

Università degli Studi di Salerno

Dipartimento di Informatica



Corso di Laurea Magistrale in Informatica

GLL Parsing su linguaggi non lineari

Relatore

Prof. Gennaro Costagliola

Candidato

Mazzotta Fabio

Anno Accademico 2018-2019

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

Indice

1	Introduzione	1
1.1	Obiettivi	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	Ricorsione a sinistra	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)	11
2.3.4	Parsing predittivo non ricorsivo	12
2.4	Conclusioni	13
3	GLL Parsing	15
3.1	Introduzione	15
3.2	Stack e descrittori elementari	15
3.3	Graph structured stacks	18
3.3.1	Insiemi U e P	19
3.4	Definizione GLL Parsing	19
3.4.1	Funzioni Fondamentali	19
3.4.2	Gestione degli item	20
3.4.3	Gestione delle produzioni	21
3.4.4	Shared packed parse forests	22
3.5	Costruzione del GLL Parser	23

4	Grammatiche Posizionali	25
4.1	Introduzione	25
4.2	Definizione formale	25
4.2.1	Relazioni spaziali	26
4.2.2	Regole di valutazioni	27
4.3	Parser Posizionali	28
4.3.1	Gestione dell'input	28
4.3.2	Gestione degli operatori spaziali	29
5	Implementazione del GLL parsing	30
5.1	Introduzione	30
5.2	Le componenti del sistema	30
5.3	Le classi degli insiemi R, P e U	31
5.4	La classe GLL Parser	34
6	GLL parsing posizionale	45
6.1	Introduzione	45
6.2	Gestione dell'input	45
6.3	La classe GLLParserBidimensionale	47
	Bibliografia	48

Elenco delle figure

2.1	<i>Posizione del parser all'interno del compilatore.</i>	2
2.2	<i>Albero di parsing relativo alla stringa id + id</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 id+id*id</i>	7
2.5	<i>Procedura di un non-terminale per un parser top down</i>	8
3.1	<i>Sppf relativo alla stringa id+id*id</i>	22
5.1	<i>Class diagram del software del GLL Parsing</i>	31

Elenco delle tabelle

2.1	<i>Tabella di parsing della grammatica 2.4</i>	12
2.2	<i>Mosse di un parser predittivo sulla stringa cdd</i>	13
3.1	<i>Tabella di parsing della grammatica 3.1</i>	16

Capitolo 1

Introduzione

1.1 Obiettivi

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

Capitolo 2

Parsing top down

2.1 Introduzione

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

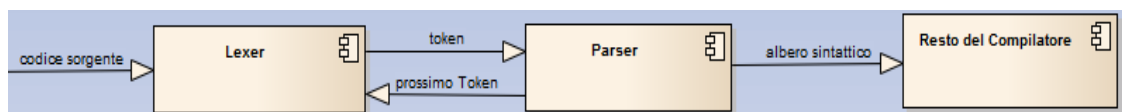


Figura 2.1: *Posizione del parser all'interno del compilatore.*

I metodi di parsing più comunemente utilizzate dai compilatori sono:

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

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

2.2 Grammatiche context-free

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

while (*espressione*) *statement*

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

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

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

2.2.1 Definizione di grammatica

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

1. **Terminali.** I terminali sono simboli di base con cui la grammatica definisce il linguaggio. Il termine "*token*" è un sinonimo di terminale.
2. **Non-Terminali.** I non-terminali sono variabili sintattiche che denotano un insieme di stringhe. Nella produzione 2.1 *stmt* e *expr* sono non-terminali. Gli insiemi di stringhe rappresentati dai non-terminali concorrono a definire il linguaggio generato dalla grammatica.
3. **Simbolo Iniziale.** In una grammatica uno dei non-terminali costituisce il simbolo iniziale e l'insieme di stringhe che esso denota coincide con l'intero linguaggio generato dalla grammatica.
4. **Produzione.** Le produzioni di una grammatica definiscono come i terminali e i non-terminali possono essere combinate a formare stringhe. Ogni produzione è formata da:

- (a) un non-terminale chiamato **testa**; la produzione definisce alcune delle stringhe denotate alla sua testa;
- (b) il simbolo \rightarrow ; a volte il simbolo $::=$ è utilizzato al posto della freccia;
- (c) un **corpo** o **lato destro** costituito da zero o più non-terminali o terminali; i componenti descrivono un modo in cui le stringhe denotate dal non-terminale della testa possono essere costruite.

2.2.2 Convenzioni notazionali

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

1. I seguenti simboli rappresentano i terminali:
 - (a) le singole lettere minuscole dell'alfabeto;
 - (b) i simboli degli operatori matematici e di punteggiatura;
 - (c) le stringhe minuscole in grassetto;
 - (d) le cifre numeriche.
2. I seguenti simboli sono non-terminali:
 - (a) le singole lettere maiuscole dell'alfabeto;
 - (b) se usate per descrivere i singoli costrutti della programmazione, le lettere maiuscole possono indicare i non-terminali del linguaggio.
3. La testa della prima produzione è il simbolo iniziale.
4. Un insieme di produzioni del tipo $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$, 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 della sua produzione. Tale visione *derivazionale* corrisponde al metodo di costruzione top-down degli alberi di parsing. Facciamo un esempio. Consideriamo la seguente grammatica:

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \text{id} \quad (2.2)$$

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

$$E \Rightarrow E + E \Rightarrow \mathbf{id} + E \Rightarrow \mathbf{id} + \mathbf{id} \quad (2.3)$$

che si legge " E deriva $E + E$ ". La produzione $E \rightarrow E + E$ può essere utilizzata per sostituire qualsiasi occorrenza di E con $E + E$ in una qualsiasi stringa di simboli della grammatica. La sequenza 2.3 viene definita come una derivazione della stringa $\mathbf{id} + \mathbf{id}$ a partire da E . Questa derivazione dimostra che la stringa $\mathbf{id} + \mathbf{id}$ è una particolare istanza di un'espressione. Ora diamo una definizione formale di concetto di derivazione. << Consideriamo un non-terminale A posizionata in mezzo ad una sequenza di simboli grammaticali $\alpha A \beta$ dove α e β sono stringhe arbitrarie di simboli grammaticali. Supponiamo che $A \rightarrow \gamma$ sia una produzione. In tal caso possiamo scrivere $\alpha A \beta \Rightarrow \alpha \gamma \beta$, in cui il simbolo \Rightarrow significa "deriva in un solo passo". Quando abbiamo una sequenza di passi di derivazione del tipo $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ in cui possiamo riscrivere α_1 come α_n diremo α_1 *deriva* α_n . Per esprimere che una stringa "deriva in zero o più passi" una nuova stringa utilizziamo il simbolo \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 $S \Rightarrow^+ \alpha$, dove S è il simbolo iniziale della grammatica G , diciamo che α è una **forma sentenziale** di G . Una forma sentenziale può contenere sia terminali che non terminali e può essere vuota. Una **sentenza** o **frase** di G è una forma sentenziale che non contiene nessun non-terminale. Il **linguaggio generato** da una grammatica G è l'insieme di tutte le sue frasi. Quindi una stringa di terminali w appartiene a $L(G)$, il linguaggio generato da G , se e solo se w è una frase di G , cioè se $S \Rightarrow^* w$. Un linguaggio che può essere generato da una grammatica è detto un **linguaggio libero dal contesto**. Se due grammatiche generano lo stesso linguaggio sono dette **equivalenti**. >> La stringa $\mathbf{id} + \mathbf{id}$ è una frase della grammatica 2.2 poichè esiste la derivazione 2.3. Le sequenze di derivazioni prevedono che ad ogni vengano fatte due scelte: la prima scelta consiste nello scegliere il non-terminale da sostituire; la seconda scelta consiste nello scegliere una delle produzioni in cui il non-terminale scelto risulta essere la testa della produzione. Infatti nella derivazione 2.3 ogni non-terminale è sostituito con il corpo della produzione corrispondente. Ogni non-terminale da sostituire viene selezionato in questo modo:

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

2.2.4 Alberi di parsing

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

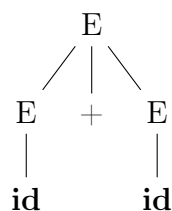


Figura 2.2: Albero di parsing relativo alla stringa **id + id**

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

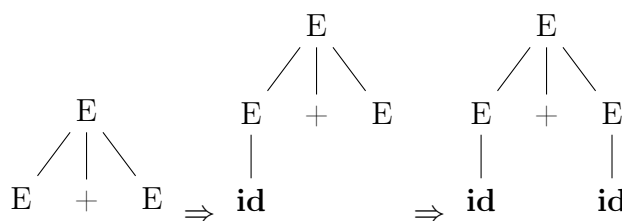


Figura 2.3: Sequenza di alberi di parsing relativi alla derivazione 2.3

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

nodo figlio **id**. Così facendo otteniamo al terzo passo il corrispondente albero sintattico per la stringa **id + id**.

2.2.5 Ambiguità

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

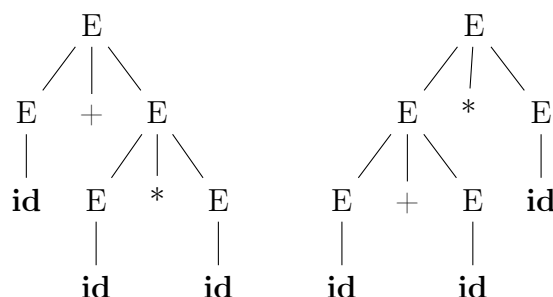


Figura 2.4: Alberi di parsing relativi alla stringa **id+id*id**

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

2.2.6 Ricorsione a sinistra

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

$$term \rightarrow term + fact$$

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

2.3 Parsing top down

Il parsing top down è una tecnica che prevede di costruire l'albero di parsing per una determinata stringa partendo dalla radice dell'albero fino ad arrivare alle foglie che rappresentano i 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 gli strumenti che usa il parsing top down. Verrà presentato il parsing a discesa ricorsiva che richiede *backtracking* per trovare la produzione opportuna da applicare al non-terminale. Successivamente introdurremo le funzioni FIRST e FOLLOW utilizzate per scegliere la produzione da applicare in base al simbolo in input che si sta analizzando. Poi parleremo delle grammatiche LL(1) ed infine dei parser predittivi che usano le funzioni FIRST e FOLLOW per scegliere le produzioni da sostituire.

2.3.1 Parsing a discesa ricorsiva

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

```

1) void A(){
2)   Scegli, per A, una produzione  $A \rightarrow X_1, X_2 \dots X_k$ ;
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*

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

2.3.2 Funzioni FIRST e FOLLOW

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

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

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

3. Se esiste una produzione $X \rightarrow \epsilon$, si aggiunga ϵ a $\text{FIRST}(X)$.

Per calcolare $\text{FOLLOW}(A)$ per tutti i non-terminali A si usano le seguenti regole:

1. Si aggiunga $\$$ a $\text{FOLLOW}(S)$, ricordando che S è il simbolo iniziale e $\$$ è il marcatore di fine della stringa d'ingresso;
2. Se esiste una produzione del tipo $A \rightarrow \alpha B \beta$, allora si aggiunga a $\text{FOLLOW}(B)$ ogni elemento di $\text{FIRST}(\beta)$ eccetto ϵ .
3. Se esiste una produzione del tipo $A \rightarrow \alpha B$ oppure del tipo $A \rightarrow \alpha B \beta$ per cui $\text{FIRST}(\beta)$ contiene ϵ , allora tutti i simboli in $\text{FOLLOW}(A)$ appartengono a $\text{FOLLOW}(B) \gg$

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

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

I FIRST e FOLLOW di questa grammatica sono:

1. $\text{FIRST}(I)=\text{FIRST}(A)=\text{FIRST}(S)=\text{FIRST}(C)=\{c,d\}$. 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 $\text{FIRST}(S)$ coincide con $\text{FIRST}(C)$. Lo stesso ragionamento lo si può applicare per $\text{FIRST}(S)$ e $\text{FIRST}(A)$
2. $\text{FOLLOW}(I)=\text{FOLLOW}(A)=\text{FOLLOW}(S)=\{\$\}$. Dato che I è il simbolo iniziale, il $\text{FOLLOW}(I)$ 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: \ll

1. non esiste alcun terminale a tale che sia α sia β derivano stringhe che iniziano con a ;
2. al più una tra α e β deriva la stringa nulla ϵ ;
3. se $\beta \xRightarrow{*} \epsilon$ non deriva alcuna stringa che inizia con un non terminale appartenente all'insieme $\text{FOLLOW}(A)$; allo stesso modo, se $\alpha \xRightarrow{*} \epsilon$, allora β non deriva alcuna stringa che inizia con un terminale in $\text{FOLLOW}(A)$. \gg

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

Algoritmo 1 Costruzione di una tabella di parsing predittivo.

INPUT Una grammatica G .

OUTPUT Una tabella di parsing M .

METODO Per ogni produzione $A \rightarrow \alpha$ della grammatica G si svolgono i seguenti passi.

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

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

Se dopo questi passi non vi è alcuna produzione in $M[A, a]$, si ha una condizione di errore, che viene indicata vuota nella casella corrispondente.

Facciamo un esempio e prendiamo in riferimento grammatica 2.4. Applichiamo l'algoritmo precedente ed otteniamo la seguente tabella di parsing.

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

Tabella 2.1: *Tabella di parsing della grammatica 2.4*

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

2.3.4 Parsing predittivo non ricorsivo

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

1. riceve in input una stringa w e una tabella di parsing M relativa a una grammatica G ;
2. ad ogni passo di computazione controlla il simbolo X in cima allo stack e un simbolo a della stringa w in input.

3. Se X è un non-terminale, il parser lo sostituisce con il corpo della produzione che si trova nella posizione $M[X, a]$;
4. Altrimenti, se X è un terminale, allora il parser verifica la corrispondenza di X con il simbolo a e se esiste legge il simbolo successivo della stringa w .
5. Ripetere il passo 2.
6. Il parser termina con successo se lo stack non contiene nessun simbolo X e ciò determina che la stringa letta fa parte della grammatica G .

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

Input	Stack	Azione	Riconosciuta
cdd \$	<i>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***

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

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

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

2.4 Conclusioni

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

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

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

Capitolo 3

GLL Parsing

3.1 Introduzione

Nel capitolo precedente abbiamo discusso i concetti e il funzionamento del parsing su grammatiche LL(1). In questo capitolo discuteremo di un 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 ambigue e ricorsive a sinistra. In questo capitolo vedremo come questo parser supera i limiti che hanno i parser LL(1) e mostreremo i concetti base di questo parsing e del suo funzionamento.

3.2 Stack e descrittori elementari

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

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

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

$main()\{$ $i = 0$

```

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

Quando l'algoritmo di parsing trova un simbolo 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 ed eseguiamo l'algoritmo con 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*, usato per leggere i simboli dell'input, e 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:

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

Processando di nuovo questi elementi otteniamo:

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

Da queste triple otteniamo:

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

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

3.3 Graph structured stacks

Nel paragrafo precedente abbiamo visto che ogni volta si trovava un non-terminale non LL(1) applicavamo il non-determinismo duplicando lo stack usato dal parser. Per rappresentare tutti questi stack in unica struttura dati utilizzeremo il **Graph structured stacks (GSS)**. Viene definito [4] come un grafo diretto aciclico (DAG) i cui nodi hanno etichette L^k che corrispondono agli elementi usati dallo stack, dove L indica un etichetta di una linea di codice e k la posizione su un simbolo della stringa in input. Questi nodi sono raggruppati in vari insiemi 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 formata da (L, u, i) , dove L è un'etichetta, u è un nodo del GSS e i che indica il simbolo della stringa in input $I[]$. Queste triple vengono aggiunte ad \mathbf{R} .

3.3.1 Insiemi \mathbf{U} e \mathbf{P}

Per costruire il GSS è necessario utilizzare un altro insieme \mathbf{U} che contiene gli stessi descrittori che inseriamo nell'insieme \mathbf{R} . Infatti abbiamo che $\mathbf{U}_i = \{ (L, u) \mid (L, u, i) \text{ è stato aggiunto ad } \mathbf{R} \}$. Un problema [2] che può sorgere sia ha quando un nodo figlio w è aggiunto ad u dopo aver eseguito le operazioni di pop sul GSS perchè l'azione di pop necessita di essere applicata a questo nodo figlio. Per risolvere ciò usiamo l'insieme \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 3.4.1

3.4 Definizione GLL Parsing

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

3.4.1 Funzioni Fondamentali

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

- **add()**: La funzione $add(L, u, j)$ controlla se c'è un descrittore (L, u) in \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:

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

for each child v of u { **add** (L_u, v, j); } }

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

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

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

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

3.4.2 Gestione degli item

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

$$\begin{aligned} C &\rightarrow .DKL \\ C &\rightarrow D.KL \\ C &\rightarrow DK.L \\ C &\rightarrow DKL. \end{aligned}$$

La produzione $C \rightarrow \epsilon$ genera l'item $C \rightarrow .$.

Un item indica una porzione di una produzione che si sta analizzando ad un certo punto del processo di parsing. Sia α la parte di produzione dopo il punto di un item. Nel GLL Parsing gli item 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:

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

3.4.3 Gestione delle produzioni

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:

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

3.4.4 Shared packed parse forests

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

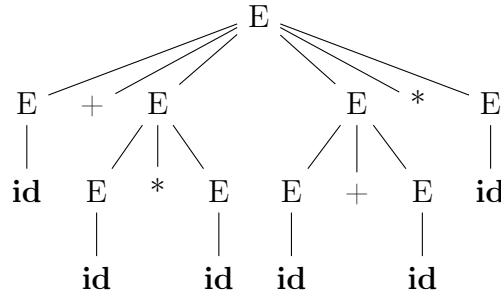


Figura 3.1: Sppf relativo alla stringa **id+id*id**

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

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

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

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

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

3.5 Costruzione del GLL Parser

Ora, avendo definito le varie funzioni e le varie operazioni da eseguire, siamo in grado di costruire in GLL Parser. Qui di seguito riportiamo lo pseudocodice del parser ¹ e nei capitoli successivi ne vedremo l'implementazione.

Sia Γ una grammatica [2] 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
 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

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

```

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

Grammatiche Posizionali

4.1 Introduzione

In questo capitolo introduciamo un'estensione delle grammatiche context-free, le *grammatiche posizionali* che producono i cosiddetti **linguaggi non lineari**. Queste grammatiche presentano produzioni contenenti *relazioni spaziali*, ossia relazioni che indicano come deve essere letto l'input, se in senso verticale, orizzontale, obliquo ecc... Illustreremo la definizione base di questa grammatica e il funzionamento degli operatori spaziali.

4.2 Definizione formale

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

1. **Non-Terminali (N)**: variabili sintattiche che denotano un insieme di stringhe.
2. **Terminali (T)**: simboli di base con la grammatica definisce 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 dell'input.

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

Ogni produzione presenta la seguente forma:

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

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

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

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. Infatti, se diamo in input una posizione p nel piano, la relazione spaziale \rightarrow in *type-2* applicata a p restituisce le prime posizioni a destra di p ; la relazione spaziale \rightarrow in *type-1* restituisce un insieme di posizioni a destra di p e nella stessa linea di p ; la relazione spaziale \rightarrow in *type-0* restituisce un insieme di posizioni a destra di p anche se non sono nella stessa posizione di p . 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:

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

\leftarrow_0 spostamento a sinistra

\uparrow_0 spostamento in alto

\downarrow_0 spostamento in basso

Nei nostri esempi utilizzeremo le relazioni spaziali HOR e VER che corrispondono alle relazioni spaziali \rightarrow_0 e $\downarrow_0 - \rightarrow_0$, rispettivamente, dove $\downarrow_0 - \rightarrow_0$ è un insieme che risulta essere la differenza tra l'insieme di posizioni generate da \downarrow_0 e \rightarrow_0 quando sono applicate alla stessa posizione.

4.2.2 Regole di valutazioni

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

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

dove ogni p_i è una posizione ogni REL_i è una relazione spaziale, il suo output è una **immagine** dove gli elementi p_1, p_2, \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.

Un esempio di applicazione di semplici regole di valutazione sono i seguenti:

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

Derivazioni

Denotiano $\alpha \xRightarrow{*} \beta$ (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 $\alpha_0 \alpha_1 \dots \alpha_m$ 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.

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

Una frase posizionale di questa grammatica è:

$$\begin{aligned}
&id \text{ HOR } + \text{HOR } (\text{HOR } id \text{ HOR } + \text{HOR } id \text{ HOR}) \text{ VER } hbar \text{ VER } id \\
&\quad \text{HOR } + \text{HOR } + id
\end{aligned}$$

Da questa immagine possiamo ottenere una delle possibili immagini:

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

4.3 Parser Posizionali

In questo paragrafo descriveremo una modalità usata per gestire le stringhe in ingresso e come fanno i parser a gestire gli operatori spaziali delle grammatiche posizionali.

4.3.1 Gestione dell'input

L'input che viene dato in pasto al parser è un'immagine simbolica dove ogni simbolo che contiene è un token. Questo input viene rappresentato da un array X dove vengono memorizzati i token e da una matrice M dove i token sono inseriti in base alla loro posizione spaziale e sono memorizzati con le posizioni *i-esime* dell'array X . Ciò permette di individuare ogni volta un token in X data la loro posizione spaziale. In questo modo abbiamo che l'input non risulta essere una concatenazione di simboli. 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 una funzione *getNextToken()* che prende in input l'indice dell'array X dell'ultimo token visto e l'operatore spaziale. Restituisce il token successivo, dopo aver consultato la matrice M . La ricerca sulla matrice avviene in questo modo:

- Se l'operatore spaziale è *HOR* il calcolo del token successivo avviene in questo modo: si cerca un token non visto nelle prime colonne a destra della matrice M del token già visto.
- Se invece l'operatore spaziale è *VER* il calcolo del token successivo avviene cercando un token non ancora visto nella righe successive della matrice M ad un token già visto.

La funzione *getNextToken()* prende in input anche una lista di token già visti. Ciò comporta che la ricerca del token successivo avviene consultando anche questa lista, ovviamente seguendo i principi definiti dagli operatori spaziali.

Capitolo 5

Implementazione del GLL parsing

5.1 Introduzione

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

5.2 Le componenti del sistema

Per realizzare il GLL parsing si è cercato di creare tutte le componenti che utilizza l'algoritmo durante la sua computazione e risultano essere:

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

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

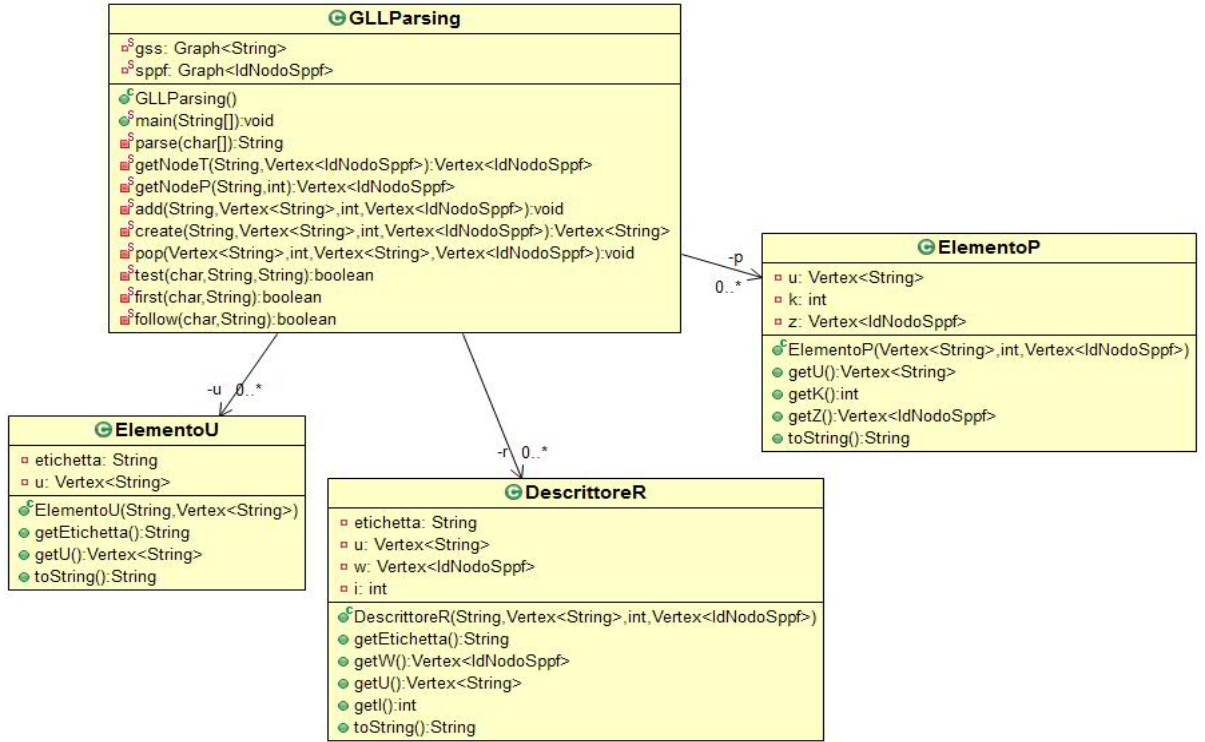


Figura 5.1: Class diagram del software del GLL Parsing

5.3 Le classi degli insiemi R, P e U

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

```

1 package gllparsing;
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 }
  
```

```

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*, ed un nodo *u*, di tipo *Vertex*, che mantiene traccia del nodo del GSS che il parser sta processando. Presenta un costruttore per inizializzare le variabili d'istanza al momento della creazione dell'oggetto (linee 8-10), dei metodi d'accesso alle variabili d'istanza (linee 13-19) ed il metodo *toString()* per descrivere lo stato dell'oggetto (linee 21-22).

Ora descriviamo la classe *ElementoP.java*

```

1  package gllparsing;
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*, che rappresenta un nodo del GSS, un intero *k* e 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). Notiamo da questa classe che è presente un nodo del sppf. Questo perchè ogni volta che viene fatto un *pop()* il parser deve riprendere la costruzione dal nodo padre, che rappresenta il corpo di una produzione, a cui è stato aggiunto un nodo figlio, che sono i terminali che sono utilizzati per sostituire i non-terminali di un corpo di una produzione.

Ora presentiamo la classe *DescrittoreR.java*.

```

1  package gllparsing;
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
28     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 sul quale sta avvenendo la computazione, un intero *i*, che indica un simbolo della stringa in ingresso che si vuole trovare ed un nodo *w*, che indica un nodo del sppf su cui il parser deve effettuare la sostituzione con un corpo di una produzione (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.java* che definisce l'implementazione del GLL Parsing. Illustremo in più parti le varie operazioni che svolge. Il parsing viene eseguito sulla grammatica 3.1

```

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

```



```

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

```

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

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

```

1  private static void add(String etichetta, Vertex<String> nu, int j,
    Vertex<IdNodoSppf> cn){
2      if((u.size()==0)&&(r.size()==0)){
3          u.add(new ElementoU(etichetta,nu));
4          r.add(new DescrittoreR(etichetta,nu,j,cn));
5      }
6      else{
7          ElementoU el=u.get(j);
8          if(!((el.getEtichetta().equals(etichetta))&&(el.getU().element().
              equals(nu.element())))){
9              u.add(j,new ElementoU(etichetta,nu));
10             r.add(new DescrittoreR(etichetta,nu,j,cn));
11         }
12     }
13 }

```

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

```

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

```

```

18     while(eset.hasNext()){
19         Edge<String> ed=eset.next();
20         Vertex<String> u1=ed.getStartVertex();
21         Vertex<String> u2=ed.getEndVertex();
22         if((u1.element().equals(v.element()))&&(u2.element().equals(u.
           element()))){
23             flag=false;
24         }
25     }
26     if(flag){
27         gss.insertDirectedEdge(v, u, "");
28         for(ElementoP elp:p){
29             if(elp.getU().element().equals(v.element())){
30                 add(etichetta,u,elp.getK(),elp.getZ());
31             }
32         }
33     }
34     return v;
35 }

```

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

```

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

```

```

15     }
16     }
17 }

```

Il metodo *pop()* ha come parametri un nodo del GSS *u*, un intero *j*, che indica una posizione del simbolo della stringa in input da trovare, il nodo *u0*, che 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 figlio *v1* di *u* nel GSS chiamiamo il metodo *add()* e gli passiamo come parametri l'etichetta (che sarebbe una parte del nome del nodo *u*), il nodo *v* del GSS, l'intero *j* e il nodo *cn* del sppf.

```

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

```

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 3.4.1. Il metodo *getNodeT()* prende come parametro

nomeNodo che indica il nome del nodo che deve avere il nuovo nodo del sppf e un nodo *cn*. Questo metodo inserisce l'arco tra il nodo *cn* e il nodo *v* di nome *nomeNodo*. Per identificare univocamente i nodi dell'sppf settiamo un id con l'*hashcode* dell'oggetto. Il metodo *getNodeP()* prende come parametro un intero *i*, un identificativo del nodo dell'sppf. Questo metodo restituisce il nodo padre *v1* che ha per figlio il nodo con l'identificativo *i*.

Ora analizziamo il metodo *parse()*;

```

1  private static String parse(char []buf){
2      //dichirazione indici
3      int i=0;
4      //inizializzo etichette
5      String etichetta ="LS";
6      //creiamo i nodi del gss inserendo un arco tra u0 e u1
7      Vertex<String> u0=gss.insertVertex("$");
8      gss.insertVertex("LsOLO");
9      gss.insertDirectedEdge(gss.getLastNode(),u0, "");
10     Vertex<String> cu=gss.getLastNode();
11     //creazione nodo sppf
12     Vertex<IdNodoSppf> cn=sppf.insertVertex(new IdNodoSppf("S"));
13     cn.element().setId(cn.hashCode());
14     while(true){
15         switch(etichetta){
16             //non terminale LL1
17             //S
18             case "LS":
19                 if(test(buf[i],"S","ASd")){
20                     add("LS1",cu,i,cn);
21                 }
22                 if(test(buf[i],"S","BS")){
23                     add("LS2",cu,i,cn);
24                 }
25                 if((buf[i]=='d')||(buf[i]=='$')) {
26                     add("LS3",cu,i,cn);
27                 }
28                 etichetta="L0";
29                 break;
30             //terminale LL1 A
31             case "LA":
32                 if(test(buf[i],"A","a")){
33                     etichetta="La";
34                 }
35                 if(test(buf[i],"A","c")){
36                     etichetta="Lc";
37                 }
38                 break;
39             //terminale LL1 B
40             case "LB":

```

```
41         if(test(buf[i], "B", "a")){
42             etichetta="Lab";
43         }
44         if(test(buf[i], "B", "b")){
45             etichetta="Lb";
46         }
47         break;
48     //ASd
49     case "LS1":
50         if(test(buf[i], "S", "ASd")){
51             cu=create("L1", cu, i, cn);
52             cn=getNodeT("ASd", cn);
53             etichetta="LA";
54         }
55         else{
56             etichetta="LO";
57         }
58         break;
59     //A.Sd
60     case "L1":
61         if(test(buf[i], "S", "Sd")){
62             cu=create("L2", cu, i, cn);
63             etichetta="LS";
64         }
65         else{
66             etichetta="LO";
67         }
68         break;
69     //AS.d
70     case "L2":
71         if(buf[i]=='d'){
72             i++;
73             etichetta="Ld";
74         }
75         else{
76             etichetta="LO";
77         }
78         break;
79     //ASd.
80     case "Ld":
81         pop(cu, i, u0, cn);
82         etichetta="LO";
83         break;
84     //BS
85     case "LS2":
86         if(test(buf[i], "S", "BS")){
87             cu=create("L3", cu, i, cn);
88             cn=getNodeT("BS", cn);
89             etichetta="LB";
```

```

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

```

```

139         etichetta="La1b";
140     }
141     else{
142         etichetta="L0";
143     }
144     break;
145 //a.
146 case "La1b":
147     cn=getNodeP("a",cn.hashCode());
148     pop(cu,i,u0,cn);
149     etichetta="L0";
150     break;
151 //b
152 case "Lb":
153     if(buf[i]=='b'){
154         cn=getNodeT("b",cn);
155         i++;
156         etichetta="Lb1";
157     }
158     else{
159         etichetta="L0";
160     }
161     break;
162 //b.
163 case "Lb1":
164     cn=getNodeP("b",cn.hashCode());
165     pop(cu,i,u0,cn);
166     etichetta="L0";
167     break;
168 //c
169 case "Lc":
170     if(buf[i]=='c'){
171         cn=getNodeT("c",cn);
172         i++;
173         etichetta="Lc1";
174     }
175     else{
176         etichetta="L0";
177     }
178     break;
179 //c.
180 case "Lc1":
181     cn=getNodeP("c",cn.hashCode());
182     pop(cu,i,u0,cn);
183     etichetta="L0";
184     break;
185 case "L0":
186     if(r.size()>0){
187         etichetta=r.get(0).getEtichetta();

```



```

188         i=r.get(0).getI();
189         cu=r.get(0).getU();
190         cn=r.get(0).getW();
191         System.out.println(r.get(0));
192         r.remove(0);
193     }
194     else{
195         if(u.size()==0){
196             return "NON SUCCESSO";
197         }
198         else{
199             if((u.get(u.size()-1).getEtichetta().equals("LO"))&&(u.get
200                 (u.size()-1).getU().element().equals(u0.element()))){
201                 return "SUCCESSO";
202             }
203             else{
204                 return "NON SUCCESSO";
205             }
206         }
207     }
208     break;
209 }
210 }

```

Il metodo *parse()* prende come parametro un array *buf* che rappresenta la stringa in input su cui fare il parsing. Alle linee 3-13 vengono inizializzate le variabili usate dal parsing durante la computazione. Si inizializza l'indice *i* a 0 per posizionarlo sul primo elemento della stringa, inseriamo un arco tra i nodi *\$* e *Ls0L0* nel GSS ed inseriamo il simbolo iniziale nell'*sppf* ed inizializza l'etichetta al simbolo iniziale. I **goto** presentati nell'algoritmo del paragrafo 3.5 sono implementati con uno **switch** all'interno di ciclo **while** infinito. I **case** dello **switch** assumono i possibili valori che può avere un etichetta. Un etichetta rappresenta un item o un non-terminale da sostituire. Le linee 18-29 abbiamo la gestione di un non-terminale non LL(1), infatti che abbiamo che se il metodo *test()* ha successo, viene chiamata il metodo *add()*, e ciò di applicare il non-determinismo usando l'insieme **R** ed **U**. Le linee 31-47 mostrano come viene trattato un non-terminale è LL(1). Infatti abbiamo che qui non viene applicato il non determinismo e di conseguenza questo ci permette direttamente di sostituire il non-terminale settando direttamente l'etichetta all'item opportuno. Ora analizziamo le linee 48-83. In queste linee abbiamo la gestione degli item per un corpo di una produzione. Alle linee 49-58 abbiamo che se il metodo *test()* ha successo facciamo varie operazioni. La prima operazione che facciamo è creare un nodo nel GSS con l'etichetta dell'item successivo (metodo *create()*), in questo modo

quando verrà sostituito il simbolo puntato da questo item, sarà possibile continuare la computazione all'item successivo. Poi aggiungiamo un nodo al sppf etichettato con il corpo della produzione in questione (metodo *getNodeT()*). Infine settiamo l'etichetta con il valore che ci permette di sostituire il non-terminale dell'item che si sta analizzando. Se il metodo *test()* fallisce torniamo al caso base *L0*. Le linee 60-68 fanno le stesse operazioni però in questo caso non chiamano il metodo *getNodeT()* in quanto viene chiamato solo dal primo item del corpo della produzione. Nelle linee 70-78 si verifica se è stato trovato un simbolo della stringa in input. Se è vero incrementiamo l'indice *i* in maniera tale da farlo passare al simbolo successivo. Poi viene settata l'etichetta all'item successivo. Se la verifica fallisce viene settata l'etichetta al caso base *L0*. Alle linee 80-83 abbiamo completato un item, ossia il punto è alla fine del corpo della produzione, e di conseguenza viene chiamato il metodo *pop()* e l'etichetta viene settata l'etichetta al caso base *L0*. Alle linee 111-116 viene gestita una produzione che come corpo della produzione è ϵ . Viene chiamato il metodo *getNodeT()* per aggiungere un nodo all'sppf, poi viene chiamato il metodo *getNodeP()* per recuperare il nodo padre del nodo aggiunto precedentemente, chiamiamo il metodo *pop()* e settiamo l'etichetta al caso base *L0*. Osserviamo che il metodo *getNodeT()* va inserito solo quando gestiamo il primo item di un corpo di una produzione e il metodo *getNodeP()* va inserito solo nell'ultimo item il cui corpo di produzione è composto solo da un terminale. Ciò viene dimostrato nelle linee 152-167. Nel caso di produzione con ϵ vanno aggiunti entrambi, prima *getNodeT()* e poi *getNodeP()*. Alle linee 185-207 è presente la gestione del caso base *L0*. Si verifica se l'insieme **R** è pieno. Se lo è viene estratta il primo descrittore da **R** e vengono impostate le variabili *i*, *etichetta*, il nodo del GSS e il nodo dell'sppf per poterlo computare. Dopodichè viene rimosso da **R**. Se **R** non è vuoto si verifica se l'insieme **U** è vuoto. Se lo è il parsing non ha avuto successo e termina. Altrimenti se l'ultimo descrittore di **U** ha come etichetta *L0* e il nodo del GSS è \$ il parsing termina con successo altrimenti termina con un non successo.

Capitolo 6

GLL parsing posizionale

6.1 Introduzione

Nel capitolo seguente vedremo come è stato implementato il GLL parsing su una grammatica posizionale. Descriveremo come sono gestiti gli operatori spaziali e l'input.

6.2 Gestione dell'input

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

```
1 package dataset;
2
3 import java.io.*;
4 import java.util.*;
5 //classe che si occupa di gestire l'input
6 public class InputDataset {
7     private ArrayList<String> buf;
8     private String [][]picture;
9     private int row;
10    private int col;
11
12    public InputDataset() {
13        buf=new ArrayList<String>();
14    }
15
16    //caricamento picture e buffer
17    public void caricaDataset(FileReader inMatrix) throws IOException{
18        Scanner lettore=new Scanner(inMatrix);
19        row=Integer.parseInt(lettore.nextLine());
20        col=Integer.parseInt(lettore.nextLine());
21        picture=new String[row][col];
```

```
22     int i=0,in=0;
23     while(lettore.hasNextLine()) {
24         String [] buf=lettore.nextLine().split(" ");
25         for(int j=0;j<buf.length;j++) {
26             this.buf.add(buf[j]);
27             picture[i][j]=""+in;
28             in++;
29         }
30         i++;
31     }
32     lettore.close();
33 }
34
35 //ottieni token
36 public String getToken(int i) {
37     return buf.get(i);
38 }
39
40 //metodo per settare un valore trovato
41 public void setTokenFound(int index,ArrayList<String> tokenViews) {
42     if(tokenViews.size()==0) {
43         tokenViews.add(""+index);
44     }
45     else {
46         boolean flag=true;
47         for(String t:tokenViews) {
48             if(t.equals(""+index)) {
49                 flag=false;
50             }
51         }
52         if(flag) {
53             tokenViews.add(""+index);
54         }
55     }
56 }
57
58 //verifica se e' stato visto il token index
59 private boolean isViewed(String index,ArrayList<String> tokenViews) {
60     for(String s:tokenViews) {
61         if(s.equals(index)){
62             return false;
63         }
64     }
65     return true;
66 }
67
68 //metodo che calcola il simbolo da processare
69 public int getNextToken(int index,String dirSymbol,ArrayList<String>
    tokenViews){
```

```
70     int i,j,newCol=0,newRow=0;
71     for(i=0;i<row;i++){
72         for(j=0;j<col;j++){
73             if((picture[i][j]!=null)&&(picture[i][j].equals(""+index))){
74                 switch(dirSymbol){
75                     case "<":
76                         if(i==(row-1)) {newRow=0;}else {newRow=i+1;}
77                         newCol=0;
78                         break;
79                     case ">":
80                         if(j==(col-1)) {newCol=0;}else {newCol=j+1;}
81                         newRow=0;
82                         if(picture[i][newCol]==null) {newCol=0;}
83                         break;
84                 }
85                 for(int k=newRow;k<row;k++) {
86                     for(int l=newCol;l<col;l++) {
87                         if((picture[k][l]!=null)&&(isViewed(picture[k][l],
88                             tokenViews))) {
89                             return Integer.parseInt(picture[k][l]);
90                         }
91                     }
92                 }
93             }
94         }
95     }
96     return -1;
97 }
```

6.3 La classe GLLParserBidimensionale

mettere la classe del gll bidimensionale

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).