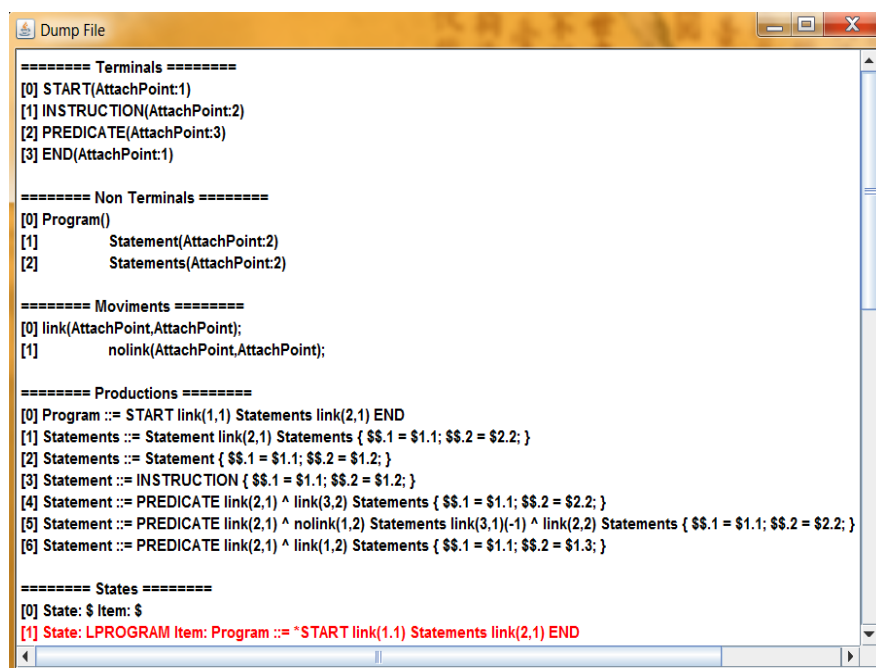


Figura 7.7: *Stato Corrente*

La finestra in figura 7.7 viene utilizzata per rappresentare lo stato corrente su cui il parser sta eseguendo le operazioni. È composta da un nodo che indica il nome dello stato e l'item che sta processando.



```

===== Terminals =====
[0] START(AttachPoint:1)
[1] INSTRUCTION(AttachPoint:2)
[2] PREDICATE(AttachPoint:3)
[3] END(AttachPoint:1)

===== Non Terminals =====
[0] Program()
[1] Statement(AttachPoint:2)
[2] Statements(AttachPoint:2)

===== Moviments =====
[0] link(AttachPoint,AttachPoint);
[1] nolink(AttachPoint,AttachPoint);

===== Productions =====
[0] Program ::= START link(1,1) Statements link(2,1) END
[1] Statements ::= Statement link(2,1) Statements { $$1 = $1.1; $$2 = $2.2; }
[2] Statements ::= Statement { $$1 = $1.1; $$2 = $1.2; }
[3] Statement ::= INSTRUCTION { $$1 = $1.1; $$2 = $1.2; }
[4] Statement ::= PREDICATE link(2,1) ^ link(3,2) Statements { $$1 = $1.1; $$2 = $2.2; }
[5] Statement ::= PREDICATE link(2,1) ^ nolink(1,2) Statements link(3,1)(-1) ^ link(2,2) Statements { $$1 = $1.1; $$2 = $2.2; }
[6] Statement ::= PREDICATE link(2,1) ^ link(1,2) Statements { $$1 = $1.1; $$2 = $1.3; }

===== States =====
[0] State: $ Item: $
[1] State: LPROGRAM Item: Program ::= *START link(1.1) Statements link(2.1) END

```

Figura 7.8: Informazioni sulla grammatica e sugli stati del parser

La finestra in figura 7.8 viene utilizzata per rappresentare le informazioni inerenti alla grammatica che implementa il parser. Vengono descritti i terminali, i non-terminali, le produzioni, il simbolo iniziale (che corrisponde sempre alla prima produzione) e i vari item (stati) che vengono gestiti dal parser. Ogni volta che il parser esegue le operazioni di uno stato viene colorato di rosso.

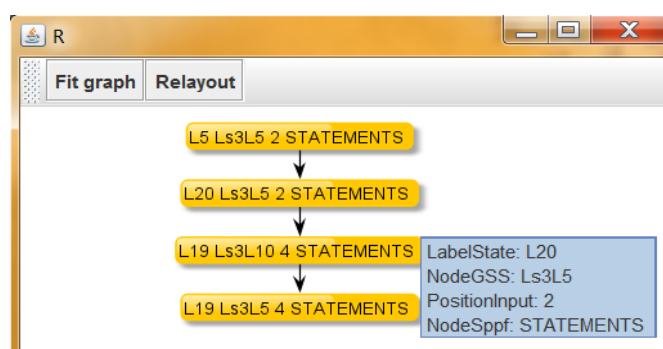


Figura 7.9: Insieme R

L'insieme R (fig. 7.9) viene rappresentato come una lista di nodi. Ogni nodo riporta le seguenti informazioni: il nome dello stato, la posizione del simbolo da trovare, il nodo del GSS e il nodo dell'SPPF.

Capitolo 8

Conclusioni

8.1 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, in quanto 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. 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 essere diverse 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 Theoretical 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. Economopoulos, *Generalized LR parsing algorithms*. Tesi di dottorato, Royal Holloway, University of London (2006).