UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI

Corso di Laurea in Informatica

Tesi di Laurea

Realizzazione di Definite Clause Grammar in Java tramite JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Andrea Longo

Anno Accademico 2010/2011

Ai miei genitori, Gino e Patrizia.
A mia sorella, Elena.

Indice

Introduzione

Le *Definite Clause Grammar* (**DCG** [1][4]) sono uno specifico formalismo utilizzato per descrivere grammatiche. Il termine *Definite Clause Grammar* deriva dall'interpretare una grammatica libera dal contesto come un insieme di *Clausole di Horn* su una logica del prim'ordine. Tale logica è ciò su cui si basano i linguaggi di programmazione logica, quali il Prolog. Risulta perciò del tutto naturale integrare le **DCG** nel Prolog [5].

JSetL [2][3][10] è una libreria Java che offre, all'interno di un contesto *object-oriented*, molti dei concetti tipici dei linguaggi di programmazione logica come il Prolog, quali ad esempio variabili logiche, liste parzialmente specificate, unificazione, risoluzione di vincoli e non-determinismo.

Scopo di questa tesi è di indagare la possibilità di realizzare **DCG** all'interno di un programma Java sfruttando le possibilità offerte da **JSetL**, in modo del tutto analogo a quanto avviene con il Prolog.

L'idea di base è quella di realizzare ciascuna produzione di una **DCG** come un nuovo vincolo **JSetL** (classe `NewConstraintsClass`), sfruttando per la sua implementazione quanto offerto dalla libreria per rappresentare e operare su liste parzialmente specificate di oggetti qualsiasi e per esprimere e gestire il non-determinismo.

Il lavoro di tesi è organizzato nel modo seguente:
Il capitolo 1 introduce brevemente le nozioni di *grammatica* e di *clausola definita*, quindi descrive più in dettaglio le **DCG** ed il loro utilizzo all'interno del Prolog.
Il capitolo 2 introduce brevemente il concetto di *algoritmo non-deterministico* e quindi presenta la libreria **JSetL**, soffermandosi sulle funzionalità da noi utilizzate, con particolare attenzione alle liste logiche, alla definizione di nuovi vincoli ed alla gestione del non-determinismo all'interno di questi.
Il capitolo 3 mostra come descrivere **DCG** utilizzando le strutture di **JSetL** presentate in precedenza; mostra di fatto una mappatura dalla notazione Prolog nei costrutti di **JSetL**.
Il capitolo 4 mostra l'implementazione in Java di un traduttore completo

(analizzatore lessicale, sintattico e generatore di codice) per un linguaggio di semplici espressioni aritmetiche, realizzato tramite **JSetL** a partire dalla descrizione con **DCG** della grammatica del linguaggio, secondo la tecnica descritta nel capitolo precedente; il capitolo mostra anche come tale traduttore possa essere utilizzato all'interno di JSetL stesso per permettere di esprimere in modo più semplice le espressioni che appaiono nei vincoli aritmetici presenti nella libreria.
Il capitolo 5 spiega le ottimizzazioni generali da applicare per rendere più efficienti le **DCG** implementate in **JSetL**, mostrando i risultati di tali applicazioni alle grammatiche descritte nel capitolo precedente.
Infine nel capitolo 6 vi è spazio per conclusioni e sviluppi futuri.

Capitolo 1

Definite Clause Grammar

Con il termine *Definite Clause Grammar* (**DCG**) ci riferiamo ad uno specifico formalismo utilizzato per descrivere grammatiche. Tale formalismo viene sfruttato da linguaggi basati su paradigmi di programmazione logica, in particolare Prolog. Il termine *Definite Clause Grammar* deriva dall'interpretare una grammatica come insieme di *Clausole di Horn* su una logica del prim'ordine; questa logica è ciò su cui si basa il paradigma di programmazione logica per rappresentare ed elaborare l'informazione. Per dare una definizione formale di **DCG** bisogna prima introdurre i concetti di *Grammatica Formale* e di *Clausola di Horn*.

1.1 Grammatiche formali e CFG

Si definisce *Grammatica Formale* un insieme di regole formali per costruire stringhe a partire da un dato alfabeto; l'insieme delle stringhe generate da una *Grammatica Formale* è detto *Linguaggio Formale*.

La definizione comunemente utilizzata, data da Noam Chomsky, è la seguente:

DEFINIZIONE: Una grammatica G è una quadrupla $< V, T, I, P >$ dove:

- V è un insieme di simboli *non-terminali*;
- T è un insieme di simboli *terminali* tale che $T \cap V = \emptyset$;
- $I \in V$ è un simbolo speciale detto *Simbolo Iniziale*;
- P è l'insieme delle regole di produzione, dove ogni regola è della forma:

$$(T \cup V)^* V (T \cup V)^* \rightarrow (T \cup V)^*$$

dove * è la chiusura di Kleene.

Data una grammatica $G =< V, T, I, P >$ si definisce la relazione binaria $\Rightarrow \subseteq (V \cup T)^* \times (V \cup T)^*$ come segue:

$$x \Rightarrow y \iff \exists u, v, p, q \in (V \cup T)^* : (x = upv) \wedge (p \to q \in P) \wedge (y = uqv)$$

Questa relazione si dice *Singolo Passo di Derivazione*. La chiusura riflessiva e transitiva della relazione $\Rightarrow$ si scrive $\Rightarrow^*$.

Quindi:

$$x \Rightarrow^* y$$

si legge "x deriva y in zero o più passi".

Il linguaggio descritto da una grammatica, denotato come $L(G)$, è pertanto l'insieme delle stringhe tali che:

$$L(G) \ := \ \{x \in T^* \mid S \Rightarrow^* x\}$$

Un caso particolare di grammatiche formali sono le *Grammatiche Libere dal Contesto* (*Context-Free* [6]). In queste grammatiche le regole di produzione sono della forma:

$$V \to (T \cup V)^*$$

Un linguaggio si dice *Libero dal Contesto* se esiste una *Grammatica Libera dal Contesto* che lo genera.
Per esempio possiamo prendere la grammatica $G \ = < V, T, I, P >$ così descritta:

$T = \{ -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$,
$V = \{ i, c, n \}$,
$I = i$,
$P = \{ \quad i \to c \quad , \quad i \to -c \quad , \quad i \to cn \quad , \quad i \to -cn \ , \quad i \to 0 \quad ,$
$\quad\quad n \to c \quad , \quad n \to cn \quad , \quad n \to 0 \quad , \quad n \to 0n \ ,$
$\quad\quad c \to 1 \quad , \quad c \to 2 \quad , \quad c \to 3 \quad , \quad c \to 4 \quad , \quad c \to 5 \quad ,$
$\quad\quad c \to 6 \quad , \quad c \to 7 \quad , \quad c \to 8 \quad , \quad c \to 9 \quad \}$

Il linguaggio generato da G è l'insieme delle stringhe rappresentanti i numeri interi.

La derivazione per il numero -2506 per esempio è la seguente:

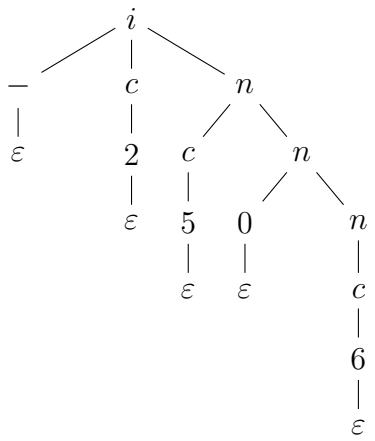
$$i \Rightarrow -cn \Rightarrow -2n \Rightarrow -2cn \Rightarrow -25n \Rightarrow -250n \Rightarrow -250c \Rightarrow -2506$$

Si noti che la scelta fra più produzioni di uno stesso non-terminale è non-deterministica.

È possibile dare una rappresentazione grafica delle derivazioni utilizzando gli *Alberi di Derivazione* (detti anche *Parse Tree* o *Parsing Tree*).
Data una grammatica $G =< V, T, I, P >$, un albero è un *Albero di Derivazione* per G se:

- ogni nodo è etichettato con un simbolo appartenente a $V \cup T \cup \{ \varepsilon \}$, dove ε rappresenta la stringa vuota;
- la radice ed i nodi interni dell'albero sono etichettati ognuno con un simbolo appartenente a V;
- se un nodo n è etichettato con il simbolo $A \in V$ ed i nodi (ordinatamente da sinistra verso destra) $n_1, \ldots, n_k$ sono i figli di n tali che ogni n_i ha etichetta $X_i \in V \cup T$, allora $A \to X_1 \ldots X_k \in P$;
- se un nodo n è etichettato con il simbolo ε, allora n è una foglia ed è l'unico figlio di suo padre.

Ogni stringa generata da una grammatica è rappresentabile tramite un *Albero di Derivazione* radicato nel simbolo iniziale della grammatica.
Per fare un esempio concreto, l'*Albero di Derivazione* della stringa -2506 utilizzando la grammatica precedente è:



Una produzione di una grammatica libera dal contesto (*CFG* da qui in poi, per *Context-Free Grammar*) è composta da:

$$testa \to corpo_1,\ corpo_2,\ \ldots\ ,\ corpo_n$$

dove *testa* è un simbolo non-terminale e $corpo_1,\ \ldots\ ,\ corpo_n$ sono simboli terminali o non-terminali.
Introduciamo ora alcune regole sintattiche per la scrittura delle *Grammatiche Libere dal Contesto*:

- i simboli non-terminali (V) saranno scritti usando delle stringhe che iniziano con lettera minuscola; per esempio:

  $$x,\ nome,\ float\_15$$

  sono non-terminali validi;

- i simboli terminali (T) saranno simboli o stringhe racchiusi fra parentesi quadre; nel caso la stringa terminale inizi per lettera maiuscola o contenga caratteri particolari, quali spazi o segni di punteggiatura, è buona norma racchiuderla fra apici singoli[1]; per esempio:

  $$[a],\ [18],\ [\text{'}Il\ linguaggio\ Prolog\text{'}],\ [\,]$$

  sono terminali validi ([ ] indica la stringa vuota);

- ogni simbolo a destra di una produzione sarà diviso da un altro simbolo da una virgola; nel caso più simboli terminali siano vicini essi potranno esser contenuti fra le stesse parentesi quadre, dividendoli tramite virgole; per esempio:

  $$[Il],\ soggetto,\ [osserva,\ il],\ complemento\_oggetto$$

  è una sequenza valida di terminali e non-terminali;

- con il simbolo di pipe ('|') suddivideremo più produzioni per uno stesso non-terminale; per esempio:

  $$soggetto \rightarrow [gatto] \mid [cane] \mid [fagiano]$$

  indica tre diverse produzione per il non-terminale *soggetto*.

Per fare un esempio pratico:

$$\begin{array}{lll}
expr & \rightarrow & num \mid \\
 & & [-],\ num \mid \\
 & & expr,\ [+],\ expr \mid \\
 & & expr,\ [-],\ expr \mid \\
 & & expr,\ [*],\ expr \mid \\
 & & expr,\ [/],\ expr \\
num & \rightarrow & cifra \mid [0] \mid cifra,\ num \mid [0],\ num \\
cifra & \rightarrow & [1] \mid [2] \mid [3] \mid [4] \mid [5] \mid [6] \mid [7] \mid [8] \mid [9]
\end{array}$$

sono le produzioni di una grammatica che descrive il linguaggio contenente semplici espressioni aritmetiche fra interi.

[1] Questa notazione viene introdotta per essere logicamente leggibile ed allo stesso tempo il più simile possibile alla sintassi Prolog, che verrà presentata in seguito.

1.2 Clausole di Horn e clausole definite

1.2.1 Definizione

Una *Clausola di Horn* è una disgiunzione finita di letterali con al massimo un letterale positivo. Una clausola di Horn con esattamente un letterale positivo è detta *Clausola Definita* (*Definite Clause*). In una Clausola Definita il letterale positivo è detto *Testa* mentre i letterali negativi formano il *Corpo* della clausola. Una clausola di Horn composta di soli letterali negativi è invece detta *Goal*. Una clausola definita è solitamente scritta come:

$$\neg L_1 \vee \neg L_2 \vee ... \vee \neg L_n \vee L$$

dove $n \geq 0$ e L è l'unico letterale positivo. Questa si può anche scrivere:

$$L_1 \wedge L_2 \wedge ... \wedge L_n \rightarrow L$$

1.2.2 Programmazione Logica e Prolog

Le *Clausole Definite* appena descritte formano la base del linguaggio di programmazione Prolog [1][9]. Prolog opera su oggetti denominati *Termini*. Un *Termine* può essere:

- una **costante**, ovvero un **intero** o un **atomo** e denotano oggetti elementari con valore definito e immutabile; un *atomo* è un simbolo denotato da una serie di caratteri solitamente scritti fra apici singoli, a meno che non vi sia possibilità di confonderlo con altri simboli (come variabili o interi); i seguenti simboli:

  `1`, `999`, `'Pw'`, `pw`, `'Prolog è un linguaggio dichiarativo'`

  rappresentano *costanti* valide per il linguaggio;

- una **variabile**, ovvero un identificatore per un oggetto ben preciso ma il cui valore può essere sconosciuto; si noti che a differenza delle variabili dei linguaggi convenzionali, le variabili Prolog non sono semplici contenitori a cui assegnare valori, bensì identificativi per termini di valore arbitrario, similarmente alle variabili matematiche; una *variabile* Prolog è denotata da una serie di caratteri che comincia con una lettera maiuscola; i seguenti simboli:

  `X`, `Contatore`, `Albero_di_sintassi`, `A1`

  sono *variabili* valide per il linguaggio;

- un **compund term** o **termine composto**, ovvero un oggetto composto da una combinazione dei precedenti; i *compund terms* sono denotati da un **funtore** (detto funtore **principale** del termine) ed una sequenza di uno o più termini detti **argomenti**, racchiusi fra parentesi e divisi da virgole; un *funtore* è caratterizzato dal suo **nome**, che è un *atomo*, e dalla sua **arità**, ovvero il numero di *argomenti*; per esempio:

  `frazioni_di_Parma('Vigheffio', 'Alberi', 'Vigatto')`, `punto(X, Y, Z)`, `persona(Nome, Cognome, maschio)`

  sono *termini composti* validi; in alcuni casi può essere utile utilizzare una notazione infissa del *funtore*, ponendolo fra i suoi *argomenti*, come nel caso di:

  `X + Y`, `(P;Q)`, `A>B`

  che sono equivalenti a:

  `+(X, Y)`, `;(P,Q)`, `>(A,B)`;

  un *atomo* può essere visto come un *funtore* di *arità* 0.

Un importante classe di dati Prolog è rappresentata dalle **Liste**. Una *Lista* è essenzialmente:

- l'atomo `[ ]`, rappresentante la lista vuota;
- un *termine composto* con funtore `'.'` e due argomenti che sono, rispettivamente, la **testa** e la **coda** della lista;

Una lista i cui elementi sono, nell'ordine, `a`, `b` e `c` può essere rappresentata dal *compound term*:

`.(a, .(b, .(c, [ ])))`

È tuttavia possibile (e consigliato) utilizzare una notazione più intuitiva per le liste, ovvero:

`[a, b, c]`

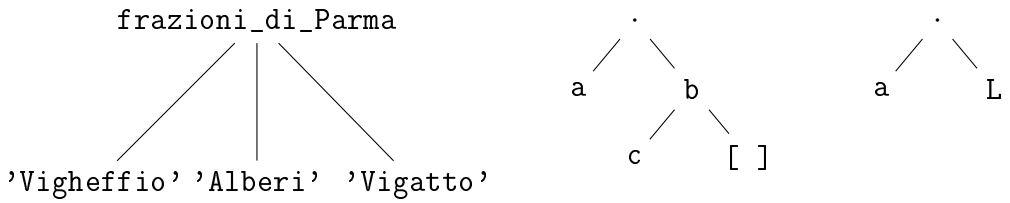
Nel caso in cui la coda di una lista sia una variabile, è utilizzata la notazione:

`[a|L]`

che equivale alla notazione standard:

```
·(a, L)
```

I *termini composti* sono funzionalmente immaginabili come alberi; per esempio:



rappresentano alcuni dei *compound terms* presentati come esempio in precedenza.

Un programma logico consiste semplicemente di una sequenza di **clausole**, ognuna delle quali è composta da una **testa** ed un **corpo**; *testa* e *corpo* di una *clausola* sono costituiti da uno o più **letterali**. I *letterali* (predicati positivi o negativi) in Prolog hanno la stessa forma sintattica dei *termini* e si distinguono da questi solo per il contesto in cui compaiono all'interno del programma. I *letterali* che appaiono nel corpo di una *clausola* sono detti **goal**. Il *funtore principale* di un *goal* è detto **predicato**. Una *clausola* si presenta in tre forme:

- **clausola non-unitaria**, se sia la *testa* che il *corpo* sono non vuoti; in questo caso viene scritta nella forma:

  ```
  P :- Q, R, S.
  ```

  dove `P` è la *testa* e `Q`, `R` e `S` sono i *goal* che compongono il *corpo*; possiamo leggere questa clausola in maniera **dichiarativa**:

  "Per ogni $\overline{\texttt{X}}$*,* `P` *è vera se* `Q`*,* `R` *e* `S` *sono vere"*

  (ovvero come *clausola definita* $Q \wedge R \wedge S \rightarrow P$) oppure in maniera **procedurale**:

  "Per soddisfare `P`*, soddisfare* `Q`*,* `R` *e* `S`*".*

- **clausola unitaria**, se il *corpo* della clausola è vuoto; in questo caso viene scritta nella forma:

`P.`

e viene interpretata *dichiarativamente* come:

"Per ogni $\overline{\texttt{X}}$*,* `P` *è vera"*

o *proceduralmente* come:

"`P` *è soddisfatta.".*

- **domanda**, se la *testa* della clausola è vuota; in questo caso viene scritta nella forma:

`?-P, Q.`

che può essere letta in modo *dichiarativo* come:

"Esiste $\overline{\texttt{X}}$ *tale che* `P` *e* `Q` *sono veri?"*

o *procedurale* come:

"Soddisfa `P` *e* `Q`*.".*

Le clausole generalmente contengono *variabili*; queste sono locali alla clausola. Una *variabile* indica un valore qualsiasi.
Diamo alcuni esempi di programmi Prolog con la relativa interpretazione *dichiarativa*:

```
sposato(X) :- moglie(Y, X).
  % Per ogni X e Y, X e' sposato se esiste un Y moglie di X.

moltiplicazione(X, 0, 0).
  % Per ogni X, la moltiplicazione di X con 0 e' 0.

?- mammifero(X), guaisce(X).
  % Esiste un X tale che X e' un mammifero e X guaisce?
```

1.3 Dalle CFG alle DCG

Con *Definite Clause Grammar* (**DCG**) intendiamo un formalismo utilizzato per esprimere grammatiche formali e naturali. Il termine, coniato da Fernando C. N. Pereira e David H. D. Warren [1], deriva dal voler esprimere le grammatiche come un insieme di clausole definite in una logica del prim'ordine.

Abbiamo precedentemente introdotto le grammatiche libere dal contesto come un caso specifico di grammatiche formali. Ogni produzione di una *CFG* può essere vista come zucchero sintattico per esprimere una clausola definita. Possiamo vedere le sue produzioni come segue:

$$testa(S_0, S) \leftarrow corpo_1(S_0, S_1),\ corpo_2(S_1, S_2),\ \dots\ ,\ corpo_n(S_{n-1}, S)$$

dove:

- $\leftarrow$ indica semplicemente l'implicazione;
- le stringhe che iniziano per lettera maiuscola sono *Variabili*;
- un simbolo non-terminale $nt \in V$ è visto come predicato di arità 2 $nt(S_i,\ S_f)$, vero se la sottostringa che si estende da S_i a S_f è derivabile da nt;
- una lista di simboli terminali $[t_1,\ \dots\ ,\ t_n]$ con $t_k \in T,\ \forall k \in \{1,\ \dots\ ,\ n\}$, è vista come un predicato di arità 2 $[t_1,\ \dots\ ,\ t_n](S_i,\ S_f)$, vero se la sottostringa che si estende da S_i a S_f è $t_1 ... t_n$.

La grammatica mostrata in precedenza può quindi essere riscritta così:

$$\begin{array}{lcl}
expr(S_0, S) & \leftarrow & num(S_0, S) \\
expr(S_0, S) & \leftarrow & [-](S_0, S_1),\ num(S_1, S) \\
expr(S_0, S) & \leftarrow & expr(S_0, S_1), [+](S_1, S_2),\ expr(S_2, S) \\
expr(S_0, S) & \leftarrow & expr(S_0, S_1), [-](S_1, S_2),\ expr(S_2, S) \\
expr(S_0, S) & \leftarrow & expr(S_0, S_1), [*](S_1, S_2),\ expr(S_2, S) \\
expr(S_0, S) & \leftarrow & expr(S_0, S_1), [/](S_1, S_2),\ expr(S_2, S) \\
num(S_0, S) & \leftarrow & cifra(S_0, S) \\
num(S_0, S) & \leftarrow & [0](S_0, S) \\
num(S_0, S) & \leftarrow & cifra(S_0, S_1),\ num(S_1, S) \\
num(S_0, S) & \leftarrow & [0](S_0, S),\ num(S_1, S) \\
cifra(S_0, S) & \leftarrow & [1](S_0, S) \\
\vdots & \vdots & \vdots \\
cifra(S_0, S) & \leftarrow & [9](S_0, S)
\end{array}$$

Queste clausole possono leggersi come precedentemente descritto per le clausole definite. La terza produzione del non-terminale *expr*, per esempio, la si può interpretare in questo modo:

" un'espressione si estende da S_0 a S se vi è:

- un'espressione che si estende da S_0 a S_1;
- un simbolo terminale '+' fra S_1 e S_2;
- un'espressione che si estende da S_2 a S ".

Esprimendo le *CFG* come clausole definite è possibile trattarle in maniera meccanica, dividendo ogni *goal* in una serie di *sotto-goal* risolvibili sfruttando regole di inferenza. Una serie di clausole di questo tipo rappresentano un vero e proprio programma per linguaggi come il Prolog.

Per ottenere la nostra definizione di *Definite Clause Grammar* possiamo generalizzare le *CFG* stando attenti a mantenere la corrispondenza con le clausole definite.

Aggiungiamo alla notazione che abbiamo dato per le *CFG* due punti basilari:

- i simboli non-terminali possono essere predicati e quindi avere argomenti; per esempio:

  $$soggetto(nome),\ frazione(N,\ D),\ immagine(Nome,\ tipo(jpg))$$

  sono non-terminali validi;

- a destra di una produzione possono esserci delle cosiddette "*Procedure Calls*", scritte fra parentesi graffe ('{' e '}'), rappresentanti condizioni extra da soddisfare, le quali non producono simboli; per esempio:

  $$num(X) \ \rightarrow \ [X],\ \{\ X\ \in\ \mathbb{R},\ 15 \leq X \leq 18\ \}$$

  è una produzione valida.

Prendendo la grammatica mostrata in precedenza e immaginando di voler calcolare anche il risultato dell'espressione potremmo per esempio riscriverla come segue:

$$
\begin{array}{lcl}
expr(X) & \rightarrow & num(X) \mid \\
 & & expr(Y),\ [+],\ expr(Z),\ \{\ X = Y + Z\ \} \mid \\
 & & expr(Y),\ [-],\ expr(Z),\ \{\ X = Y - Z\ \} \mid \\
 & & expr(Y),\ [*],\ expr(Z),\ \{\ X = Y * Z\ \} \mid \\
 & & expr(Y),\ [/],\ expr(Z),\ \{\ X = Y/Z\ \} \\
num(X) & \rightarrow & [X],\ \{\ X\ \in\ \mathbb{Z}\ \}
\end{array}
$$

Nell'esempio qui esposto le condizioni extra sono scritte in maniera più informale, dando per scontata la conoscienza dell'insieme dei numeri interi e delle loro operazioni aritmetiche di base, ma il significato è quello ovvio e la forma risulta più concisa.

Dobbiamo ora tradurre questa notazione estesa in clausole definite, come fatto prima per le *CFG*. Quindi:

- un simbolo non-terminale di arità K *nt*($a_1,\ \ldots\ ,\ a_k$) viene tradotto come predicato di arità $K + 2$, dove i primi K argomenti sono quelli esplicitati nel non-terminale e gli ultimi 2 sono come quelli descritti per le *CFG*, ovvero *nt*($a_1,\ \ldots\ ,\ a_k,\ S_i,\ S_f$);
- le *'Procedure Calls'* vengono semplicemente trascritte, omettendo le parentesi graffe.

Possiamo quindi riscrivere l'esempio di prima nel seguente modo:

$$
\begin{array}{lcl}
expr(X, S_0, S) & \leftarrow & num(X, S_0, S) \\
expr(X, S_0, S) & \leftarrow & expr(Y, S_0, S_1),\ [+](S_1, S_2),\ expr(Z, S_2, S),\ X = Y + Z \\
expr(X, S_0, S) & \leftarrow & expr(Y, S_0, S_1),\ [-](S_1, S_2),\ expr(Z, S_2, S),\ X = Y - Z \\
expr(X, S_0, S) & \leftarrow & expr(Y, S_0, S_1),\ [*](S_1, S_2),\ expr(Z, S_2, S),\ X = Y * Z \\
expr(X, S_0, S) & \leftarrow & expr(Y, S_0, S_1),\ [/](S_1, S_2),\ expr(Z, S_2, S),\ X = Y/Z \\
num(X, S_0, S) & \leftarrow & [X](S_0, S),\ X\ \in\ \mathbb{Z}
\end{array}
$$

La seconda produzione per *expr(X)* si può interpretare come segue:

" un'espressione si estende da S_0 a S con valore X se:

- vi è un'espressione che si estende da S_0 a S_1 con valore Y;
- vi è un simbolo terminale '+' fra S_1 e S_2;
- vi è un'espressione che si estende da S_2 a S con valore Z;
- X è uguale a $Y + Z$ ".

È importante notare che le **DCG** rappresentano un formalismo più potente delle *CFG*, nonostante la definizione qui data derivi da queste ultime. Grammatiche dipendenti dal contesto possono essere definite tramite le **DCG** aggiungendo argomenti extra ai non-terminali. Consideriamo questa grammatica di esempio:

$$
\begin{array}{ll}
stringa(X) & \rightarrow a(X),\ b(X),\ c(X) \\
a(0) & \rightarrow [a] \\
a(s(X)) & \rightarrow [a],\ a(X) \\
b(0) & \rightarrow [b] \\
b(s(X)) & \rightarrow [b],\ b(X) \\
c(0) & \rightarrow [c] \\
c(s(X)) & \rightarrow [c],\ c(X)
\end{array}
$$

dove *stringa* è il simbolo iniziale. Questa grammatica genera l'insieme delle stringhe della forma $a^n b^n c^n$ con $n > 0$[2]. Per esempio la stringa *aaabbbccc* si ottiene dalla derivazione:

$$
\begin{array}{ll}
stringa(s(s(0))) & \Rightarrow a(s(s(0))), b(s(s(0))), c(s(s(0))) \\
& \Rightarrow [a], a(s(0)), [b], b(s(0)), [c], c(s(0)) \\
& \Rightarrow [a, a], a(0), [b, b], b(0), [c, c], c(0) \\
& \Rightarrow [a, a, a, b, b, b, c, c, c]
\end{array}
$$

1.4 DCG in Prolog

In Prolog abbiamo due notazioni per esprimere le **DCG**:

- notazione *implicita*;
- notazione *esplicita*.

Nella notazione *implicita* una **DCG** è descritta utilizzando le regole descritte per le grammatiche. La sintassi Prolog denota il simbolo di implicazione con i tre caratteri "`-->`". Possiamo sfruttare alcuni costrutti di Prolog per avere una notazione il più simile possibile a quella già data in precedenza; possiamo quindi raggruppare più produzione per uno stesso non terminale fra le parentesi '(' e ')' e dividendo le produzioni con il simbolo '|'[3].

[2] Si ricorda che in questo ambito x^n è da interpretarsi come "concatenazione del carattere x, n volte", quindi $a^3b^3c^3$ rappresenta la stringa *aaabbbccc*.

[3] Questa notazione è molto comoda se ogni produzione è esprimibile in una, al massimo due righe di testo; in caso contrario è preferibile spezzare le produzioni.

Per esempio le produzioni:

frase → [*'Non posso uscire'*], *motivo*
motivo → [*'devo studiare.'*] | [*'non ho soldi.'*] | [*'mia mamma non vuole.'*]

scritte nell'equivalente notazione *implicita* in Prolog sono:

```
frase -->
  ['Non posso uscire'], motivo.
motivo -->
  ( ['devo studiare.']
  | ['non ho soldi.']
  | ['mia mamma non vuole.']
  ).
```

Si noti che in questo esempio le frasi generate partendo dal non-terminale `frase` sono composte da 2 soli terminali. Per esempio:

```
['Non posso uscire', 'devo studiare.']
```

è composta dai due terminali `['Non posso uscire']` e `['devo studiare.']`; la virgola divide i due elementi della lista.
Se volessimo una grammatica che genera l'intera stringa carattere per carattere possiamo utilizzare la notazione Prolog per le stringhe, ovvero:

```
frase -->
  "Non posso uscire", motivo.
motivo -->
  ( "devo studiare."
  | "non ho soldi."
  | "mia mamma non vuole."
  ).
```

In Prolog le stringhe sono infatti una notazione speciale per liste di interi rappresentanti caratteri.

Una **DCG** in notazione *esplicita* è descritta direttamente tramite predicati Prolog, esplicitando la concatenazione delle stringhe.
Una grammatica definita in forma *implicita* viene tradotta in forma *esplicita* aggiungendo gli argomenti necessari.
Consideriamo per esempio la produzione:

testa → *corpo1, corpo2*

Questa viene trascritta in Prolog nella seguente clausola:

```
testa(S0, S) :-
  corpo1(S0, S1),
  corpo2(S1, S).
```

Nel caso la produzione contenga simboli terminali, ne viene esplicitata la posizione. Per esempio:

$$testa \rightarrow corpo1,\ [a, b],\ corpo2$$

viene tradotta in:

```
testa(S0, S) :-
  corpo1(S0, S1),
  S1 = [a, b|S2],
  corpo2(S2, S).
```

Le variabili `S0`, `S1`, `S2` e `S` sono stringhe, quindi liste di interi. Si può notare che la clausola `testa(S0, S)` ha 2 parametri; queste sono "*Difference Lists*".

1.4.1 Difference Lists

In Prolog è possibile rappresentare liste parzialmente specificate.
Prendiamo per esempio la lista $Y = [a, b, c|X]$: questa notazione indica che conosciamo la struttura della lista solo fino ad un certo punto. Sappiamo che i suoi primi 3 elementi sono, nell'ordine, a, b e c, mentre il resto della lista equivale alla lista X. Se X non è specificata, la lista Y si dice *aperta* o *parzialmente specificata*.

Le *Difference Lists* sono un modo per rappresentare un'unica lista come differenza di 2 liste.
Per esempio, la lista $[a, b, c]$ può essere rappresentata dalla coppia di liste $([a, b, c|X], X)$, ovvero:

- dalla lista aperta composta dagli elementi a, b, c in testa e con in coda la lista X;
- dalla lista X, che è quanto bisogna rimuovere dalla fine della lista precedente per ottenere la lista interessata.

Questa rappresentazione ha come vantaggio, rispetto alla rappresentazione classica, quello di poter concatenare molto facilmente elementi alle liste. Prendiamo per esempio la lista aperta $Y = [a, b, c|X]$ e supponiamo di volere aggiungere l'elemento d alla fine della lista, ovvero dire che $Y = [a, b, c, d|Z]$, utilizzando la rappresentazione classica delle liste. Una implementazione per questa operazione in Prolog è la seguente:

```
aggiungi_elemento(Elemento, Lista) :-
  nonvar(Lista),
  Lista=[_Primo_elemento|Coda], !,
  aggiungi_elemento(Elemento, Coda).
aggiungi_elemento(Elemento, Lista) :-
  var(Lista),
  Lista=[Elemento|_Coda].
```

In questo metodo vi sono alcuni punti che è meglio spiegare:

- `nonvar(X)` è vera se il valore di `X` è specificato;
- `var(X)` è vera se il valore `X` non è specificato;
- le variabili che iniziano con il carattere '`_`' rappresentano elementi il cui valore non ci interessa.

Se quindi volessimo usare questo predicato per aggiungere d alla lista Y, il `goal`:

```
?- Y = [a, b, c|X], aggiungi_elemento(d, Y).
```

verrebbe risolta in questo modo:

```
aggiungi_elemento(d, [a, b, c|X])
    aggiungi_elemento(d, [b, c|X])
        aggiungi_elemento(d, [c|X])
            aggiungi_elemento(d, X)
                X = [d|_]
```

Questa operazione ha costo lineare ($\boldsymbol{O(n)}$) rispetto alla lunghezza della parte specificata della lista.
Supponiamo invece di utilizzare le *Difference Lists* e di avere quindi la lista $Y = ([a, b, c|X], X)$, dove X è ovviamente la coda della nostra lista, e di voler aggiungere l'elemento d in fondo a Y. Il predicato per eseguire questa operazione è il seguente:

```
aggiungi_elemento(Elemento, (Lista, [Elemento|Nuova_Coda]),
                  (Lista, Nuova_Coda)).
```

Il terzo argomento del predicato conterrà il risultato dell'operazione. A questo punto il `goal`:

```
?- Y = ([a, b, c|X], X), aggiungi_elemento(d, Y, Z).
```

si risolverebbe con due unificazioni, a costo costante (***O(1)***):

```
aggiungi_elemento(d, ([a, b, c|X], X), ([a, b, c, d|W]))
    X = [d|W]
    Z = ([a, b, c, d|W])
```

Ciò torna molto utile quando si parla di **DCG**. Abbiamo visto che la produzione:

$$testa \rightarrow corpo_1,\ corpo_2$$

viene tradotta in:

$$testa(S0, S) \leftarrow corpo_1(S0, S1),\ corpo_2(S1, S)$$

che equivale alla concatenazione di $(S0, S1)$ e $(S1, S)$. Si può facilmente vedere che tale concatenazione equivale alla lista rappresentata dalla coppia di *Difference Lists* $(S0, S)$.

1.4.2 Limiti delle DCG

Prendiamo per esempio la grammatica precedentemente introdotta scritta in Prolog con notazione *implicita*:

```
expr(X) -->
  ( num(X)
  | expr(Y), [+], expr(Z), {X is Y + Z}
  | expr(Y), [-], expr(Z), {X is Y - Z}
  | expr(Y), [*], expr(Z), {X is Y * Z}
  | expr(Y), [/], expr(Z), {X is Y / Z}
  ).

num(X) -->
  [X], {integer(X)}.
```

Il predicato `integer(X)` equivale alla scrittura precedentemente usata $X \in \mathbb{Z}$.
L'operatore `X is Expr` invece è un predicato Prolog *built-in* usato per valutare un'espressione aritmetica: l'espressione `Expr` è valutata ed il valore risultante è unificato con `X`.
La stessa grammatica scritta in notazione *esplicita* diventa:

```
expr(X, S0, S) :-
  num(X, S0, S).
expr(X, S0, S) :-
  expr(Y, S0, S1), S1 = [+|S2], expr(Z, S2, S), X = Y + Z.
expr(X, S0, S) :-
  expr(Y, S0, S1), S1 = [-|S2], expr(Z, S2, S), X = Y - Z.
expr(X, S0, S) :-
  expr(Y, S0, S1), S1 = [*|S2], expr(Z, S2, S), X = Y * Z.
expr(X, S0, S) :-
  expr(Y, S0, S1), S1 = [/|S2], expr(Z, S2, S), X = Y / Z.

num(X, S0, S) :-
  S0 = [X|S], integer(X).
```

Questa grammatica, seppur logicamente corretta, in Prolog è inutilizzabile. La strategia di esecuzione di Prolog produce un *Recursive Descent Parser* (*Parser a discesa ricorsiva*) per la **DCG**.
Un *Recursive Descent Parser* implementa un processo di *parsing* di tipo *top-down* che cerca di verificare la sintassi di uno stream di input leggendolo da sinistra verso destra, espandendo gli elementi nell'ordine in cui li trova. Nella grammatica presentata questo risulta essere un problema in quanto si avrebbe un ciclo infinito di espansioni per il non-terminale `expr`. Bisogna pertanto avere l'accortezza di evitare grammatiche con *ricorsioni a sinistra*. Per esempio la grammatica prima presentata si può riscrivere senza *ricorsioni a sinistra* nel seguente modo:

```
expr(X) -->
  ( num(X)
  | num(Y), [+], expr(Z), {X is Y + Z}
  | num(Y), [-], expr(Z), {X is Y - Z}
  | num(Y), [*], expr(Z), {X is Y * Z}
  | num(Y), [/], expr(Z), {X is Y / Z}
  ).

num(X) -->
  [X], {integer(X)}.
```

L'equivalente grammatica in notazione *esplicita* è la seguente:

```
expr(X, S0, S) :-
  num(X, S0, S).
expr(X, S0, S) :-
  num(Y, S0, S1), S1 = [+|S2], expr(Z, S2, S), X = Y + Z.
expr(X, S0, S) :-
  num(Y, S0, S1), S1 = [-|S2], expr(Z, S2, S), X = Y - Z.
expr(X, S0, S) :-
  num(Y, S0, S1), S1 = [*|S2], expr(Z, S2, S), X = Y * Z.
expr(X, S0, S) :-
  num(Y, S0, S1), S1 = [/|S2], expr(Z, S2, S), X = Y / Z.

num(X, S0, S) :-
  S0 = [X|S], integer(X).
```

Capitolo 2

Non-determinismo in JSetL

In questo capitolo tratteremo la gestione del non-determinismo in JSetL. Introdurremo quindi la liberia JSetL e faremo alcuni esempi classici di algoritmi non-deterministici, mostrando come questi sono implementati nella suddetta libreria.

2.1 Algoritmi non-deterministici

Non-determinismo indica la possibilità, data una certa situazione, di avere potenzialmente più di una situazione successiva; si contrappone al determinismo che invece definisce la situazione successiva solo in base allo stato attuale. Più formalmente: C è una *Computazione* se è un insieme ordinato ed al più numerabile di *Stati di Computazione*. Possiamo vedere uno *Stato di Computazione* come una fotografia del calcolatore in un istante dell'esecuzione di un programma.
Sia $C = \{ S_0, \ldots, S_i, \ldots \}$ una computazione; C si dice *Finita* se:

$$\exists\, k \in \mathbb{N} \,.\, \forall\, S_i \in C \,:\, i \leq k$$

Una computazione si dice *deterministica* se la sequenza di stati è completamente determinata da S_0, ovvero:

$$\forall\, i \,:\, S_i \rightarrow S_{i+1}$$

Il simbolo $\rightarrow$ in questo ambito viene interpretato come "*lo stato* S_{i+1} *segue lo stato* S_i". Una coppia $< S_i, S_h >$ tale che $S_i \rightarrow S_h$ viene detto *Passo di Computazione*. In caso la computazione sia finita avremo uno stato ultimo al quale non ne seguiranno altri. Quindi $\rightarrow$ è la relazione che stabilisce l'ordine in cui vengono attraversati i vari stati di computazione.

Una computazione si dice *non-deterministica* se dato uno stato S_i e k distinti stati $S_{i,1}$, ... , $S_{i,k}$:

$$\exists\ S_{i+1} \in \{\ S_{i,1},\ \ldots\ ,\ S_{i,k}\ \}\ .\ S_i\ \rightarrow\ S_{i+1}$$

Questo significa che dato un certo stato iniziale S_0, la sequenza di passi di computazione $\rightarrow$ può non essere unica.

Di fatto un *Algoritmo non-deterministico* è un algoritmo che, in caso di fallimento dell'esecuzione, può tornare indietro alla ricerca di un punto di scelta (*Choice Point*) da cui provare un altro percorso.
Per esempio in Prolog il non-determinismo è implementato nella scelta della clausola tramite *backtracking* cronologico.
In **JSetL** invece vengono messi a disposizione gli strumenti necessari per implementare uno speciale costrutto astratto ***either-orelse*** il quale permette una scelta non-deterministica fra più possibilità. Immaginando che $St_1, St_2, ..., St_n$ siano n diversi *statements*, l'applicazione del costrutto:

$$\textbf{\textit{either}}\ St_1\ \textbf{\textit{orelse}}\ St_2\ \textbf{\textit{orelse}}\ \ldots\ \textbf{\textit{orelse}}\ St_n$$

rappresenta la disgiunzione logica $St_1 \vee St_2 \vee \cdots \vee St_n$, mentre da un punto di vista computazionale è interpretabile come l'esplorazione di tutte le possibili alternative St_1, St_2, ... , St_n, a partire da St_1, applicando il *backtracking* in caso di fallimento.

2.1.1 Esempi

Supponiamo di volere controllare se un elemento appartiene o meno ad una lista. Questa operazione è implementabile in Prolog come segue:

```
member(List, Element) :-
  List = [Element|_Rest].
member(List, Element) :-
  List = [_First|Rest],
  member(Rest, Element).
```

`member(List, Element)` è vero se `Element` appartiene a `List`. Il predicato è interpretabile come "`Element` appartiene a `List` se è il primo elemento di `List` o se appartiene alla coda di `List`".

Data una lista `L = [a, b, c, d]`, il *goal* `member(L, c)` viene risolto come segue:

```
member([a, b, c, d], c)
  member([b, c, d], c)
    member([c, d], c)
      true.
```

Questa implementazione risulta vantaggiosa anche per un altro motivo: supponiamo, data una lista, di volere uno qualsiasi degli elementi che ne fanno parte. Il *goal* `member([a, b, c, d], X)` viene risolto come segue.:

```
?- member([a, b, c, d], X).
  X = a ;
  X = b ;
  X = c ;
  X = d ;
  false.
```

`member([a, b, c, d], X)` calcola in modo non-deterministico una possibile soluzione per `X`; in caso di successivo fallimento è in grado di trovare, se esiste, un'altra soluzione per `X`.

Immaginiamo di volere ora un predicato che svolga l'operazione di concatenazione fra due liste:

```
concat(List_1, List_2, List_3) :-
  List_1 = [],
  List_3 = List_2.
concat(List_1, List_2, List_3) :-
  List_1 = [First|Tail_1],
  List_3 = [First|Tail_3],
  concat(Tail_1, List_2, Tail_3).
```

`concat(List_1, List_2, List_3)` è vera se la concatenazione delle liste `List_1` e `List_2` è la lista `List_3`. Il predicato `concat` qui descritto è utilizzabile anche per trovare tutte le possibili suddivisioni di una lista:

```
?- concat(List_1,List_2,[a,b,c]).
List_1 = [],
List_2 = [a, b, c] ;
List_1 = [a],
```

```
List_2 = [b, c] ;
List_1 = [a, b],
List_2 = [c] ;
List_1 = [a, b, c],
List_2 = [] ;
false.
```

`concat` sfrutta il non-determinismo nella scelta fra le due clausole.

2.2 La libreria JSetL

JSetL è una libreria Java sviluppata presso il Dipartimento di Matematica dell'Università di Parma. Il suo scopo è quello di combinare il paradigma object-oriented di Java con alcuni concetti classici di linguaggi dichiarativi come il Prolog, quali ad esempio variabili logiche, liste parzialmente specificate, unificazione, risoluzione di vincoli e non-determinismo.
La libreria è un software libero, re-distribuibile e modificabile sotto i termini della licenza GNU. La versione corrente è JSetL 2.3.
Le caratteristiche della libreria che ci interessano sono:

- le variabile logiche;
- le variabile logiche intere;
- le liste logiche;
- i vincoli;
- il risolutore dei vincoli.

Inoltre daremo uno sguardo approfondito alla classe `NewConstraintsClass` nella sezione 2.3.

2.2.1 Variabili logiche

Una *variabile logica* è un'istanza della classe `LVar`. Questa classe fornisce costruttori per creare variabili logiche ed una serie di semplici metodi per manipolarle e testarle, oltre ad alcuni vincoli di base. Possiamo associare ad una `LVar` un valore (`LVar-`*value*) ed un nome esterno. Quando ad una variabile logica viene associato un `Lvar-`*value*, la variabile è detta *bound*; in caso contrario è detta *unbound*. Esponiamo rapidamente i metodi di `LVar` che ci interessano:

Costruttori

`LVar()`
`LVar(String extName)`

Crea una variabile logica *unbound*; `extName`, se specificato, viene assegnato come nome esterno alla variabile, in caso contrario le viene assegnato automaticamente il nome esterno di default "?";

`LVar(Object o)`
`LVar(String extName, Object o)`

Crea una variabile logica *bound* assegnandole `o` come `LVar`-*value*; `extName`, se specificato, viene assegnato come nome esterno alla variabile, in caso contrario le viene assegnato automaticamente il nome esterno di default "?";

Diciamo che due variabili logiche `x` e `y` sono equivalenti se sono state unificate con successo (`x.eq(y)`).

Metodi

`LVar setName(String extName)`

Assegna a questa variabile il nome esterno `extName` e ritorna la variabile stessa;

`String getName()`

Restituisce il nome esterno associato a questa variabile logica;

`Object getValue()`

Se la variabile è *bound* restituisce il valore (`LVar`-*value*) assegnato a questa variabile logica, `null` altrimenti;

`boolean isBound()`

Se la variabile è *bound* restituisce `true`;

`void output()`

Stampa in maniera leggibile il nome esterno della variabile seguito dal suo valore; nel caso la variabile sia *unbound* al posto del valore stampa la stringa "`unknown`";

`LVar getVar(String extName)`

Restituisce la variabile `LVar` con nome esterno `extName` (spiegato meglio in 4.5.2);

`Constraint eq(Object o)`

Ritorna il vincolo `this = o` che unifica la variabile logica con l'oggetto `o`.

2.2.2 Variabili logiche intere

Una variabile logica intera è un'istanza della classe `IntLVar`. Questa classe estende la classe `LVar` ed è usata per rappresentare i valori interi. Una variabile `IntLVar` ha un dominio finito ed un vincolo aritmetico, eventualmente vuoto, associato ad essa. Espandendo la classe `LVar` mantiene i suoi metodi e ne mette altri a disposizione. Il vincolo associato ad una variabile viene risolto insieme agli altri vincoli presenti nel *constraint store* nel caso la variabile sia coinvolta nella risoluzione del problema. Presentiamo solo quelli di nostro interesse:

Costruttori

Mantiene i costruttori della classe `LVar` per dichiarare variabili *unbound* e aggiunge i seguenti:

`LVar(Integer k)`
`LVar(String extName, Integer k)`

Crea una variabile logica intera *bound* assegnandole il valore `k`; `extName`, se specificato, viene assegnato come nome esterno alla variabile, in caso contrario le viene assegnato automaticamente il nome esterno di default "?";

`LVar(Integer a, Integer b)`
`LVar(String extName, Integer a, Integer b)`

Crea una variabile logica intera *unbound* restingendo il suo dominio fra `a` e `b`; `extName`, se specificato, viene assegnato come nome esterno alla variabile, in caso contrario le viene assegnato automaticamente il nome esterno di default "?".

Metodi

Mantiene i metodi presentati per la classe `LVar`. È possibile creare variabili `IntLVar` con vincoli aritmetici utilizzando i metodi per operazioni aritmetiche. Questi metodi vengono invocati da un oggetto di tipo `IntLVar` e restituiscono un `IntLVar`; per questo è possibile concatenarli per formare operazioni aritmetiche più complesse. I metodi in questione sono:

`IntLVar sum(Integer k)`
`IntLVar sum(IntLVar k)`

Ritorna una variabile logica intera $\texttt{X}_1$ associata al vincolo $\texttt{X}_1 = \texttt{X}_0 + k \wedge C_0$, dove $\texttt{X}_0$ è la variabile di invocazione, C_0 sono i vincoli a lei associata e il parametro k è l'intero o il valore dell'oggetto `IntLVar` passato al metodo;

`IntLVar sub(Integer k)`
`IntLVar sub(IntLVar k)`

Ritorna una variabile logica intera $\texttt{X}_1$ associata al vincolo $\texttt{X}_1$ `=` $\texttt{X}_0$ `-` $k \wedge C_0$, dove $\texttt{X}_0$ è la variabile di invocazione, C_0 sono i vincoli a lei associata e il parametro k è l'intero o il valore dell'oggetto `IntLVar` passato al metodo;

`IntLVar mul(Integer k)`
`IntLVar mul(IntLVar k)`

Ritorna una variabile logica intera $\texttt{X}_1$ associata al vincolo $\texttt{X}_1$ `=` $\texttt{X}_0$ `*` $k \wedge C_0$, dove $\texttt{X}_0$ è la variabile di invocazione, C_0 sono i vincoli a lei associata e il parametro k è l'intero o il valore dell'oggetto `IntLVar` passato al metodo;

`IntLVar div(Integer k)`
`IntLVar div(IntLVar k)`

Ritorna una variabile logica intera $\texttt{X}_1$ associata al vincolo $\texttt{X}_1$ `=` $\texttt{X}_0$ `/` $k \wedge C_0 \wedge k \neq 0$, dove $\texttt{X}_0$ è la variabile di invocazione, C_0 sono i vincoli a lei associata e il parametro k è l'intero o il valore dell'oggetto `IntLVar` passato al metodo[1];

`void setConstraint(Constraint c)`

Associa il `Constraint c` alla variabile di invocazione.

2.2.3 Liste logiche

Una lista logica è un caso speciale di variabile logica il cui valore è una coppia di elementi `<elems, rest>` dove `elems` è una lista `[`$\texttt{e}_0$`, ... ,` $\texttt{e}_n$`]`, con $n \geq 0$, di oggetti ti tipo arbitrario, e `rest` (detto anche *coda*) è una lista vuota o *unbound*. In **JSetL** una lista logica è un'istanza della classe `LList`, che estende la classe `LCollection` (che qui non trattiamo); questa classe permette di rappresentare anche liste parzialmente specificate. Pur non estendendola, la classe `LVar` e `LList` presentano molti metodi e caratteristiche comuni; è per esempio possibile associare loro un nome esterno, proprio come agli oggetti `LVar`. Osserviamo i metodi di nostro interesse della classe `LList`:

Costruttori

I costruttori della classe `LList` hanno la stessa forma di quelli già presentati per `LVar`. Sono presenti altri costruttori specifici per questa classe (è per

[1] I vincoli creati per associare l'operazione di divisione qui descritta rappresentano l'operazione esattamente opposta della moltiplicazione; nel caso si volesse eseguire una divisione intera con troncamento è necessario utilizzare non questo metodo, bensì il metodo `truncDiv`, che qui non viene descritto.

esempio possibile passare al costruttore come parametro una Java `List<?>`), ma nel nostro caso non vengono utilizzati.

Metodi

Molti dei metodi visti per `LVar` sono presenti anche per `LList`. Osserviamo i metodi specifici di `LList` di nostro interesse:

`static LList empty()`
: Ritorna la lista logica vuota;

`LList ins1(Object o)`
: Ritorna una lista logica avente `o` come primo elemento della lista e la lista di invocazione come coda; la lista di ritorno sarà *bound* o *unbound* a seconda della variabile di invocazione;

`LList insn(Object o)`
: Come `ins1`, ma `o` è aggiunto come ultimo elemento;

`Object get(int i)`
: Se la lista è *bound* restituisce l'i-esimo elemento della lista, altrimenti solleva un eccezione `NotInitVarException`;

`int getSize()`
: Se la lista è *bound* restituisce il numero di elementi che contiene, altrimenti solleva un eccezione `NotInitVarException`.

2.2.4 Vincoli

I vincoli rappresentano operazioni applicabili alle variabili logiche presentate prima. Queste operazioni possono essere eseguite anche se gli oggetti logici coinvolti non hanno un valore preciso (variabili *unbound*). In **JSetL** un vincolo è un'istanza della classe `Constraint`. Metodi di altre classi, come per esempio `eq(...)` di `LVar`, generano oggetti di tipo `Constraint`. Osserviamo alcuni metodi di nostro interesse di questa classe:

Costruttori

Mantiene i costruttori della classe `LVar` per dichiarare variabili *unbound* e aggiunge i seguenti:

`Constraint()`
Crea il vincolo vuoto (con nome di default "`no name`");

```
Constraint(String extName, Object o1)
Constraint(String extName, Object o1, Object o2)
Constraint(String extName, Object o1, Object o2, Object o3)
Constraint(String extName, Object o1, Object o2, Object o4, Object o4)
```
Crea un vincolo di nome `extName` e da uno a quattro argomenti `o1, o2, o3, o4`[2].

Metodi

`int getArg(int i)`
Restituisce l'i-esimo argomento del vincolo;

`Constraint and(Constraint c)`
Restituisce un `Constraint` che specifica la congiunzione fra il vincolo di invocazione e il `Constraint c`;

`void notSolved(Constraint c)`
Pone questo vincolo a "*unsolved*";

`void fail()`
Causa il fallimento del vincolo di invocazione; questo causa il *backtracking* alla ricerca dell'ultimo *choice point* (spiegato meglio in 2.4);

`int getAlternative()`
Ritorna un contatore interno al vincolo il cui valore non viene ripristinato in caso di *backtracking* (spiegato meglio in 2.4).

2.2.5 Risolutore di vincoli

I vincoli definiti tramite oggetti di tipo `Constraint` vengono risolti da un risolutore di vincoli (*constraint solver*). In **JSetL** questo viene creato come istanza della classe `SolverClass`. Questa classe fornisce i metodi per aggiungere `Constraint` alla collezione attuale di vincoli (*constraint store*), per controllarne la soddisfacibilità e per trovare tutte le soluzioni dei vincoli memorizzati. D'ora in poi ci riferiremo al risolutore di vincoli anche con il

[2] La versione attuale di **JSetL** permette di definire solo vincoli con al massimo quattro argomenti; vincoli con più argomenti possono essere realizzati utilizzando vettori o liste di argomenti, come vedremo successivamente in alcuni esempi.

termine `solver`. Osserviamo quali sono i metodi che ci interessano fra quelli offerti da questa classe:

Costruttori

`SolverClass()`

Crea un oggetto di tipo `SolverClass`; il *constraint store* del `solver` è inizialmente vuoto.

Metodi

`void add(Constraint c)`

Aggiunge il `Constraint c` al *constraint store* del `solver`; il vincolo aggiunto non viene ancora processato;

`void clearStore()`

Rimuove tutti i vincoli collezionati nel *constraint store* e tutti i punti di scelta (*choice points*) ad esso associati;

`void addChoicePoint(Constraint c)`

Aggiunge un punto di scelta associato al `Constraint c` (spiegato meglio in 2.4);

`boolean check(Constraint c)`
`boolean check()`

Valuta il vincolo $C \wedge$`c`, dove C rappresenta la collezione di vincoli attualmente nel *constraint store*, e restituisce `true` o `false` a seconda che il vincolo sia soddisfacibile o meno; se il vincolo è soddisfacibile il vincolo viene anche processato; se `c` non è specificato viene valutata solamente la collezione di vincoli nel *constraint store*;

`void solve(Constraint c)`
`void solve()`

Esattamente come `check(c)` e `check()`, ma nel caso il vincolo non sia soddisfacibile viene sollevata un'eccezione `Failure`.

2.3 NewConstraintsClass

JSetL consente di definire nuovi vincoli, eventualmente non-deterministici. Ciò si ottiene estendendo la classe astratta `NewConstraintsClass`.
Per esempio:

```
public class MyConstraints extends NewConstraintsClass {

  public MyConstraints(SolverClass currentSolver) {
    super(currentSolver);
  }

  // ...

}
```

è usato per definire una collezione di nuovi vincoli contenenti operazioni definite dall'utente. Il costruttore prende come argomento un *solver*; questo dovrà essere il risolutore utilizzato per risolvere i vincoli definiti in `MyConstraints`[3].
Per aggiungere vincoli contenuti in `MyConstraints` al *constraint store* dovremo prima creare un oggetto `MyConstraints`. Una volta fatto ciò si potranno utilizzare i metodi già descritti della classe `SolverClass` per aggiungere e risolvere vincoli.
Supponendo per esempio di avere il vincolo `constr1(v1, v2)`, con `v1, v2` variabili, definito in `MyConstraints` e di volerlo aggiungere al constraint store del solver `mySolver`, scriveremmo il seguente frammento di codice:

```
// ...
MyConstraints myConstr = new MyConstraints(mySolver);
mySolver.add(myConstr.constr1(v1, v2));
// ...
```

Richiamando il metodo `mySolver.solve()` potremo risolvere il vincolo espresso in `constr1(...)`.
L'implementazione di una nuova collezione di vincoli, ottenuta estendendo `NewConstraintsClass`, richiede di rispettare alcune convenzioni.

[3] La versione attuale di **JSetL** permette di assegnare ad un `solver` un'unica collezione di vincoli `NewConstraintsClass`, obbligando l'utente a definire tutti i vincoli in un'unica classe che espanda `NewConstraintsClass`; questo limite è in via di valutazione e verrà presumibilmente risolto in futuro.

2.3.1 Implementazione di un nuovo vincolo

Supponiamo di voler definire all'interno della classe `MyConstraints` i vincoli `constr1(v1, v2)` e `constr2(v1)`, con `v1` e `v2` variabili rispettivamente di tipo `t1` e `t2`; per prima cosa dobbiamo creare i metodi per restituire i vincoli da inserire nel *constraint store* del `solver`, quindi:

```
// ...
public Constraint constr1(t1 v1, t2 v2) {
  return new constraint("constr1", v1, v2);
}

public Constraint constr2(t1 v1) {
  return new constraint("constr2", v1);
}
// ...
```

Questi metodi servono per poter aggiungere i vincoli nel modo descritto prima e rappresentano l'interfaccia utente della classe. Una volta fatto ciò è necessario aggiungere i metodi che implementano effettivamente le operazioni che vogliamo includere in `MyConstraints`. Un'altra convenzione è che questi metodi debbano ricevere come unico argomento un oggetto di tipo `Constraint`. Quindi, ipotizzando che `implem1(c)` sia l'implementazione di `constr1(...)` e `implem2(c)` sia l'implementazione di `constr2(...)`, scriveremo:

```
protected void user_code(Constraint c)
throw Failure, NotDefConstraintException {
  if (c.getName() == "constr1")
    implem1(c);
  else if (c.getName() == "constr2")
    implem2(c);
  else throw new NotDefConstraintException();
}

private void implem1(Constraint c) {
  t1 x = (t1)c.getArg(1);
  t2 y = (t2)c.getArg(2);

  // ...
  // Implementazione del vincolo constr1 sugli oggetti x e y

  return;
}
```

```
private void implem2(Constraint c) {
  t1 x = (t1)c.getArg(1);

  // ...
  // Implementazione del vincolo constr2 sull'oggetto x

  return;
}
```

Il metodo `user_code(Constraint c)` ha il compito di associare il nome di un vincolo definito dall'utente con il relativo metodo da richiamare; questo metodo viene richiamato dal `Solver` in fase di risoluzione. Si può notare che gli argomenti dei metodi privati vengono ricavati direttamente dal `Constraint` passato utilizzando i metodi `getArg(`n`)`, dove n è l'indice dell'argomento richiesto. Di default il `Solver`, quando processa un vincolo definito dall'utente, pone il vincolo a "*solved*", ovvero risolto. Se questo non fosse ciò che si vuole (per esempio se si volesse lasciare il vincolo nel *constraint store* per attendere che altri vincoli vengano risolti prima di lui), l'utente deve specificarlo esplicitamente. Per fare ciò si utilizza il metodo `notSolved()` della classe `Constraint`, il quale pone il vincolo che l'ha invocato a "*unsolved*".

2.4 Sfruttare il non-determinismo

In un vincolo definito dall'utente si può implementare il non-determinismo sfruttando quanto offre **JSetL**. In particolare ci interessano i seguenti metodi:

- `int` **getAlternative()** : metodo della classe `Constraint` il quale restituisce un intero associato al vincolo che l'ha invocato utilizzabile per contare le alternative non-deterministiche per quel dato vincolo; il suo valore iniziale è 0 e viene incrementato automaticamente quando il vincolo viene riprocessato a causa di *backtracking*;

- `void` **fail()** : metodo della classe `Constraint` il quale causa un fallimento relativo al vincolo chiamante; di conseguenza il *solver* inizia il *backtracking* alla ricerca del più vicino *choice point*; in caso non ve ne siano, viene decretato il fallimento e lanciata la relativa eccezione;

- `void` **addChoicePoint(**`Constraint c`**)** : aggiunge un *choice point* relativo al vincolo `c`; in caso di *backtracking* verrà ripristinato lo stato del vincolo a questo istante, con l'unica eccezione del contatore delle alternative interno al vincolo (il cui valore è leggibile tramite `c.getAlternative()`).

2.4.1 Esempi

Prendiamo per esempio i predicati `member` e `concat` presentati in precedenza e supponiamo di volerli implementare in **JSetL** all'interno della collezione `ListOps`. Iniziamo quindi la creazione della classe descrivendo l'interfaccia, i metodi che saranno richiamabili direttamente dagli utenti, per esprimere i predicati specificati:

```
import JSetL.*;

public class ListOps extends NewConstraintsClass{

  public ListOps(SolverClass solver) {
      super(solver);
  }

  // True if x is a member of l
  public Constraint member(LList l, LVar x) {
      return new Constraint("member", l, x);
  }

  // True if l3 is the concatenation of l1 and l2
  public Constraint concat(LList l1, LList l2, LList l3) {
          return new Constraint("concat",l1, l2, l3);
  }

  // ...
  // metodo user_code ed implementazione dei vincoli dichiarati

}
```

Osserviamo l'implementazione del predicato `member`:

```
private void member(Constraint c)
throws Failure {
  LList l = (LList)c.getArg(1);
  LVar z = (LVar)c.getArg(2);
  if (!l.isBound()) {
    c.notSolved();
    return;
  };  // irreducible case
  if (l.isEmpty())
    c.fail();

  // ...
  // casi non-deterministici per lista non vuota
}
```

Tramite chiamate a `getArg(`n`)` identifichiamo gli argomenti; `l` è la lista di elementi mentre `z` il singolo elemento. Nel caso la lista non sia definita (`if (!l.isBound())`) il vincolo viene posto a "*unsolved*"; questo serve nell'eventualità in cui il vincolo `member` sia processato prima dei vincoli che specificano la lista. Il controllo se la lista è vuota (`l.isEmpty()` è invece una semplice ottimizzazione. I casi in cui `l` è non vuota vengono trattati prendendo la scelta non-deterministica nel modo seguente:

```
  // member(List, Element) :-
  switch(c.getAlternative()) {
    case 0: {
      // List = [Element|_Rest].

      Solver.addChoicePoint(c);
      LList r = new LList();
      Solver.add(l.eq(r.ins1(z)));
      break;
    }
    case 1: {
      // List = [_First|Rest], member(Rest, Element).

      LList r = new LList();
      Solver.add(l.eq(r.ins1(new LVar())));
      Solver.add(member(r,z));
      break;
    }
  }
  return;
}
```

Ogni ramo del costrutto `switch` è un percorso alternativo:

- `case 0:` aggiunge i vincoli per controllare se `z` è il primo elemento della lista;
- `case 1:` applica il vincolo `member` alla coda della lista.

La lettura del codice non è intuitiva come in Prolog ma il significato è lo stesso.

Il predicato `concat` è invece implementabile nel modo seguente:

```
private void concat(Constraint c)
throws Failure {
  LList l1    = (LList)c.getArg(1);
  LList l2    = (LList)c.getArg(2);
  LList l3    = (LList)c.getArg(3);
```

```
  // concat(List_1, List_2, List_3) :-
  switch(c.getAlternative()) {
    case 0: {
      // List_1 = [], List_3 = List_2.

      Solver.addChoicePoint(c);
      Solver.add(l1.eq(LList.empty()));
      Solver.add(l2.eq(l3));
      break;
    }
    case 1: {
      // List_1 = [First|Tail_1], List_3 = [First|Tail_3],
      // concat(Tail_1, List_2, Tail_3).

      LVar  x = new LVar();
      LList l1_aux = new LList();
      LList l3_aux = new LList();
      Solver.add(l1.eq(l1_aux.ins1(x)));
      Solver.add(l3.eq(l3_aux.ins1(x)));
      Solver.add(concat(l1_aux,l2,l3_aux));
      break;
    }
  }
}
```

Anche in questo caso ogni ramo del costrutto `switch` rappresenta un percorso alternativo:

- `case 0:` aggiunge il vincolo "`l1` è vuota e `l3` è uguale a `l2`";
- `case 1:` aggiunge i vincoli necessari alla ricorsione, ovvero esplicita che "`l1` e `l3` hanno la stessa testa e la coda di `l3` equivale alla concatenazione della coda di `l1` con la lista `l2`".

L'ultimo passaggio per implementare correttamente `ListOps` è la semplice aggiunta del metodo `user_code`:

```
protected void user_code(Constraint c)
throws Failure, NotDefConstraintException {
  if (c.getName() == "member")
    member(c);
  else if(c.getName() == "concat")
    concat(c);
  else
    throw new NotDefConstraintException();
}
```

Capitolo 3

DCG in JSetL

Nel primo capitolo abbiamo detto che le produzioni di una **DCG** sono direttamente associabili a clausole definite; quindi abbiamo mostrato che ogni produzione è traducibile in predicati Prolog. Abbiamo inoltre visto come in **JSetL** vengono trattati i vincoli definiti dagli utenti, equivalenti ai predicati definiti dagli utenti in un programma Prolog. Quello che vogliamo fare ora è sfruttare la classe `NewConstraintsClass` presentata nel capitolo precedente per implementare una grammatica.
Mostreremo ora una serie di **DCG** di esempio definite in Prolog, prima in notazione *implicita* ed in seguito in notazione *esplicita*, e l'implementazione equivalente in **JSetL**.

Nota. Il linguaggio Prolog richiede che le variabili siano denotate con identificatori che iniziano con lettera maiuscola. Durante l'implementazione in **JSetL** si cercherà, dove possibile, di mantenere lo stesso nome della variabile corrispondente nella clausola Prolog, scritta in minuscolo. Quindi le variabili Prolog `X`, `S0` e `Parse_tree` e le rispettive variabili Java `x`, `s0` e `parse_tree` rappresentano le stesse variabili nelle diverse implementazioni.

3.1 Derivazioni semplici

Se un simbolo non-terminale ha un'unica produzione la derivazione attraversa un unico percorso possibile, perciò tale derivazione è *deterministica*.
Consideriamo una singola produzione di una **DCG** senza argomenti extra, da simbolo non-terminale a due simboli non-terminali:

```
testa -->
  corpo_1, corpo_2.
```

Questa sappiamo essere equivalente alla clausola Prolog:

```
testa(S0, S) :-
  corpo_1(S0, S1),
  corpo_2(S1, S).
```

Osservando gli accorgimenti elencati in 2.3 ed immaginando di avere già definito i metodi relativi a `corpo_1` e `corpo_2`, dobbiamo aggiungere nella nostra classe (la quale estende `NewConstraintsClass`) i metodi:

```
public Constraint testa(LList s0, LList s) throws Failure {
  return new Constraint("testa", s0, s);
}

// ...

private void testa(Constraint c) throws Failure {
  LList s0 = (LList)c.getArg(1);
  LList s = (LList)c.getArg(2);

  LList s1 = new LList();

  Solver.add(corpo_1(s0, s1));
  Solver.add(corpo_2(s1, s));
  return;
}
```

con relativa aggiunta al metodo *user_ code*:

```
  // ...

  [else] if (c.getName() == "testa")
    testa(c);

  // ...
```

Ovviamente il metodo qui esposto mantiene invariati gli identificativi dei vincoli e dei metodi per maggiore comprensibilità, non per reale necessità di linguaggio.
Ogni traduzione in **JSetL** di una produzione è suddivisibile in tre parti principali:

- identificazione degli argomenti, chiamando i metodi `getArg(...)`;

- creazione di variabili logiche di supporto;
- aggiunta dei vincoli necessari per soddisfare la clausola.

Prendiamo ora una produzione da non-terminale a due terminali:

```
testa -->
  [a, b].
```

Traducendo in notazione *esplicita* questa produzione otteniamo:

```
testa(S0, S) :-
  S0 = [a, b|S].
```

La rappresentazione con *Difference Lists* spezza la produzione in una lista con in testa i terminali ed in coda la lista `S`. L'equivalente **JSetL** è:

```
public Constraint testa(LList s0, LList s) throws Failure {
  return new Constraint("testa", s0, s);
}

[ ... ]

private void testa(Constraint c) throws Failure {
  LList s0 = (LList)c.getArg(1);
  LList s = (LList)c.getArg(2);

  Solver.add(s0.eq(s.ins1('b').ins1('a')));
  return;
}
```

con relativa aggiunta al metodo *user_code*.
Come spiegato in 2.2.3, la chiamata `s.ins1(x)` restituisce una `LList` il cui primo valore è `x` e la cui coda è la lista `s`. La concatenazione delle chiamate al metodo `ins1(...)` permette di scrivere il vincolo di cui sopra più rapidamente, avendo l'accortezza di chiamare i metodi nel giusto ordine: `Solver.add(s0.eq(s.ins1('b').ins1('a')))` aggiunge al *constraint store* il vincolo `S0 = [a,b|S]`.

I due casi presentati possono ovviamente essere combinati fra loro. Prendiamo per esempio la produzione seguente:

```
testa -->
  corpo_1, [a, b], corpo_2, [c], corpo_3.
```

Osserviamo la traduzione in notazione *esplicita*, più simile alla traduzione **JSetL**:

```
testa(S0, S) :-
  corpo_1(S0, S1),
  S1 = [a, b|S2],
  corpo_2(S2, S3),
  S3 = [c|S4],
  corpo_3(S4, S).
```

Il procedimento per realizzare questa clausola in **JSetL** è lo stesso illustrato nei due esempi precedenti:

```
public Constraint testa(LList s0, LList s) throws Failure {
  return new Constraint("testa", s0, s);
}

// ...

private void testa(Constraint c) throws Failure {
  LList s0 = (LList)c.getArg(1);
  LList s = (LList)c.getArg(2);

  LList s2 = new LList();
  LList s3 = new LList();
  LList s4 = new LList();

  Solver.add(corpo_1(s0, s1));
  Solver.add(s1.eq(s2.ins1(b).ins1(a)));
  Solver.add(corpo_2(s2, s3));
  Solver.add(s3.eq(s4.ins1(c)));
  Solver.add(corpo_2(s4, s));
  return;
}
```

Questa è la struttura base di una produzione *deterministica*. La traduzione di produzioni *deterministiche* è pertanto un procedimento diretto che mantiene molte similarità con la notazione *esplicita* delle **DCG** in Prolog.

3.2 Derivazioni non-deterministiche

Abbiamo detto che se un simbolo non-terminale ha un'unica regola di produzione, la derivazione è *deterministica*. Osserviamo ora il caso in cui un simbolo non-terminale abbia più di una produzione.
Consideriamo il seguente esempio:

```
testa -->
  ( corpo_1
  | corpo_2, [a]
  | [b], corpo_3
  ).
```

Vi sono tre possibili scelte per il non-terminale `testa`, pertanto il processo di derivazione è *non-deterministico*. L'esempio qui proposto, tradotto in notazione *esplicita* diventa:

```
testa(S0, S) :-
  corpo_1(S0, S).
testa(S0, S) :-
  corpo_2(S0, S1), S1 = [a|S].
testa(S0, S) :-
  S0 = [b|S1], corpo_3(S1, S).
```

Nell'implementazione **JSetL** vogliamo poter effettuare *backtracking* in caso di fallimento per provare un'alternativa. Per realizzare ciò utilizzeremo la tecnica presentata in 2.4. Iniziamo costruendo l'interfaccia utente:

```
private Constraint testa(LList s0, LList s) throws Failure {
  return new Constraint("testa", s0, s);
}
```

Osserviamo ora le tre parti principali, introdotte in precedenza, in cui è suddivisibile la traduzione in **JSetL** di una produzione. Identifichiamo quindi gli argomenti tramite chiamate a `getArg(...)`:

```
public void testa(Constraint c) throws Failure {
  LList s0 = (LList)c.getArg(1);
  LList s = (LList)c.getArg(2);

  // ...
```

A questo punto creiamo eventuali variabili di supporto, a seconda del ramo dello `switch`, e aggiungiamo i vincoli necessari al *constraint store*:

```
    // ...

    // testa(S0, S) :-
    switch (c.getAlternative()) {
      case 0: {
        // corpo_1(S0, S).
        Solver.addChoicePoint(c);

        Solver.add(corpo_1(s0, s);
        break;
      }
      case 1: {
        // corpo_2(S0, S1), S1 = [a|S].
        Solver.addChoicePoint(c);

        LList s1 = new LList();

        Solver.add(corpo_2(s0, s1));
        Solver.add(s1.eq(s.ins1(a)));
        break;
      }
      case 2: {
        // S0 = [b|S1], corpo_3(S1, S).

        LList s1 = new LList();

        Solver.add(s0.eq(s1.ins1(b)));
        Solver.add(corpo_3(s1, s));
        break;
      }
    }
    return;
  }
```

Vi sarà poi da aggiungere al metodo `user_code` il relativo controllo sul nome del vincolo con chiamata al metodo corretto.

3.3 Produzioni con argomenti e Procedure Calls

Fra le caratteristiche delle **DCG** vi è quella di poter aggiungere argomenti ai simboli non-terminali. L'implementazione in **JSetL** di questa caratteristica risulta anch'essa direttamente equiparabile alla notazione *esplicita* delle

DCG in Prolog. Prendiamo per esempio la grammatica G avente le seguenti produzioni:

```
coppia(X) --->
  ( [X, X]
  | [X, X], coppia(X)
  ).
```

con `X` $\in \Sigma^{[1]}$. L'insieme di stringhe generate da questa grammatica è:

$$L(G) = \{\ x^{(2*n)} \mid x \in \Sigma\ \wedge n > 0\}$$

La notazione *esplicita* mostra come vengono aggiunti gli argomenti impliciti alle clausole:

```
coppia(X, S0, S) :-
  S0 = [X, X|S].
coppia(X, S0, S) :-
 S0 = [X, X|S1],
 coppia(X, S1, S).
```

Un primo problema dell'implementazione **JSetL** di questo predicato è rappresentato dal dover decidere il tipo del parametro passato. Il tipo dell'argomento è decidibile osservando due punti principali:

- stabilire se l'argomento di un non-terminale appartiene ad una tipologia specifica o meno: per esempio un predicato potrebbe dover essere richiamato da una certa clausola con argomento un carattere e da un'altra clausola con argomento un intero;
- stabilire se l'argomento è modificabile o meno: il *backtracking* di **JSetL** si applica alle sole classi specificate dalla libreria; se viene preso come argomento un oggetto ad esempio di tipo `String` con valore "*abc*" e questo viene modificato in "*abcde*", in caso di successivo fallimento e di *backtracking* il valore della stringa non viene ripristinato.

La soluzione per questi due problemi è unica: mappare gli argomenti utilizzando le classi specificate da **JSetL** il cui valore viene ripristinato in caso di *backtracking*.
La classe `LVar` per esempio è un contenitore per oggetti di qualsiasi tipo.
Osserviamo ora una possibile implementazione dell'esempio esposto prima:

[1] Σ di solito denota un alfabeto; nel nostro caso Σ rappresenta l'insieme contenente tutti i possibili simboli.

```
public Constraint coppia(LVar x, LList s0, LList s)
throws Failure {
  return new Constraint("coppia", x, s0, s);
}

// ... user_code

private void coppia(Constraint c) throws Failure {
  LVar x = (LVar)c.getArg(1);
  LList s0 = (LList)c.getArg(2);
  LList s = (LList)c.getArg(3);

  // coppia(X, S0, S) :-
  switch (c.getAlternative()) {
    case 0: {
      // S0 = [X, X|S].

      Solver.addChoicePoint(c);

      Solver.add(s0.eq(s.ins1(x).ins1(x)));
      break;
    }
    case 1: {
      // S0 = [X, X|S1], coppia(X, S1, S).

      LList s1 = new LList();

      Solver.add(s0.eq(s1.ins1(x).ins1(x)));
      Solver.add(coppia(s1, s));
      break;
    }
  }
  return;
}
```

In questo esempio l'argomento passato rappresenta il simbolo da utilizzare per la generazione di una stringa. Non viene modificato durante tutto il processo, quindi non era necessario mapparlo su un oggetto di tipo `LVar`. Tuttavia questo non vale per tutte le grammatiche, in quanto l'argomento di un non-terminale di una **DCG** può essere sfruttato esattamente come in una qualsiasi clausola Prolog. Prendiamo per esempio la grammatica seguente:

```
albero_dati(nil) -->
  [nil].
albero_dati([Sx, Dx]) -->
  ['('], albero_dati(Sx), ['.'], albero_dati(Dx), [')'].
```

Questa grammatica descrive alberi binari con la seguente forma:

- `nil` è un `albero_dati`;
- `(D1.D2)` è un `albero_dati` se `D1` è un `albero_dati` e `D2` è un `albero_dati`.

La traduzione in forma esplicita di questa grammatica è la seguente:

```
albero_dati(nil, S0, S) :-
  S0 = [nil|S].
albero_dati([Sx, Dx], S0, S) :-
  S0 = ['('|S1],
  albero_dati(Sx, S1, S2),
  S2 = ['.'|S3],
  albero_dati(Dx, S3, S4),
  S4 = [')'|S].
```

In questo caso l'argomento viene unificato con una lista strutturata esattamente come il *parse tree* della stringa generata. Per esempio la stringa "`(nil.(nil.nil))`" data al predicato restituisce come output:

```
?- albero_dati(X,['(',nil,'.','(',nil,'.',nil,')',')'],[]).
X = [nil, [nil, nil]] .
```

In `X` possiamo osservare il *parse tree* relativo alla stringa specificata.
L'implementazione in **JSetL** di questa grammatica è la seguente:

```
public Constraint albero_dati(LVar x, LList s0, LList s)
throws Failure {
  return new Constraint("albero_dati", x, s0, s);
}

// ... user_code

private void albero_dati(Constraint c) throws Failure {
  LList x = (LList)c.getArg(1);
  LList s0 = (LList)c.getArg(2);
  LList s = (LList)c.getArg(3);

  switch (c.getAlternative()) {
    case 0: {
      // albero_dati(nil) --> [nil].

      Solver.addChoicePoint(c);
```

```
        Solver.add(x.eq("nil"));
        Solver.add(s0.eq(s1.ins1("nil")));
        break;
      }
      case 1: {
        // albero_dati([Sx, Dx]) -->
        // ['('], albero_dati(Sx), ['.'], albero_dati(Dx), [')'].

        LList s1 = new LList();
        LList s2 = new LList();
        LList s3 = new LList();
        LList s4 = new LList();
        LList sx = new LList();
        LList dx = new LList();

        Solver.add(x.eq(LList.empty().ins1(dx).ins1(sx)));
        Solver.add(s0.eq(s1.ins1('(')));
        Solver.add(albero_dati(sx, s1, s2));
        Solver.add(s2.eq(s3.ins1('.')));
        Solver.add(albero_dati(dx, s3, s4));
        Solver.add(s4.eq(s.ins1(')')));
        break;
      }
    }
    return;
  }
```

Il vincolo prende gli stessi tre argomenti visti nell'implementazione Prolog in notazione *esplicita*. I vincoli relativi agli argomenti vengono esplicitati ed aggiunti al *constraint store*; nella nostra implementazione il termine `nil` è sostituito dalla stringa **"`nil`"**. Il risultato è del tutto equivalente.

All'interno della produzione di una **DCG** è possibile inserire delle "*Procedure Calls*", ovvero vincoli extra da soddisfare da cui non derivano altri simboli. Prendiamo per esempio questa semplice grammatica:

$$
\begin{array}{lcl}
abb & \rightarrow & [a] \\
abb & \rightarrow & [b, b] \\
abb & \rightarrow & [a], abb \\
abb & \rightarrow & [b, b], abb
\end{array}
$$

Supponiamo ora di volere uno strumento per contare il numero di occorrenze di 'a' e di 'b'. Osserviamo la **DCG** seguente scritta in Prolog in notazione *implicita*:

```
abb(A, B) -->
  ( [a], {A = 1, B = 0}
  | [b, b], {A = 0, B = 2}
  | [a], abb(Tmp, B), {A is Tmp + 1}
  | [b, b], abb(A, Tmp), {B is Tmp + 2}
  ).
```

Gli argomenti `A` e `B` restituiscono rispettivamente il numero di occorrenze di '*a*' e di '*b*'. La stessa grammatica scritta in forma *esplicita* è:

```
abb(A, B, S0, S) :-
  S0 = [a|S],
  A = 1,
  B = 0.
abb(A, B, S0, S) :-
  S0 = [b, b|S],
  A = 0,
  B = 2.
abb(A, B, S0, S) :-
  S0 = [a|S1],
  abb(Tmp, B, S1, S),
  A is Tmp + 1.
abb(A, B, S0, S) :-
  S0 = [b, b|S1],
  abb(A, Tmp, S1, S),
  B is Tmp + 2.
```

Le *Procedure Calls* vengono aggiunte come *goals*; infatti la stringa denotata dalla grammatica viene vincolata ai `Constraint` che coinvolgono le liste `S0` e `S`.
Vediamo ora l'implementazione **JSetL**: per prima cosa dobbiamo mappare gli argomenti sulle strutture equivalenti della libreria. Per `A` e `B`, essendo semplici contatori, utilizzeremo la classe `IntLVar`. La struttura è la solita:

- dichiarazione dell'interfaccia utente;

```
public Constraint abb(IntLVar a, IntLVar b, LList s0, LList s)
throws Failure {
  return new Constraint("abb", a, b, s0, s);
}

// ...
```

- identificazione degli argomenti;

```
private void abb(Constraint c) throws Failure {
  IntLVar a = (LList)c.getArg(1);
  IntLVar b = (LList)c.getArg(1);
  LList s0 = (LList)c.getArg(2);
  LList s = (LList)c.getArg(3);

  // ...
```

- implementazione della produzione:

```
abb(A, B) -> [a], {A = 1, B = 0} ;
```

lo stesso codice, con poche modifiche, implementa la produzione:

```
abb(A, B) -> [b, b], {A = 0, B = 2} ;
```

```
  // ...

  // abb(A, B) -->
  switch (c.getAlternative()) {
    case 0: {
      // [a], {A = 1, B = 0}

      Solver.addChoicePoint(c);

      Solver.add(s0.eq(s.ins1('a')));
      Solver.add(a.eq(1));
      Solver.add(b.eq(0));
      break;
    }

  // ... case 1: [b, b], {A = 0, B = 2}
```

- implementazione della produzione:

```
abb(A, B) -> [a], abb(Tmp, B), {A is Tmp + 1} ;
```

lo stesso codice, con poche modifiche, implementa la produzione:

```
abb(A, B) -> [b, b], abb(A, Tmp), {B is Tmp + 1} ;
```

```
    // ...

    case 2: {
      // [a], abb(Tmp, B), {A is Tmp + 1}

      Solver.addChoicePoint(c);

      LList s1 = new LList();
      IntLVar tmp = new IntLVar();

      Solver.add(s0.eq(s1.ins1('a')));
      Solver.add(abb(tmp, b, s1, s));
      Solver.add(a.eq(tmp.sum(1)));
      break;
    }

    // ... case3: [b, b], abb(A, Tmp), {B is Tmp + 2}

  }
  return;
}
```

I vincoli specificati dalle *Procedure Calls* vengono aggiunti al *constraint store* allo stesso modo degli altri vincoli.

Osserviamo infine questa variante della grammatica precedentemente esposta:

```
inizio(a) -->
  [a], coppia(a).
inizio(X) -->
  {X \= a},
  coppia(X).

coppia(X) -->
  ( [X, X]
  | [X, X], coppia(X)
  ).
```

dove `inizio` è il simbolo iniziale.
Il linguaggio descritto da questa grammatica è:

$$L(G) = \{\ x^{(2*n)} \mid x \in \Sigma \setminus \{a\} \wedge n > 0\ \} \cup \{\ a^{(2*n)+1} \mid n > 0\ \}$$

Osserviamo la traduzione in notazione *esplicita*:

```
inizio(a, S0, S) :-
  S0 = [a|S1],
  coppia(a, S1, S).
inizio(X, S0, S) :-
  X \= a,
  coppia(X, S0, S).

coppia(X, S0, S) :-
  S0 = [X, X|S].
coppia(X, S0, S) :-
  S0 = [X, X|S1],
  coppia(X, S1, S).
```

L'implementazione **JSetL** del predicato `coppia` è la stessa di prima; un'implementazione del predicato `inizio` è la seguente:

```
private void inizio(Constraint c) throws Failure {
  LVar x = (LList)c.getArg(1);
  LList s0 = (LList)c.getArg(2);
  LList s = (LList)c.getArg(3);

  // inizio(X, S0, S) :-
  if ('a' == (Character)x.getValue()) {
    // {X = a}, S0 = [a|S1], coppia(a, S1, S).

    LList s1 = new LList();
    Solver.add(s0.eq(s1.ins1(x)));
    Solver.add(coppia(x, s1, s));
  }
  else {
    // X \= a, coppia(X, S0, S).

    Solver.add(coppia(x, s0, s));
  }
  return;
}
```

Si nota la mancanza del costrutto `switch` nonostante la produzioni di `inizio` siano due.
L'utilizzo del costrutto `if-else` di Java ci permette di stabilire il ramo corretto senza bisogno di aggiungere punti di scelta per il *backtracking*; questo perchè nonostante la presenza di due diverse produzioni per `inizio`, la derivazione è assolutamente *deterministica*, dipendendo dal valore di `x`.

3.4 Esempio: espressioni aritmetiche semplici

Prendiamo in considerazione la grammatica proposta in 1.4.2 e la sua scrittura in notazione *esplicita* in Prolog:

```
expr(X) -->
  ( num(Y), [+], expr(Z), {X is Y + Z}
  | num(Y), [-], expr(Z), {X is Y - Z}
  | num(Y), [/], expr(Z), {X is Y / Z}
  | num(Y), [*], expr(Z), {X is Y * Z}
  | num(X)
  ).

num(X) -->
  [X], {integer(X)}.
```

dove `expr` è il simbolo iniziale. La scrittura in notazione *esplicita* è la seguente:

```
expr(X, S0, S) :-
  num(Y, S0, S1),
  S1 = [+|S2],
  expr(Z, S2, S),
  X is Y + Z.

% ... qui ci sono gli operatori '-' e '/'

expr(X, S0, S) :-
  num(Y, S0, S1),
  S1 = [*|S2],
  expr(Z, S2, S),
  X is Y * Z.
expr(X, S0, S) :-
  num(X, S0, S).

num(X, S0, S) :-
  S0 = [X|S],
  integer(X).
```

Costruiamo per gradi l'implementazione **JSetL** equivalente.
Partiamo dallo scheletro della classe:

```
public class SimpleExpr extends NewConstraintsClass {

  SolverClass solverAux = new SolverClass();

  public SimpleExpr(SolverClass CurrentSolver) {
    super(CurrentSolver);
  }

  // ... tutti gli altri metodi verranno aggiunti qui

}
```

La classe, di nome `SimpleExpr`, estende `NewConstraintsClass`; il costruttore prende in input un oggetto di tipo `SolverClass` e richiama il costruttore di `NewConstraintsClass` passandogli il parametro ricevuto.
Da notare la dichiarazione di un altro `solver`, chiamato `solverAux`. Questo è un risolutore ausiliario il cui utilizzo verrà spiegato meglio in seguito.
I seguenti metodi rappresentano l'interfaccia utente e, quando richiamati, restituiscono i `Constraint` da aggiungere al *constraint solver*.

```
  public Constraint expr(IntLVar x, LList l1, LList l2)
  throws Failure {
    return new Constraint("expr", x, l1, l2);
  }

  public Constraint num(IntLVar x, LList l1, LList l2)
  throws Failure {
    return new Constraint("num", x, l1, l2);
  }
```

Il metodo `user_code` rappresenta l'interfaccia con cui il `solver` richiama i metodi da noi definiti necessari alla risoluzione:

```
  protected void user_code(Constraint c)
  throws Failure, NotDefConstraintException {
    if (c.getName() == "expr")
      expr(c);
    else if (c.getName() == "num")
      num(c);
    else
      throw new NotDefConstraintException();
    return;
  }
```

Come descritto negli esempi precedenti, la creazione di un metodo si può dividere in tre punti principali. Abbiamo quindi:

- identificazione degli argomenti;

```
private void expr(Constraint c) throws Failure {
  IntLVar x = (IntLVar)c.getArg(1);
  LList l1 = (LList)c.getArg(2);
  LList l2 = (LList)c.getArg(3);
```

- creazione di variabili logiche di supporto;

```
  // expr(X) -->
  switch (c.getAlternative()) {
    case 0: {
      // expr(Y), [+], expr(Z), {X is Y + Z}
      Solver.addChoicePoint(c);

      IntLVar y = new IntLVar();
      IntLVar z = new IntLVar();
      LList s1 = new LList();
      LList s2 = new LList();
```

- aggiunta dei vincoli e controlli necessari.

```
      Solver.add(num(y, l1, s1));
      Solver.add(s1.eq(s2.ins1('+')));
      Solver.add(expr(z, s2, s));
      Solver.add(x.eq(y.sum(z)));
      break;
    }
```

Gli ultimi due punti si reiterano per ogni ramo esplorabile:

```
    // ... un case differente per ogni operazione

    case 3: {
      // expr(Y), [*], expr(Z), {X is Y * Z}

      IntLVar y = new IntLVar();
      IntLVar z = new IntLVar();
      LList s1 = new LList();
      LList s2 = new LList();
```

```
        Solver.add(num(y, l1, s1));
        Solver.add(s1.eq(s2.ins1('*')));
        Solver.add(expr(z, s2, s));
        Solver.add(x.eq(y.mul(z)));
        break;
      }
      case 4: {
        // num(X)

        Solver.add(num(x, l1, l2));
        break;
      }
    }
    return;
  }
```

Le produzioni per gli operatori '-' e '/' sono state tralasciate in quanto del tutto simili a quelle di somma e prodotto.
In questa implementazione i vincoli relativi alle operazioni vengono aggiunti al *constraint store* del `solver` passato al costruttore di `SimpleExpr`.
L'implementazione della produzione di `num` è la seguente:

```
  // num(X) -> [X], {integer(X)}.
  private void num(Constraint c) throws Failure {
    IntLVar x = (IntLVar)c.getArg(1);
    LList l1 = (LList)c.getArg(2);
    LList l2 = (LList)c.getArg(3);
    if (!l1.isBound()) {
      c.notSolved();
      return;
    }

    LVar z = new LVar();
    solverAux.clearStore();

    if (solverAux.check(l1.eq(l2.ins1(z)).and(x.eq(z)))) {
      if (z.getValue() instanceof Integer) {
        return;
      }
    }
    c.fail();
  }
```

L'implementazione di `num` utilizza una struttura differente dalle precedenti. La produzione di `num` infatti è del tutto deterministica.

Si noti l'utilizzo di `solverAux`; questo `solver` è utilizzato per eseguire controlli ed unificazioni dirette sugli elementi senza aggiungerli al *constraint store* del `solver` principale. Guardando questo caso specifico:

`solverAux.check(l1.eq(l2.ins1(z)).and(x.eq(z)))`

è `true` se `l1` è una lista con testa `z` e coda `l2` e se `x` è uguale a `z`.

Notiamo inoltre l'implementazione di quello che in Prolog era il vincolo `integer(X)`: sfruttando i metodi Java possiamo vedere se il valore della variabile `z` è un'istanza di tipo `Integer` per ottenere il risultato che vogliamo. In caso `z` non sia un intero verrà richiamata `c.fail()`, la quale solleverà un fallimento.
Per utilizzare questa grammatica è necessario definire un oggetto di tipo `SimpleExpr` e aggiungere i `Constraint` da lei definiti al *constraint store*. Osserviamo in questo esempio la valutazione dell'espressione "$2 + 4 * 5$":

```
public static void main(String[] args) throws Failure {
  SolverClass Solver = new SolverClass();
  SimpleExpr simple_expr = new SimpleExpr(Solver);

  IntLVar x = new IntLVar("X");
  // expr = '2+4*5'
  LList l_expr =
    LList.empty().insn(2).insn('+').insn(4).insn('*').insn(5);

  Solver.solve(simple_expr.expr(x, l_expr, LList.empty()));
  x.output();
}
```

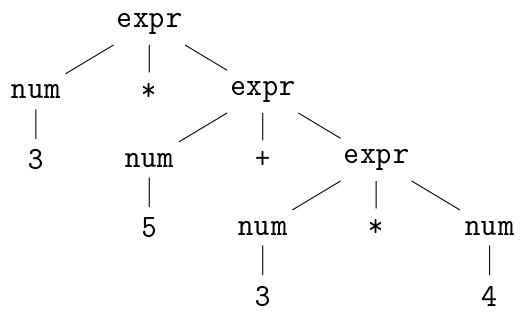
In questo codice dichiariamo una variabile logica intera `x` ed una lista logica `l_expr` rappresentante l'espressione da valutare. Ogni elemento della lista è un intero o un carattere rappresentante un operatore aritmetico. Possiamo inoltre notare la chiamata `LList.empty()`; questa permette di passare una lista vuota come parametro al metodo `simple_expr.expr(...)`.
La chiamata a `solve(...)` passando come parametro il `Constraint` creato dal metodo `simple_expr.expr(...)` permette la valutazione del vincolo da noi definito e l'output sarà "`X = 22`".

Si nota facilmente che questa grammatica non gestisce la priorità degli operatori.

Prendiamo in considerazione l'espressione "$3 * 5 + 3 * 4$" nell'esempio seguente:

```
public static void main(String[] args) throws Failure {
  SolverClass Solver = new SolverClass();
  SimpleExpr simple_expr = new SimpleExpr(Solver);

  IntLVar x = new IntLVar("X");
  // expr = '3*5+3*4'
  LList l_expr =
    LList.empty().insn(3).insn('*').insn(5).insn('+')
    .insn(3).insn('*').insn(4);

  Solver.solve(simple_expr.expr(x, l_expr, LList.empty()));
  x.output();
}
```

Il *parse tree* relativo all'espressione "$3 * 5 + 3 * 4$" è quindi il seguente:

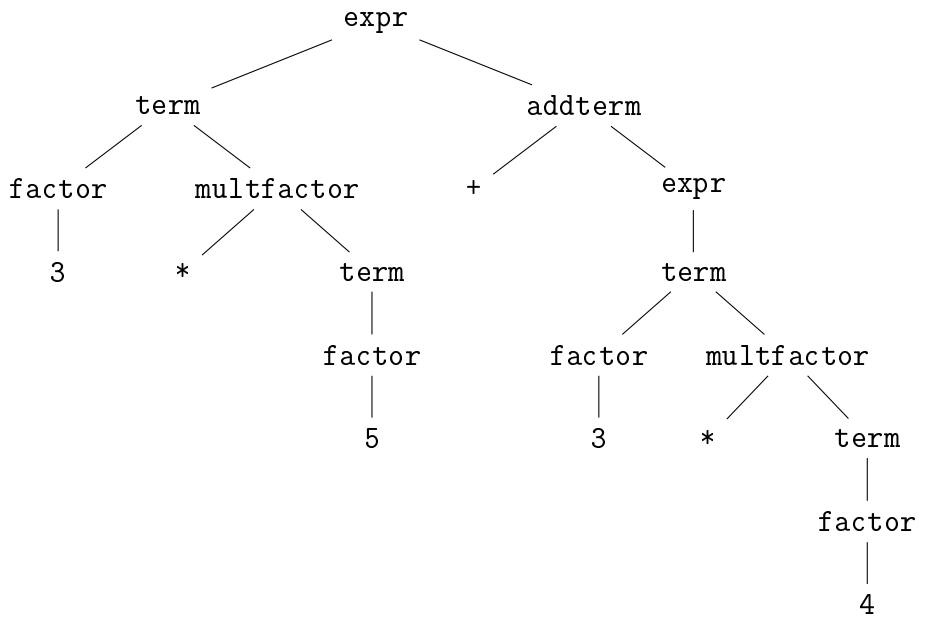


L'espressione valuta gli operatori da destra verso sinistra; l'espressione viene quindi interpretata logicamente come "$((3 * 5) + 3) * 4$", dando come output "`X = 72`".
Possiamo creare una variante della grammatica precedente in modo che venga gestita la priorità degli operatori nel modo seguente:

```
expr(X) -->
  ( term(X)
  | term(Y), addterm(X, Y)
  ).

addterm(X, Y) -->
  ( ['+'], expr(Z), {X is Y + Z}
  | ['-'], expr(Z), {X is Y - Z}
  ).
```

```
term(X) -->
  ( factor(X)
  | factor(Y), multfactor(X, Y)
  ).

multfactor(X, Y) -->
  ( ['*'], term(Z), {X is Y * Z}
  | ['/'], term(Z), {X is Y / Z}
  ).

factor(X) -->
  [X], {integer(X)}.
```

Seguendo questa grammatica, il *parse tree* dell'espressione "$3 * 5 + 3 * 4$" dell'esempio precedente è:



Questo *albero di derivazione* ci permette di interpretare l'espressione dando le giuste priorità.

Capitolo 4

Traduttore per espressioni aritmetiche

Scopo di questo capitolo è descrivere la realizzazione di un traduttore in **JSetL** utilizzando le **DCG**. Analizzeremo quindi come avviene un processo di traduzione a livello generale, descrivendo quali parti di questo processo sono di nostro interesse, e mostreremo l'implementazione di un semplice traduttore per espressioni aritmetiche.
In un secondo momento mostreremo anche in che modo concreto traduttori di questo tipo possono essere applicati al pacchetto **JSetL**.

4.1 Fasi di traduzione

Il processo di traduzione di un programma da un linguaggio ad un altro è chiamato *Compilazione*[8][7].
Indichiamo con P^L un programma scritto nel linguaggio L; P^L realizza una funzione parziale:

$$P^L : \mathbb{D} \rightarrow \mathbb{D}$$
$$P^L(Input) = Output$$

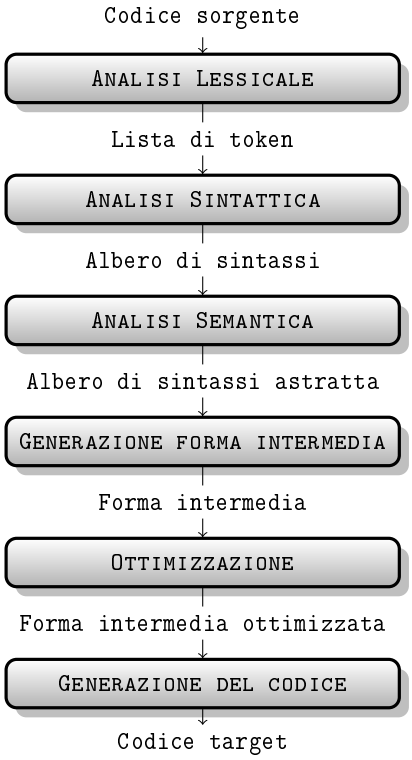
Indichiamo con $Prog^L$ l'insieme di tutti i possibili programmi scritti nel linguaggio L.
Formalmente un *Compilatore* C_{L_1,L_2} è un programma che implementa una funzione:

$$C_{L_1,L_2} : Prog^{L_1} \rightarrow Prog^{L_2}$$

tale che:

$$C_{L_1,L_2}(P^{L_1}) = P^{L_2} \rightarrow P^{L_1}(Input) = P^{L_2}(Input),\ \forall Input \in \mathbb{D}$$

Concettualmente, un *Compilatore* segue differenti fasi, ognuna delle quali trasforma il codice sorgente da una rappresentazione ad un'altra. Le fasi vengono solitamente schematizzate come segue:



Analizziamo le diverse fasi:

- Analisi Lessicale : è svolta da una funzione, o un programma, detta *Scanner*, il cui scopo è prendere in input una sequenza di caratteri e spezzarla in una sequenza di componenti sintattici primitivi detti *tokens*;

- Analisi Sintattica : è svolta dal *Parser*, il cui compito è prendere in input la sequenza di *tokens* generata dallo *Scanner*, controllarne la correttezza sintattica e costruire una struttura dati gerarchica rappresentante il *codice sorgente* (solitamente un *Albero di sintassi* o *Parse Tree*);

- Analisi Semantica : esegue controlli di semantica statica per rilevare eventuali errori e raggruppa le informazioni a proposito dei tipi utilizzati per la successiva generazione di codice; un importante compito di questa fase è il *type-checking*, in cui vengono analizzati gli operandi di

ogni operatore per controllare che siano permessi in quel contesto dalle specifiche del linguaggio; espande la struttura dati generata dal *Parser* (solitamente genera l'*Albero di sintassi astratta*);

- GENERAZIONE FORMA INTERMEDIA : alcuni compilatori generano una rappresentazione intermedia esplicita del codice sorgente; una rappresentazione intermedia deve avere due importanti caratteristiche: deve essere facile da produrre e molto facile da tradurre nel *codice target*;
- OTTIMIZZAZIONE : la fase di ottimizzazione modifica il codice intermedio o l'albero sintattico rappresentante un programma in modo da migliorarne l'efficienza;
- GENERAZIONE DEL CODICE : la fase finale del compilatore è la generazione del *codice target* a partire dal *codice sorgente* analizzato.

In particolar modo possiamo basarci sui concetti presentati di *Analisi Lessicale* e *Analisi Sintattica* per produrre un traduttore per espressioni aritmetiche. Quello che vogliamo ottenere sono tre classi che implementino tre delle fasi precedentemente descritte:

- una classe `ExprParser` che implementi un *Parser*;
- una classe `ExprTokenizer` che implementi uno *Scanner*;
- una classe `Translator` che richiami *Parser* e *Scanner* ed esegua la fase di *Generazione del Codice*.

4.2 Parser: la classe ExprParser

`ExprParser` descrive un *Parser* il cui scopo è valutare la lista di *token* restituita dallo *Scanner* e generarne una rappresentazione ad albero contenente i vincoli necessari ad esprimere l'espressione presa in input dal processo di traduzione.

Consideriamo questa **DCG** in notazione *implicita* Prolog:

```
expr(Z, Parse_tree) -->
  ( term(Z, Parse_tree)
  | term(Z1, P1), addterm([Z, Z1], P2), {Parse_tree = [P1, P2]}
  ).
```

```
addterm([Z, Z1], Parse_tree) -->
  ( ['+'], expr(Z2, P1), {Parse_tree = [P1, (Z is Z1 + Z2)]}
  | ['-'], expr(Z2, P1), {Parse_tree = [P1, (Z is Z1 - Z2)]}
  ).

term(Z, Parse_tree) -->
  ( factor(Z, Parse_tree)
  | factor(Z1, P1), multfactor([Z, Z1], P2), {Parse_tree = [P1, P2]}
  ).

multfactor([Z, Z1], Parse_tree) -->
  ( ['*'], term(Z2, P1), {Parse_tree = [P1, (Z is Z1 * Z2)]}
  | ['/'], term(Z2, P1), {Parse_tree = [P1, (Z is Z1 / Z2)]}
  ).

factor(Z, Parse_tree) -->
  ( [X], {integer(X), Parse_tree = [(Z = X)]}
  | [X], {variable(X), Parse_tree = [(Z = X)]}
  | ['('], expr(Z, Parse_tree), [')']
  ).
```

A differenza delle grammatiche presentate in precedenza si può notare che gli argomenti non sono più soltanto numeri interi, ma anche variabili. Il predicato `variable(V)` è vero se `V` è un identificatore valido per una variabile esistente. Per ora non è importante il come questo sia realizzato.
Vengono inoltre gestite le parentesi, permettendo di esplicitare la priorità degli operatori in un'espressione.
Tutte le produzioni presentano due argomenti:

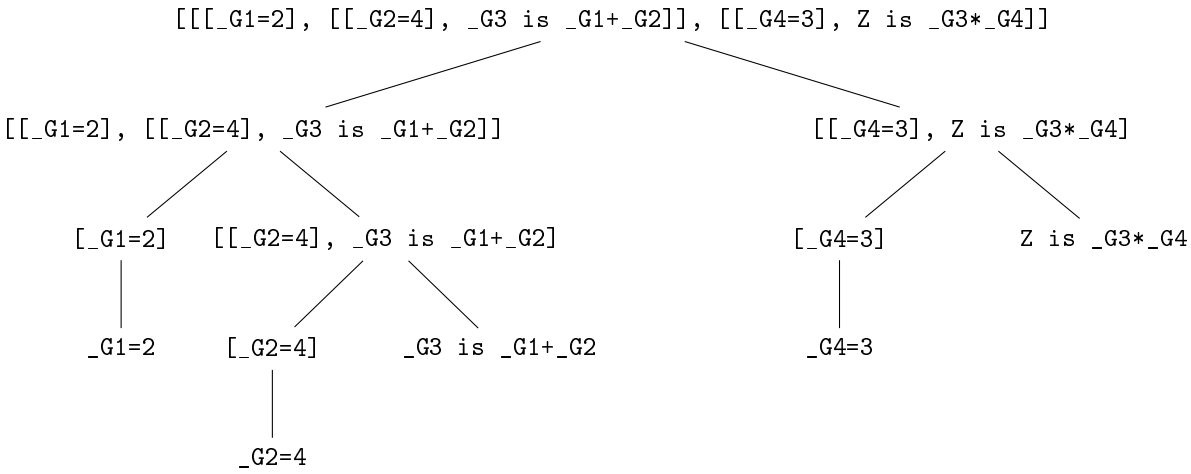
- `Z` o la coppia `[Z, Z1]` : `Z` e `Z1` sono variabili logiche intere utilizzate nella costruzione dei vincoli memorizzati all'interno di `Parse_tree`;
- `Parse_tree` : è una lista strutturata esattamente come l'albero di parsing della nostra espressione, seguendo le regole di precedenza specificate; ogni foglia di `Parse_tree` è un vincolo[1].

Consideriamo per esempio l'espressione "`(2+4)*3`"; questa genera la seguente lista per `Parse_tree`:

```
?- expr(Z, Parse_tree, ['(',2,'+',4,')','*',3], []).
Parse_tree = [[[_G1=2], [[_G2=4], _G3 is _G1+_G2]],
              [[_G4=3], Z is _G3*_G4]]
```

[1] Si noti che i vincoli presenti in `Parse_tree` non vengono valutati ma solo memorizzati.

`_G1`, `_G2`, `_G3` e `_G4` sono le variabili generate per descrivere i vincoli memorizzati in `Parse_tree`. Questa lista rappresenta l'albero:



La classe ideale per mapparla nell'implementazione **JSetL** è la `LList`.

Prendiamo ora un gruppo di produzioni per volta e traduciamolo in notazione *esplicita*:

```
expr(Z, Parse_tree, S0, S) :-
 term(Z, Parse_tree, S0, S).
expr(Z, Parse_tree, S0, S) :-
  term(Z1, P1, S0, S1),
  addterm([Z, Z1], P2, S1, S),
  Parse_tree = [P1, P2].
```

Il non-terminale `expr` ha due possibili produzioni. Vediamo come queste vengono implementate in **JSetL**. Dichiariamo ora l'interfaccia utente per richiamare il vincolo:

```
public Constraint expr(IntLVar z, LList parse_tree,
                       LList s0, LList s)
throws Failure {
  return new Constraint("expr", z, k, s0, s);
}
```

e implementiamo il vincolo seguendo i tre punti principali presentati in precedenza; identifichiamo gli argomenti:

```
// expr(Z, S0, S) :-
private void expr(Constraint c) throws Failure {
  IntLVar z = ((IntLVar)c.getArg(1));
  LList parse_tree = (LList)c.getArg(2);
  LList s0 = (LList)c.getArg(3);
  LList s = (LList)c.getArg(4);
  if (!s0.isBound()) {
    c.notSolved();
    return;
  }
```

e a seconda del ramo scelto creiamo variabili di supporto ed aggiungiamo i vincoli necessari:

```
  switch (c.getAlternative()) {
    case 0: {
      // term(Z, S0, S).
      Solver.addChoicePoint(c);

      Solver.add(term(z, parse_tree, s0, s));
      break;
    }
    case 1: {
      // term(Z1, P1, S0, S1), addterm([Z, Z1], P2, S1, S),
      // Parse_tree = [P1, P2].

      IntLVar z1 = new IntLVar();
      LList r = new LList();
      LList p1 = new LList();
      LList p2 = new LList();

      Solver.add(term(z1, p1, s0, r));
      LList pair = LList.empty().ins1(z1).ins1(z);
      Solver.add(addterm(pair, p2, r, s));
      Solver.add(parse_tree.eq(LList.empty().ins1(p2)
                               .ins1(p1)));
      break;
    }
  }
  return;
}
```

Possiamo notare come le produzioni di `term` siano del tutto simili nella forma a quelle di `expr`, pertanto la traduzione segue la stessa struttura.

Osserviamo ora la notazione *esplicita* delle produzioni di `addterm`:

```
addterm([Z, Z1], Parse_tree, S0, S) :-
 S0 = ['+'|S1],
 expr(Z2, P1, S1, S),
 Parse_tree = [P1, (Z is Z1 + Z2)].
addterm([Z, Z1], Parse_tree, S0, S) :-
 S0 = ['-'|S1],
 expr(Z2, P1, S1, S),
 Parse_tree = [P1, (Z is Z1 - Z2)].
```

Abbiamo detto che `Parse_tree` è un albero rappresentato come lista le cui foglie sono dei vincoli. Possiamo quindi rappresentare le foglie utilizzando la classe `Constraint` definita da **JSetL**. L'implementazione per il non-terminale `addterm` è la seguente:

```
public Constraint addterm(LList pair, LList parse_tree,
                          LList s0, LList s)
throws Failure {
  return new Constraint("addterm", pair, parse_tree, s0, s);
}

// ... user_code

private void addterm(Constraint c) throws Failure {
  LList pair = (LList)c.getArg(1);
  LList parse_tree = (LList)c.getArg(2);
  LList s0 = (LList)c.getArg(3);
  LList s = (LList)c.getArg(4);
  IntLVar z = (IntLVar)pair.get(0);
  IntLVar z1 = (IntLVar)pair.get(1);
  if (!s0.isBound()) {
      c.notSolved();
      return;
  }

  // ...
  // produzioni (switch)
```

Qui possiamo vedere la dichiarazione dell'interfaccia utente[2] e le chiamate a `getArg(`n`)` per ricavare gli argomenti. Notiamo inoltre l'utilizzo dei

[2] I metodi dell'interfaccia non devono essere dichiarati per forza `public`; `addterm` e `multfactor` possono per esempio essere dichiarati `private` se si desidera non sia possibile utilizzarli come simboli iniziali di una derivazione.

metodi `get(`n`)` della classe `LList` per ricavare la coppia di variabili `[Z, Z1]`, passata attraverso la `LList pair`.
Esplicitiamo ora le diverse produzioni attraverso il costrutto `switch`:

```
    // ...

    // addterm([Z, Z1], Parse_tree, S0, S) :-
    switch(c.getAlternative()) {
      case 0: {
        // S0 = ['+'|S1],
        // expr(Z2, P1, S1, S),
        // Parse_tree = [P1, (Z is Z1 + Z2)].
        Solver.addChoicePoint(c);

        LList p1 = new LList();
        LList s1 = new LList();
        IntLVar z2 = new IntLVar();

        Solver.add(s0.eq(s1.ins1('+')));
        Solver.add(expr(z2, p1, s1, s));
        Solver.add(parse_tree.eq(LList.empty().ins1(p1)
                                 .ins1(z.eq(z1.sum(z2)))));
        break;
      }
      case 1: {

        // ... del tutto simile al ramo 0

      }
    }
    return;
  }

  // ...
```

La `LList p1` deve essere unificata con il *parse tree* della sottoespressione `expr(z2, p1, s1, s)`; viene inoltre aggiunto l'equivalente del vincolo `Parse_tree = [P1, Z is Z1 + Z2]` in **JSetL** , ovvero:

```
Solver.add(parse_tree.eq(LList.empty().ins1(p1).
                         ins1(z.eq(z1.sum(z2)))));
```

L'implementazione delle produzioni del non-terminale `multfactor` è del tutto simile a quella presentata per le produzioni del non-terminale `addterm`. Osserviamo perciò l'ultimo non-terminale di nostro interesse:

```
factor(Z, Parse_tree, S0, S) :-
  S0 = [X|S],
  integer(X),
  Parse_tree = [(Z = X)].
factor(Z, Parse_tree, S0, S) :-
  S0 = [X|S],
  variable(X),
  Parse_tree = [(Z = X)].
factor(Z, Parse_tree, S0, S) :-
  S0 = ['('|S1], expr(Z, Parse_tree, S1, S2), S2 = [')'|S].
```

Queste produzioni indicano che `factor` può essere:

- un intero;
- una variabile;
- una sottoespressione racchiusa fra parentesi.

Osserviamo direttamente la parte che ci interessa dell'implementazione, ovvero i diversi rami rappresentanti le diverse produzioni:

```
// ...

switch(c.getAlternative()) {
  case 0: {
    // S0 = [X|S], integer(X), Parse_tree = [(Z = X)].
    Solver.addChoicePoint(c);

    LVar x = new LVar();
    Object o = s0.get(0);
    if (o instanceof LVar)
      o = ((LVar)o).getValue();

    if (o instanceof Integer) {
      Solver.add(s0.eq(s.ins1(x)));
      Solver.add(parse_tree.eq(LList.empty()
                               .ins1(z.eq(o))));
    }
    else
      c.fail();
    break;
  }
```

Possiamo notare come il costrutto `if-else` in questo caso implementi la *Procedure Call* `integer(X)`, valutando se il primo elemento della lista `s0`

sia una istanza di tipo `integer` (lanciando un fallimento nel caso non lo sia). Il primo elemento della lista `s0` potrebbe essere anche una variabile di tipo `LVar`; in questo caso per eseguire i controlli che ci interessano abbiamo bisogno di estrarre il valore dalla variabile (`o = ((LVar)o).getValue()`).

```
      case 1: {
        // S0 = [X|S], variable(X), Parse_tree = [(Z = X)].
        Solver.addChoicePoint(c);

        LVar x = new LVar();
        Object o = s0.get(0);
        LVar v = new LVar();
        if (o instanceof LVar)
          o = ((LVar)o).getValue();

        if (variable(o, v)) {
          Solver.add(s0.eq(s.ins1(x)));
          Solver.add(parse_tree.eq(LList.empty()
                                   .ins1(z.eq(v))));
        }
        else
          c.fail();
        break;
      }
```

La chiamata `variable(o, v)` è vera se il *token* `o` denota una variabile valida e restituisce questa variabile sfruttando il parametro `v`. Per ora non entriamo in dettaglio nell'implementazione di `variable`, la quale verrà spiegata meglio in 4.5.2. Il procedimento quindi è lo stesso del ramo precedente: in caso il controllo su `o` fallisca avviene un fallimento.

```
      case 2: {
        // S0 = ['('|S1],
        // expr(Z, Parse_tree, S1, S2), S2 = [')'|S].

        LList s1 = new LList();

        Solver.add(s0.eq(s1.ins1('(')));
        Solver.add(expr(z, k, s1, s.ins1(')')));
        break;
      }
    }
    return;
  }
```

Notiamo che la coppia di vincoli:

```
expr(Z, Parse_tree, S1, S2), S2 = [')'|S]
```

equivale al singolo vincolo:

```
expr(Z, Parse_tree, S1, [')'|S])
```

Nell'implementazione **JSetL** si è optato per questa versione più stringata, la quale mantiene inalterato il significato della produzione.

4.2.1 Esempi

Nel seguente esempio prendiamo in considerazione l'espressione $4 * 3 + 2$:

```
public static void main(String[] args) throws Failure {
  SolverClass Solver = new SolverClass();
  ExprParser expr_p = new ExprParser(Solver);

  IntLVar x = new IntLVar("X");
  LList parse_tree = new LList("Parse Tree");
  // expr = '4*3+2'
  LList l_expr =
    LList.empty().insn(4).insn('*').insn(3).insn('+').insn(2);

  Solver.solve(expr_p.expr(x, parse_tree,
                           l_expr, LList.empty()));
  x.output();
  parse_tree.output();
}
```

Questo codice istanzia un `solver`, costruisce l'oggetto `expr_p` di tipo `ExprParser`, crea una variabile intera logica `x` *unbound*, una lista logica `parse_tree` *unbound* ed una lista `l_expr` rappresentante l'espressione e chiama il metodo `solve(...)` sul vincolo definito dal `Constraint` restituito dalla chiamata `expr_p.expr(x, parse_tree, l_expr, LList.empty())`. L'output di questo esempio è il seguente:

```
X = unknown -- Domain: [-1073741823..1073741823]
Parse Tree = [[[_?21 = 4],
               [_?16 = _?27 AND _?27 = _?21 * _?26,[_?26 = 3]]],
               [_X = _?33 AND _?33 = _?16 + _?32,[_?32 = 2]]]
```

Si nota che la variabile `x` è rimasta *unbound*, mentre `parse_tree` è una lista rappresentante un albero le cui foglie sono vincoli. Le variabili coinvolte nei vincoli memorizzati in `parse_tree` vengono stampate con il loro nome esterno. Le variabili di supporto generate hanno tutte nome esterno "`_?`n"[3], mentre la variabile `x` è denotatata dal suo nome esterno preceduto dal carattere underscore ("`_X`").
La lista `parse_tree` rappresenta i vincoli da valutare affinchè alla variabile `x` venga unificato il valore dell'espressione in input. Osserviamo questo secondo esempio in cui il *Parser* viene applicato all'espressione $3 * (Y + Z)$:

```
public static void main(String[] args) throws Failure {
  SolverClass Solver = new SolverClass();
  ExprParser expr_p = new ExprParser(Solver);

  IntLVar x = new IntLVar("X");
  IntLVar y = new IntLVar(8).setName("Y");
  IntLVar z = new IntLVar("Z");
  LList parse_tree = new LList("Parse Tree");
  // expr = '3*(Y+Z)'
  LList l_expr =
    LList.empty().insn(3).insn('*').insn('(').insn("Y")
    .insn('+').insn("Z").insn(')');

  Solver.solve(expr_p.expr(x, parse_tree,
                           l_expr, LList.empty()));
  parse_tree.output();
}
```

ed il relativo output:

```
Parse Tree = [[_?5 = 3],
              [_X = _?11 AND _?11 = _?5 * _?10,
               [[_?19 = 8],
                [_?10 = _?25 AND _?25 = _?19 + _?24,
                 [_?24 = _Z]]]]]
```

Anche in questo caso abbiamo ottenuto un albero, sotto forma di lista, con i diversi vincoli necessari per valutare correttamente l'espressione di input. Si noti che fra le variabili coinvolte nei vincoli compaiono sia `_X`, nome esterno di

[3] La versione attuale di **JSetL** assegna a tutte le variabili il cui nome esterno non è stato specificato lo stesso nome esterno di default "`_?`"; per realizzare questo esempio è stata effettuata una piccola modifica alla classe `LVar` per dare nomi esterni di default distinti da variabile a variabile; questa modifica non fa parte del pacchetto **JSetL** ufficiale.

`x`, che `_Z`, nome esterno di `z`, ma non compaia `_Y`, che denoterebbe la variabile `y`, anch'essa coinvolta nell'espressione. Questo perchè `y`, a differenza di `z`, è *bound* e viene quindi utilizzato direttamente il suo valore per descrivere correttamente il `Constraint` che la coinvolge.

4.3 Scanner: la classe ExprTokenizer

`ExprTokenizer` descrive uno *Scanner*; `ExprTokenizer` prende in input una stringa, sotto forma di lista di caratteri, e genera da essa una lista di *token*. Il primo passo perciò è quello di identificare quali sono i *token* che ci interessano:

- **operatori** : i caratteri '+', '−', '∗' e '/' rappresentano i possibili operatori delle nostre espressioni; questi caratteri presi singolarmente identificano un *token* valido (il carattere '−' può avere un altro significato; dovrà perciò essere contestualizzato per stabilire che *token* identifica);
- **parentesi** : i caratteri '(' e ')' identificano due *token* validi;
- **numeri interi** : una sequenza di caratteri numerici posta all'inizio dell'espressione, dopo il carattere '(' o dopo un operatore rappresenta un numero intero; l'intero denotato dalla sequenza di cifre è un *token* valido; la sequenza di caratteri numerici può essere preceduta dal carattere '−' se l'intero denotato non è posto subito dopo un operatore;
- **variabili** : una sequenza di caratteri alfabetici, numerici e underscore ('_') che inizia con un carattere alfabetico (maiuscolo o minuscolo), posta all'inizio dell'espressione, dopo il carattere '(' o dopo un operatore, rappresenta un identificatore di variabile; la stringa il cui valore è l'identificatore stesso è un *token* valido[4].

Prima di presentare questa grammatica, vi sono alcuni semplici predicati che necessitano un'introduzione in quanto verranno da questa utilizzati:

- `is_operator(`Op`)` : questo predicato è vero se $Op \in \{$ '+', '−', '∗', '/' $\}$;
- `is_char(`Ch`)` : questo predicato è vero se Ch è un carattere dell'alfabeto, maiuscolo o minuscolo;
- `is_digit(`X`)` : questo predicato è vero se X è un carattere rappresentante una cifra;

[4] Non prendiamo in considerazione identificatori di variabili che iniziano con il carattere '_' per comodità; applicando una semplice modifica alla grammatica è comunque possibile riconoscere anche tali identificatori.

- `create_number(`$Chars$`, `N`)` : questo predicato è vero se i caratteri nella lista $Chars$ rappresentano il numero N;
- `create_var(`$Chars$`, `V`)` : questo predicato è vero se i caratteri nella lista $Chars$ rappresentano la stringa V.

Utilizzeremo inoltre il predicato Prolog `concat` presentato precedentemente in 2.1.1[5]. Costruiamo la **DCG** necessaria osservando quanto appena specificato.
Gli unici *token* che non si possono incontrare all'inizio dell'espressione sono il carattere ')' e gli operatori (con l'eccezione del simbolo '−', se legato ad una sequenza di cifre). Le produzioni del simbolo iniziale saranno pertanto:

```
start_tokenizer(Tokens) -->
  ( ['-', X], {is_digit(X)}, number_tail(Tokens, ['-', X])
  | ['('], start_tokenizer(Next_Tokens), {Tokens = ['('|Next_Tokens]}
  | [X], {is_digit(X)}, number_tail(Tokens, [X])
  | [X], {is_char(X)}, variable_tail(Tokens, [X])
  | [ ], {Tokens = [ ]}
  ).
```

Abbiamo cinque possibili produzioni per il simbolo iniziale. L'argomento `Tokens` conterrà il nostro output, ovvero la lista di *token* generata a partire dalla lista di caratteri dell'espressione.

Analizziamo ora come queste produzioni vengono implementate in **JSetL**. Tralasciamo la scrittura dell'interfaccia utente e del metodo `user_code`, i quali sono realizzabili esattamente come mostrato negli esempi precedenti.

```
private void start_tokenizer(Constraint c) throws Failure {
  LList tokens = (LList)c.getArg(1);
  LList s0 = (LList)c.getArg(2);
  LList s = (LList)c.getArg(3);

  switch(c.getAlternative()) {

   // ... differenti produzioni

  }
  return;
}
```

[5] Il predicato `concat` qui utilizzato equivale al predicato `built-in` di Prolog `append`, non all'omonimo predicato `concat` utilizzato in Prolog per la concatenazione di termini.

La prima produzione determina se è presente un intero negativo all'inizio dell'espressione:

```
    case 0: {
      Solver.addChoicePoint(c);

      LList s1 = new LList();
      LVar x = new LVar();

      solverAux.clearStore();
      if (solverAux.check(s0.eq(s1.ins1(x).ins1('-')))
          && is_digit(x))) {
        Solver.add(number_tail(tokens,
                               LList.empty().ins1(x).ins1('-')),
                               s1, s);
      }
      else
        c.fail();
      break;
    }

    // ... case 1 : ['('], start_tokenizer(Next_Tokens),
    //              {Tokens = ['('|Next_Tokens]}
```

Si nota l'utilizzo del `solver` ausiliario per ottenere il secondo elemento della lista `s0`, che dobbiamo controllare essere una cifra; in caso contrario il vincolo lancerà un fallimento. L'utilizzo di `solverAux` rende inutile aggiungere anche a `Solver` il vincolo `s0.eq(s1.ins1(x).ins1('-'))`, in quanto questo viene già risolto dal risolutore ausiliario.

Questo ramo equivale alla produzione in forma *esplicita*:

```
start_tokenizer(Tokens, S0, S) :-
  S0 = ['-', X|S1],
  is_digit(X),
  number_tail(Tokens, ['-', X], S1, S).
```

La seconda produzione descritta in forma *esplicita* è:

```
start_tokenizer(Tokens, S0, S) :-
  S0 = ['('|S1],
  start_tokenizer(Next_Tokens, S1, S),
  Tokens = ['('|Next_Tokens].
```

L'implementazione in **JSetL** è assolutamente canonica. Passiamo perciò alle due produzioni successive, le quali hanno identica struttura, differendo solo nel controllo effettuato (`is_digit(X)` la prima e `is_char(X)` la seconda) e nel non-terminale derivato (rispettivamente `number_tail` e `variable_tail`).

```
    // ... case 2 : [X], {is_digit(X)},
    //              number_tail(Tokens, [X])

    case 3: {
      Solver.addChoicePoint(c);

      LList s1 = new LList();
      LVar x = new LVar();

      solverAux.clearStore();
      if (solverAux.check(s0.eq(s1.ins1(x))) &&
          is_char(x)) {
        Solver.add(variable_tail(tokens,
                                 LList.empty().ins1(x)),
                                 s1, s);
      }
      else
        c.fail();
      break;
    }
```

Questo ramo rappresenta la produzione:

```
start_tokenizer(Tokens, S0, S) :-
  S0 = [X|S1],
  is_char(X),
  variable_tail(Tokens, [X], S1, S).
```

L'ultima produzione da considerare è la seguente:

```
start_tokenizer(Tokens, S0, S) :-
  S0 = S,
  Tokens = [ ].
```

Si può notare come il vincolo `S0 = S` implichi che la lista rappresentatadalle *Difference Lists* sia la lista vuota; questo accade quando la lista di caratteri da cui ricavare la lista di *tokens* è vuota oppure quando siamo giunti

in fondo alla lista di input.
Osserviamo l'implementazione di quest'ultima produzione:

```
    case 4: {
      solverAux.clearStore();
      if (solverAux.check(s0.eq(s))) {
        Solver.add(tokens.eq(LList.empty()));
      }
      else
        c.fail();
      break;
    }
```

In questo caso il `solver` ausiliario viene utilizzato per stabilire se il vincolo `s0.eq(s)` sia vero, sollevando un fallimento in caso contrario.
Abbiamo visto come questo non-terminale generi altri non-terminali, quali `number_tail` e `variable_tail`; questi hanno produzioni del tutto simili fra loro, perciò ci soffermeremo sull'implementazione di uno solo dei due.
Osserviamo ora le produzioni relative al non-terminale `number_tail` descritte in notazione *implicita*:

```
number_tail(Tokens, Tmp) -->
  [X],
  {is_digit(X), concat(Tmp, [X], New_Tmp)},
  number_tail(Tokens, New_Tmp).
number_tail(Tokens, Tmp) -->
  [')'],
  {create_number(Tmp, Element),
  Tokens = [Element, ')'|Next_Tokens]},
  expr_tokenizer(Next_Tokens).
number_tail(Tokens, Tmp) -->
  [X],
  {is_operator(X),
  create_number(Tmp, Element),
  Tokens = [Element, X|Next_Tokens]},
  expr_tokenizer(Next_Tokens).
number_tail(Tokens, Tmp) -->
  [ ],
  {create_number(Tmp, Element), Tokens = [Element]}.
```

Fra i suoi argomenti, oltre al già presentato `Tokens`, compare `Tmp`, una lista il cui scopo è di immagazzinare tutti i caratteri del *token* corrente; `number_tail` riconosce i *token* numerici, ovvero gli interi della nostra espressione. La prima produzione viene richiamata ricorsivamente fintanto che la

sequenza di caratteri in input presenta delle cifre, memorizzandole in `Tmp`; le altre produzioni rappresentano invece i casi in cui si è raggiunto il termine del *token*. Dal momento che la prima produzione non è differente da quanto già visto in altri casi, prendiamo in esame le produzioni restanti. L'implementazione in **JSetL**, osservando direttamente il ramo del costrutto `switch` rappresentante la penultima produzione, è:

```
// ...
case 3: {
 Solver.addChoicePoint(c);
 LList s1 = new LList();
 LList next_tokens = new LList();
 solverAux.clearStore();
 if (solverAux.check(s0.eq(s1.ins1(x))) &&
     is_operator(x)) {
   int element = create_number(tmp);
   Solver.add(tokens.eq(next_tokens.ins1(x).ins1(element)));
 }
 else
   c.fail();
 break;
}
// ...
```

Il metodo `create_number` è un metodo che costruisce l'intero descritto da una sequenza di caratteri numerici. Nelle produzioni di `variable_tail` viene invece chiamato il metodo `create_var`, che restituisce la stringa risultante dalla concatenazione dei caratteri incontrati da `variabile_tail`.
Nella forma Prolog della grammatica `create_number` e `create_var` sono clausole, tuttavia l'operazione che svolgono è prettamente deterministica, pertanto possiamo sfruttare un'implementazione più Java-style nel nostro codice:

```
private String create_var(LList l) {
  String result = "";
  for(int i = 0; i < l.getSize(); i++) {
    LVar x = new LVar(l.get(i));
    result = result + x.getValue().toString();
  }
  return r;
}

private int create_number(LList l) {
  String result = create_var(l);
  return Integer.valueOf(result);
}
```

L'ultima produzione di `number_tail` rappresenta il caso in cui non vi sono altri caratteri della nostra espressione da scansionare e l'implementazione **JSetL** segue banalmente quanto proposto fin'ora.
Si può notare che è stato introdotto un nuovo non-terminale in queste produzioni, ovvero `expr_tokenizer`. Osserviamo le sue produzioni:

```
expr_tokenizer(Tokens) -->
  ( ['('], start_tokenizer(Next_Tokens), {Tokens = ['('|Next_Tokens]}
  | [')'], expr_tokenizer(Next_Tokens), {Tokens = [')'|Next_Tokens]}
  | [X], {is_operator(X), Tokens = [X|Next_Tokens]},
         expr_tokenizer(Next_Tokens)
  | [X], {is_digit(X)}, number_tail(Tokens, [X])
  | [X], {is_char(X)}, variable_tail(Tokens, [X])
  | [], {Tokens = []}
  ).
```

Le produzioni di `expr_tokenizer` sono molto simili a quelle di `start_tokenizer`. Le differenza fra i due non-terminali è la seguente:

- `start_tokenizer` : rappresenta la scansione all'inizio di un'espressione o sottoespressione; indica quindi l'inizio della stringa di input o un punto della stringa immediatamente successivo ad una parentesi aperta; non può generare operatori e parentesi chiuse;

- `expr_tokenizer` : rappresenta la scansione all'interno di un'espressione o sottoespressione; non può generare interi negativi.

Questa suddivisione in due non-terminali differenti viene fatta per disambiguare il carattere '`-`' all'interno di un'espressione e generare quindi il *token* corretto; inoltre permette di evitare errori sintattici relativi ai numeri interi negativi come `2*-1`, accettando solo la corretta scrittura `2*(-1)`.

Nell'appendice A vengono presentate per intero le grammatiche del *Parser* e dello *Scanner*, mostrando il loro codice Prolog, sia in notazione *implicita* che *esplicita*.

4.3.1 Esempio

Prendiamo in considerazione il seguente esempio:

```
public static void main(String[] args) throws Failure {
  SolverClass Solver = new SolverClass();
  ExprTokenizer expr_t = new ExprTokenizer(Solver);

  LList tokens = new LList("Tokens");
  String s_expr = "-3*(12+8)-x1/y+x2*(-3)";
  LList l_expr = LList.empty();
  for(int i = 0; i < s_expr.length(); i++)
    l_expr = l_expr.insn(s_expr.charAt(i));
  l_expr.setName("l_expr");

  Solver.solve(expr_t.start_tokenizer(tokens, l_expr,
                                      LList.empty()));
  l_expr.output();
  tokens.output();
}
```

Questo codice istanzia un `solver`, crea l'oggetto `expr_t` (lo *Scanner*) di tipo `ExprTokenizer` ed una lista `tokens`. Definisce inoltre una stringa `s_expr` con valore "$-3*(12+8)-x1/y+x2*(-3)$" rappresentante il nostro input ed una lista `l_expr` nella quale vengono inseriti nello stesso ordine i caratteri della stringa `s_expr`. Infine questo esempio mostra in output la lista `l_expr` e la lista `tokens` da essa generata:

```
l_expr = [-,3,*,(,1,2,+,8,),-,x,1,/,y,+,x,2,*,(,-,3,)]
Tokens = [-3,*,(,12,+,8,),-,x1,/,y,+,x2,*,(,-3,)]
```

4.4 Translator: la classe traduttore

`Translator` si occupa di richiamare le due **DCG** appena descritte ed interpretare l'output del *Parser* per generare il codice.
Vediamo nello specifico come la classe `Translator` è implementata:

```
public class Translator {

  // solver per tokenizer
  static SolverClass t_solver = new SolverClass();
  // solver per parser
  static SolverClass p_solver = new SolverClass();
```

```
  ExprTokenizer expr_tokenizer = new ExprTokenizer(t_solver);
  ExprParser expr_parser = new ExprParser(p_solver);

  // ... definizioni dei metodi

}
```

`Translator` dichiara due `solver` differenti, uno per lo *Scanner* ed uno per il *Parser*, come si può notare nella dichiarazione degli oggetti `expr_tokenizer` e `expr_parser`.
`Translator` ci permette, tramite il metodo `expr`, di legare una `IntLVar` al valore di un'espressione. Tale metodo è così implementato:

```
  public IntLVar expr(String s_expr) throws Failure {
    IntLVar result = new IntLVar();
    translate_expr(s_expr, result);
    return result;
  }
```

Il parametro `s_expr` rappresenta l'espressione in input, sotto forma di stringa. La variabile `IntLVar result` invece è la variabile a cui viene associato il valore dell'espressione per poi essere restituita come output al chiamante.
Sia `result` che la stringa `s_expr` vengono passati come parametri al metodo `translate_expr`[6]:

```
  public Constraint translate_expr(String s_expr, IntLVar result)
  throws Failure {
    LList l_expr = LList.empty();
    // creazione della lista di caratteri dalla stringa di input
    for(int i = 0; i < s_expr.length(); i++)
      l_expr = l_expr.insn(s_expr.charAt(i));

    LList token_list = new LList();
    // generazione della lista di token
    t_solver.solve(expr_tokenizer.scan(token_list, l_expr));

    Constraint c = new Constraint();

    // in caso la lista di token sia vuota non viene
    // chiamato il Parser
```

[6] In Java gli oggetti vengono passati sempre per riferimento, pertanto le modifiche apportate a `result` dal metodo `translate_expr` vengono mantenute, che è quello che vogliamo.

```
    if (token_list.getSize() > 0) {
      LList parse_tree = new LList().setName("parse_tree");
      // generazione del Parse Tree
      p_solver.solve(expr_parser.parse(result, parse_tree,
                                       token_list));

      c = codeGeneration(parse_tree, result);
    }
    return c;
  }
```

Il metodo `translate_expr` esegue in sequenza le seguenti operazioni:

1. genera una `LList` contenente i caratteri dell'espressione ordinati come nella stringa;
2. genera la lista di *token* tramite lo *Scanner* richiamando il metodo `scan(token_list, l_expr)`; questo metodo aggiunge automaticamente il vincolo:

   `start_tokenizer(token_list, l_expr, LList.empty())`

   al *constraint store* di `t_solver`;
3. in caso la lista di *token* sia non vuota genera il *parse tree* dell'espressione (contenente i vincoli definiti dall'espressione stessa) richiamando il metodo `parse(z, k, token_list)`; questo metodo aggiunge automaticamente il vincolo:

   `expr(z, parse_tree, token_list, LList.empty())`

   al *constraint store* di `p_solver`;
4. tramite `codeGeneration(parse_tree, result)` genera, a partire da `parse_tree` il codice necessario per esprimere i vincoli rappresentanti l'espressione di input; questo metodo verrà presentato più avanti.

4.5 Applicazione a JSetL

Un primo utilizzo delle **DCG** in **JSetL** è quello di creare grammatiche che permettano di descrivere vincoli in maniera più intuitiva. Gli usi sono molteplici, dalla descrizione di operazioni complesse (per esempio espressioni aritmetiche o insiemistiche, operazioni su liste, ecc.) alla denotazione di strutture

dati composte (per esempio la possibilità di descrivere liste e insiemi sotto forma di stringa).

Il traduttore appena descritto è per esempio utilizzabile per trasformare in codice **JSetL** espressioni aritmetiche scritte in forma di stringa.

In **JSetL** inserire vincoli legati ad espressioni aritmetiche risulta piuttosto scomodo. Se per esempio volessimo aggiungere il vincolo $x = 5 + (y * 2)$ al nostro problema, dovremmo scrivere:

```
public static void main(String[] args) throws Failure {
  SolverClass solver = new SolverClass();
  IntLVar x = new IntLVar("x");
  IntLVar y = new IntLVar("y");
  IntLVar z = new IntLVar("z");
  solver.add(z.eq(y.mul(2)));
  solver.add(x.eq(z.sum(5)));
}
```

Il traduttore descritto precedentemente ci permette di descrivere vincoli sotto forma di stringa, quindi in modo più rapido e leggibile.
Utilizzando la classe `Translator` è possibile riscrivere il codice appena proposto come segue:

```
public static void main(String[] args) throws Failure {
  SolverClass solver = new SolverClass();
  Translator t = new Translator();
  IntLVar x = new IntLVar("x");
  IntLVar y = new IntLVar("y");
  solver.add(x.eq(t.expr("5+(y*2)")));
}
```

La lettura del codice è più intuitiva che nel caso precedente. Il metodo `expr` della classe `Translator` ha il compito di restituire un `IntLVar` vincolata all'espressione descritta nella stringa in input.

4.5.1 Generazione del codice

In 4.4 abbiamo presentato senza descriverlo il metodo `codeGeneration`.
Questo metodo non è stato descritto in quanto può essere implementato in maniera diversa a seconda del risultato che si vuole ottenere. Il suo scopo è generare codice **JSetL** a partire da una rappresentazione ad albero, sotto

forma di lista, dell'espressione iniziale.
Abbiamo detto che l'output di `ExprParser` è una lista contenete un *parse tree* in cui ogni foglia è un oggetto di tipo `Constraint`. Un'implementazione possibile è la seguente:

```
private
Constraint codeGeneration(LList parse_tree, IntLVar result) {
  c = parseTree_toConstraint(parse_tree);
  result.setConstraint(c);
  return c;
}
```

Il metodo `parseTree_toConstraint(...)` prende in input una lista rappresentante un *parse tree* e restituisce un unico `Constraint c` ottenuto congiungendo tutti i vincoli memorizzati al suo interno; questi è un semplice algoritmo che esplora l'albero e concatena i `Constraints` posti nelle foglie dell'albero. Il metodo `setConstraint(c)`, già presentato in 2.2.2, associa il `Constraint` ottenuto da `parseTree_toConstraint` alla variabile di invocazione in modo che questo venga valutato in caso la risoluzione di un problema coinvolga la variabile in questione.

4.5.2 Problema: variabili nelle espressioni

Durante la descrizione di `ExprParser` abbiamo introdotto, senza descriverlo, il predicato `variable(X)` 4.2. Questo predicato è vero se `X` rappresenta una variabile valida per la nostra espressione. Non è stato ulteriormente specificato questo vincolo in quanto dipendente dall'implementazione.
Prendendo l'espressione sotto forma di stringa, il *token* generato per rappresentare una variabile è anch'esso una stringa. Vi era quindi il problema di collegare il nome della variabile, passato sotto forma di stringa, all'oggetto vero e proprio.
In **JSetL** , come visto in 2.2.1, è possibile specificare il nome esterno di una variabile `LVar` tramite il metodo `setName(...)`. Inoltre la classe `LVar` mantiene un vettore contenente tutte le variabili dichiarate, chiamato `nonInitLVar`.
Si è reso necessario pertanto aggiungere il metodo `getVar(...)` alla classe `LVar` che, preso in input un nome, restituisse la variabile associata a quel nome, se presente.
Dal momento che il vettore `nonInitLVar` è dichiarato nella classe `LVar`, l'implementazione del metodo specificato è definita all'interno di questa classe ed è la seguente:

```
  public static LVar
  getVar(String name) {
    Iterator<LVar> itr = nonInitLvar.iterator();
    LVar currentVar;
    while(itr.hasNext()) {
    currentVar = itr.next();
    if (currentVar.getName().compareTo(name) == 0)
      return currentVar;
    }
    return null;
  }
```

Le classi che espandono `LVar` hanno anch'esse questo metodo, quindi è richiamabile in particolare dalla classe `IntLVar`.
Un metodo analogo, chiamato `getCollection`, è stato implementato nella classe `LCollection` per permettere la stessa operazione su oggetti di tipo `LCollection` o derivati da esso, in particolare `LList` e `LSet`; l'implementazione è la stessa, pur lavorando su un diverso vettore chiamato `nonInit`. Il codice seguente implementa il metodo `variable(o, v)`, usato per mappare in **JSetL** il predicato `Variable(X)`:

```
private boolean variable(Object o, LVar v) {
  if (o instanceof java.lang.String) {
    v = LVar.getVar((java.lang.String)o);
    if (v != null) {
      return true;
    }
  }
  return false;
}
```

Nel caso in cui venga utilizzata in un'espressione una variabile non ancora dichiarata o il cui nome non è stato specificato, il *Parser* fallirà.
È importante notare che il nome con cui riferirsi alla variabile all'interno dell'espressione è il nome *esterno* della variabile, definibile tramite il metodo `setName(...)`, non l'identificatore della variabile.
Osserviamo il seguente esempio:

```
public static void main(String[] args)
throws Failure {
  SolverClass solver = new SolverClass();
  Translator t = new Translator();
```

```
    IntLVar result1 = new IntLVar().setName("result1");
    IntLVar result2 = new IntLVar().setName("result2");
    IntLVar result3 = new IntLVar().setName("result3");
    IntLVar x = new IntLVar(5);
    IntLVar y = new IntLVar(6, 9).setName("y");
    IntLVar z = new IntLVar("PW");

    solver.solve(result1.eq(t.expr("y+8")));
    result1.output();
    solver.solve(result2.eq(t.expr("y+8-PW")));
    result2.output();
    solver.solve(result3.eq(t.expr("y+x-PW")));
    result3.output();
  }
```

L'output di questo esempio è:

```
    result1 = unknown -- Domain: [14..17]
    result2 = unknown -- Domain: [-1073741809..1073741823]
    Exception in thread "main" JSetL.Failure
      // ...
```

Si nota che nel primo caso l'espressione è corretta e viene ristretto correttamente il dominio di `result1`; anche nel secondo caso l'espressione è corretta, ma essendo la variabile `result2` dipendente da un'altra variabile *unbound* il dominio non viene ristretto; nel terzo caso, dal momento che la variabile `x` è stata dichiarata ma senza specificarne il nome *esterno*, l'espressione viene considerata sbagliata e viene lanciato un fallimento.
È importante inoltre ricordare che è possibile assegnare lo stesso nome *esterno* a due variabili differenti; dal momento che il nome *esterno* non è utilizzato ai fini della risoluzione, il comportamento potrebbe non essere quello voluto. Per esempio:

```
  public static void main(String[] args) throws Failure {
    SolverClass solver = new SolverClass();
    Translator t = new Translator();

    IntLVar result1 = new IntLVar().setName("result1");
    IntLVar x = new IntLVar(5).setName("y");
    IntLVar y = new IntLVar(6, 9).setName("y");

    solver.solve(result1.eq(t.expr("y+8")));
    result1.output();
  }
```

L'espressione in questo esempio risulta corretta e restituisce un risultato, ma il comportamento potrebbe non essere quello voluto dall'utente.
In questi casi è compito dell'utente fare in modo che non vi siano ambiguità.

Capitolo 5

Ottimizzazioni

Le implementazioni in Java delle grammatiche proposte nel capitolo 4 sono la traduzione diretta, produzione per produzione, delle **DCG** Prolog equivalenti in notazione *esplicita*. Seppur corrette, queste presentano tempi di risoluzione altissimi.
Osserviamo alcuni tempi di esecuzione registrati dallo *Scanner* e dal *Parser* precedentemente creati con delle espressioni di esempio:

Espressione	Scanner	Parser
-15+18*2	0.033 s	0.021 s
(-15+18)*2+x-y	0.056 s	0.158 s
(-15+18)*((2+x)-y*(13+z))	0.063 s	0.306 s
((-15+18)*((2+x)-y*(13+z))*(-1)+(z*2+x-(y+1)))	0.192 s	1.417 s
(-15+18)*3+y-2+28/2+2+x-y*13+z*(-1)+z*2+x-y+1-10+324/2-(x*y)	0.720 s	0.570 s
((-15+(18))*(((2)+x)-y*((13+z)))*(-1)+(z*(2+x)-(y+(1))))	0.217 s	7.205 s
(((-15)+(18))*(((2)+(x))-(y)*(((-13)+z)))*(-1)+((z)*((2)+(x))-((y)+(1))))	0.534 s	27.441 s

Si nota come i tempi dello *Scanner* crescano linearmente con la lunghezza dell'espressione mentre quelli del *Parser* crescano in modo non-lineare dipendendo dalla forma dell'espressione. Verranno presentati ora alcuni metodi per rimediare, almeno parzialmente, a questi problemi, anche se questo ci porterà a perdere la corrispondenza diretta con le **DCG**.

5.1 Backtracking

Ogni volta che viene applicato il *backtracking* avvengono una serie di eventi:

- viene cercato l'ultimo *choice point*;
- vengono ripristinate tutte le variabili modificate dall'ultimo *choice point*;

- vengono tolti dal *constraint store* i vincoli aggiunti dall'ultimo *choice point* in poi;
- viene ripristinato lo stato del `Constraint` fallito;
- viene cercata un'altra soluzione.

Tutte queste operazioni sono molto costose. La regola generale è quindi di evitare il più possibile il *backtracking*
Nell'implementazione mostrata per lo *Scanner*, tramite la classe `ExprTokenizer`, vi erano di fatto alcune ottimizzazioni implicite: i predicati `is_char(X)`, `is_digit(X)` e `is_operator(X)` infatti sono stati implementati per comodità come metodi Java e non come `Constraint`. Questo ha reso necessario l'utilizzo del risolutore ausiliario `solverAux` per ricavare il primo elemento della lista `s0` ogni volta che fosse necessario controllare quale carattere fosse. Questo ha un effetto molto importante sul *constraint store*: valutando se un ramo è corretto o meno prima di aggiungere i vincoli, si evita di doverli rimuovere dal *constraint store* in caso di *backtracking*.

Una prima modifica che si può effettuare è quella di raggruppare quante più produzioni possibili in un unico percorso, stabilendo deterministicamente il percorso corretto senza bisogno di lanciare fallimenti. Prendiamo questo frammento di codice di `ExprTokenizer`:

```
private void start_tokenizer(Constraint c)
throws Failure {
  LList tokens = (LList)c.getArg(1);
  LList s0 = (LList)c.getArg(2);
  LList s = (LList)c.getArg(3);
  switch(c.getAlternative()) {
    case 0: {
       Solver.addChoicePoint(c);
       LList s1 = new LList();
       LVar x = new LVar();
       solverAux.clearStore();
       if (solverAux.check(s0.eq(s1.ins1(x).ins1('-')))
           && is_digit(x)) {
         Solver.add(number_tail(tokens,
                                LList.empty().ins1(x).ins1('-'),
                                s1, s));
       }
       else
         c.fail();
       break;
     }
```

```
    case 1: {
      Solver.addChoicePoint(c);
      LList s1 = new LList();
      LList next_tokens = new LList();
      LVar x = new LVar();
      solverAux.clearStore();
      if (solverAux.check(s0.eq(s1.ins1('('))) {
        Solver.add(tokens.eq(next_tokens.ins1('(')));
        Solver.add(start_tokenizer(next_tokens, s1, s));
      }
      else
       c.fail();
      break;
    }
    //...
```

rappresentante le produzioni del non-terminale `start_tokenizer`.
Si può notare che entrambi i rami del costrutto `switch` osservano l'inizio della lista `s0` per stabilire se il percorso è corretto, lanciando esplicitamente un fallimento in caso contrario. Tutte le produzioni della grammatica implementata in `ExprTokenizer` hanno questa forma, con l'unica eccezione delle produzioni finali.
Osserviamo ora questa modifica al codice precedente:

```
private void start_tokenizer(Constraint c)
throws Failure {
    LList tokens = (LList)c.getArg(1);
    LList s0 = (LList)c.getArg(2);
    LList s = (LList)c.getArg(3);
    solverAux.clearStore();
    if (solverAux.check(s0.eq(s))) {
      Solver.add(tokens.eq(LList.empty()));
    }
```

Questo primo `if-else` sostituisce il ramo della produzione:

```
start_tokenizer(Tokens) -> [ ], {Tokens = [ ]};
```

```
    else {
      LList s1 = new LList();
      LVar x = new LVar();
      solverAux.clearStore();
      if (solverAux.check(s0.eq(s1.ins1(x).ins1('-')))
        && is_digit(new LVar(s0.get(1)))) {
```

```
        Solver.add(number_tail(tokens, LList.empty()
                               .ins1(x).ins1('-'), s1, s));
      }
```

Questo secondo `if-else` sostituisce il ramo della produzione:

```
start_tokenizer(Tokens) -> ['-', X], {is_digit(X)},
          number_tail(Tokens, ['-', X]);
```

```
      else {
        LList next_tokens = new LList();
        solverAux.solve(s0.eq(s1.ins1(x)));
        if ('(' == (Character)x.getValue()) {
          Solver.add(tokens.eq(next_tokens.ins1('(')));
          Solver.add(start_tokenizer(next_tokens, s1, s));
        }
        else if (is_digit(x)) {
          Solver.add(number_tail(tokens, LList.empty()
                                 .ins1(x), s1, s));
        }
        else if (is_char(x)) {
          Solver.add(variable_tail(tokens, LList.empty()
                                   .ins1(x), s1, s));
        }
        else {
          c.fail();
        }
      }
    }
    return;
  }
```

Quest'ultima serie di `if-else if [...]` sostituisce i restanti percorsi; si nota che vi è un'unica chiamata a:

```
solverAux.solve(s0.eq(s1.ins1(x))).
```

Nell'implementazione non ottimizzata era necessario l'utilizzo di:

```
solverAux.check(s0.eq(s1.ins1(x)))
```

in quanto `s0` poteva anche essere vuota; in questo caso invece il caso in cui `s0` è vuota viene già valutato, pertanto siamo sicuri che lungo questo percorso `s0` abbia almeno un elemento. Il metodo `solve` è leggermente più efficiente del metodo `check` e inoltre, in questa implementazione, viene eseguito una sola

volta, a differenza del metodo `check` che verrebbe eseguito per ogni ramo.
La derivazione del simbolo non-terminale è deterministica in quanto il fallimento lungo uno qualsiasi dei percorsi scelti decreta il fallimento lungo tutti i percorsi possibili relativi a quel non-terminale.
Applicando queste modifiche possiamo osservare un notevole miglioramento:

Espressione	Scanner	Ottimizzato
-15+18*2	0.033 s	0.014 s
(-15+18)*2+x-y	0.056 s	0.030 s
(-15+18)*((2+x)-y*(13+z))	0.063 s	0.049 s
((-15+18)*((2+x)-y*(13+z))*(-1)+(z*2+x-(y+1)))	0.192 s	0.077 s
(-15+18)*3+y-2+28/2+2+x-y*13+z*(-1)+z*2+x-y+1-10+324/2-(x*y)	0.720 s	0.086 s
((-15+(18))*(((2)+x)-y*((13+z)))*(-1)+(z*(2+x)-(y+(1))))	0.217 s	0.099 s
(((-15)+(18))*(((2)+(x))-(y)*(((-13)+z)))*(-1)+((z)*((2)+(x))-((y)+(1))))	0.534 s	0.136 s

I tempi in certi casi sono oltre che dimezzati.

Seguendo la stessa idea possiamo migliorare anche `ExprParser`, tuttavia i non-terminali di questa grammatica sono tutti non-deterministici, con l'unica eccezione di `factor`. Osserviamo perciò cosa succede ai tempi di esecuzione modificando il solo metodo `factor` di `ExprParser` secondo questa regola:

Espressione	Parser	Ottimizzato
-15+18*2	0.021 s	0.019 s
(-15+18)*2+x-y	0.158 s	0.046 s
(-15+18)*((2+x)-y*(13+z))	0.306 s	0.105 s
((-15+18)*((2+x)-y*(13+z))*(-1)+(z*2+x-(y+1)))	1.417 s	0.702 s
(-15+18)*3+y-2+28/2+2+x-y*13+z*(-1)+z*2+x-y+1-10+324/2-(x*y)	0.570 s	0.286 s
((-15+(18))*(((2)+x)-y*((13+z)))*(-1)+(z*(2+x)-(y+(1))))	7.205 s	3.194 s
(((-15)+(18))*(((2)+(x))-(y)*(((-13)+z)))*(-1)+((z)*((2)+(x))-((y)+(1))))	27.441 s	12.959 s

I tempi sono praticamente dimezzati anche in questo caso.

5.2 Altre ottimizzazioni

Altre ottimizzazioni sono ovviamente possibili. Osservando la struttura delle nostre grammatiche, è importante:

- cercare di lanciare meno fallimenti possibili, ordinando le produzioni in modo conveniente;

- cercare di dichiarare solo le variabili necessarie, a seconda del ramo scelto.

Mentre la prima dipende dalla grammatica, la seconda si applica molto bene durante la fase di identificazione dei parametri. Prendiamo per esempio questo frammento di codice di `ExprParser`:

```
private void addterm(Constraint c) throws Failure {
  LList pair = (LList)c.getArg(1);
  LList parse_tree = (LList)c.getArg(2);
  LList s0 = (LList)c.getArg(3);
  LList s = (LList)c.getArg(4);
  IntLVar z = (IntLVar)pair.get(0);
  IntLVar z1 = (IntLVar)pair.get(1);
  if (!s0.isBound()) {
      c.notSolved();
      return;
  }
  // ...
```

Nel caso in cui `s0` sia *unbound*, il vincolo viene settato a "*unsolved*" e si passa al vincolo successivo. I parametri `pair`, `parse_tree` e `s` in questo caso non vengono neanche presi in considerazione. Le dichiarazioni precedenti il controllo `if (!s0.isBound())` vengono ripetute ogni volta che il vincolo, presente nel *constraint store*, viene selezionato e valutato.
Risulta quindi buona cosa ottimizzare questa prima parte nel modo seguente:

```
private void addterm(Constraint c) throws Failure {
  LList s0 = (LList)c.getArg(3);
  if (!s0.isBound()) {
      c.notSolved();
      return;
  }
  LList pair = (LList)c.getArg(1);
  LList parse_tree = (LList)c.getArg(2);
  LList s = (LList)c.getArg(4);
  IntLVar z = (IntLVar)pair.get(0);
  IntLVar z1 = (IntLVar)pair.get(1);
  // ...
```

In generale è giusto valutare per qualsiasi percorso percorribile cosa serve e cosa non serve dichiarare: il ragionamento fatto si applica anche alle variabili di supporto.

Osserviamo che applicando questa semplice modifica a `ExprParser` vi è un ulteriore notevole miglioramento dei tempi:

Espressione	Ottimizzato 1	Ottimizzato 2
-15+18*2	0.019 s	0.003 s
(-15+18)*2+x-y	0.046 s	0.027 s
(-15+18)*((2+x)-y*(13+z))	0.105 s	0.046 s
((-15+18)*((2+x)-y*(13+z))*(-1)+(z*2+x-(y+1)))	0.702 s	0.447 s
(-15+18)*3+y-2+28/2+2+x-y*13+z*(-1)+z*2+x-y+1-10+324/2-(x*y)	0.286 s	0.199 s
((-15+(18))*(((2)+x)-y*((13+z)))*(-1)+(z*(2+x)-(y+(1))))	3.194 s	2.610 s
(((-15)+(18))*(((2)+(x))-(y)*(((-13)+z)))*(-1)+((z)*((2)+(x))-((y)+(1))))	12.959 s	8.547 s

In caso di espressioni scritte male o particolarmente complesse i tempi restano comunque piuttosto alti. Tuttavia si nota come con poche modifiche questi si riducano radicalmente.

Capitolo 6

Conclusioni e sviluppi futuri

Il lavoro di tesi descritto verte sull'implementazione di **DCG** utilizzando la libreria **JSetL**. Il lavoro si è svolto attraverso più fasi:

1. studio dell'argomento **DCG** ;
2. studio della libreria **JSetL** e delle funzionalità da essa offerte;
3. valutazione dell'applicazione delle **DCG** al pacchetto **JSetL** ;
4. realizzazione.

Per quanto riguarda il punto [4] si è trattato in particolare della realizzazione di una grammatica utilizzabile per descrivere espressioni aritmetiche che fosse minimale ma facilmente espandibile e utilizzabile come base per realizzare grammatiche simili.
Il lavoro ha mostrato l'attuabilità dell'obiettivo fissato e durante la sua realizzazione ha portato a galla alcuni limiti legati alla libreria. È stato quindi possibile proporre alcune piccole aggiunte al pacchetto base di **JSetL** che potranno costituire miglioramenti al progetto.

Utilizzando come base il lavoro svolto è possibile generare diversi traduttori per interfacciarsi in maniera più intuitiva alle diverse funzionalità proposte da **JSetL**. In particolare la tecnica sviluppata in questa tesi potrà essere applicata (ed in parte il lavoro è già stato svolto) alla definizione e realizzazione di un traduttore per espressioni che denotano insiemi e liste utilizzando una notazione più semplice come quella del Prolog, permettendo ad esempio di tradurre la stringa `{1, 2, 3}` nelle chiamate **JSetL**:

```
LSet.empty().ins(3).ins(2).ins(1);
```

In generale è possibile applicare quanto descritto ad una qualsiasi tipologia di costrutti linguistici, come per esempio espressioni insiemistiche costruite con gli usuali operatori insiemistici di unione, intersezione, ecc. o un costrutto più complesso come il costrutto astratto ***either-orelse*** presentato in 2.1. In quest'ultimo caso il traduttore dovrebbe generare il codice Java contenente le opportune definizioni e chiamate dei metodi **JSetL** per realizzare il comportamento non-deterministico previsto dal costrutto ***either-orelse***.

La struttura delle **DCG** consente tra le altre cose di combinarle facilmente fra loro, permettendo di espandere una grammatica definita precedentemente inglobandola in un'altra; questo vuol dire che nel caso di modifica della grammatica a livello basso (per esempio se si volessero aggiungere nuove operazioni, nel caso delle espressioni aritmetiche) è necessario modificare le produzioni, ma nel caso di aggiunta a livello più alto (per esempio se si volessero aggiungere operatori di confronto fra espressioni) questa può rimanere tale e quale ed essere semplicemente richiamata da un'altra grammatica.

Un altro possibile sviluppo potrebbe essere la realizzazione di uno strumento in grado di tradurre direttamente una data **DCG** descritta con notazione *implicita* nell'equivalente programma Java+**JSetL**, in modo da permettere una implementazione più rapida di strumenti di analisi e traduzione di semplici linguaggi.

Bibliografia

[1] Fernando C. N. Pereira, David H. D. Warren
Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks
Artificial Intelligence 13 231–278 (1980).

[2] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[3] Gianfranco Rossi, Roberto Amadini
JSetL User's Manual Version 2.3
Quaderni del Dipartimento di Matematica, n. 507, Università di Parma, 24 gennaio 2012.

[4] Mark Johnson
Two ways of formalizing grammars
Linguistics and Philosophy 17, pages 221-248.

[5] David S. Warren
Programming in Tabled Prolog
`http://www.cs.sunysb.edu/~warren/xsbbook/`

[6] Agostino Dovier, Roberto Giacobazzi
Fondamenti dell'Informatica: Linguaggi Formali, Calcolabilità e Complessità
`http://www.dimi.uniud.it/ dovier/DID/dispensa.pdf`

[7] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Compilers: Principles, Techniques and Tools
Addison-Wesley Publishing Company, 1986.

[8] Maurizio Gabbrielli, Simone Martini
Linguaggi di programmazione: principi e paradigmi
McGraw-Hill Italia, 2005.

[9] SWI-Prolog Reference Manual
Constraint Logic Programming over Finite Domains
`http://www.swi-prolog.org/pldoc/refman/`

[10] JSetL Home Page
`http://cmt.math.unipr.it/jsetl.html`

Appendice A

Grammatiche e classe Translator

In questa appendice vengono mostrate nella loro interezza le grammatiche `ExprParser` e `ExprTokenizer` in notazione Prolog *implicita* ed *esplicita*. Viene inoltre mostrato il codice della classe `Translator`.

A.1 ExprParser

Notazione implicita

```
1   expr(Z, Parse_tree) -->
      ( term(Z, Parse_tree)
      | term(Z1, P1), addterm([Z, Z1], P2), {Parse_tree = [P1, P2]}
      ).
5
    addterm([Z, Z1], Parse_tree) -->
      ( ['+'], expr(Z2, P1), {Parse_tree = [P1, (Z is Z1 + Z2)]}
      | ['-'], expr(Z2, P1), {Parse_tree = [P1, (Z is Z1 - Z2)]}
      ).
10
    term(Z, Parse_tree) -->
      ( factor(Z, Parse_tree)
      | factor(Z1, P1), multfactor([Z, Z1], P2), {Parse_tree = [P1, P2]}
      ).
15
    multfactor([Z, Z1], Parse_tree) -->
      ( ['*'], term(Z2, P1), {Parse_tree = [P1, (Z is Z1 * Z2)]}
      | ['/'], term(Z2, P1), {Parse_tree = [P1, (Z is Z1 / Z2)]}
      ).
20
    factor(Z, Parse_tree) -->
      ( [X], {integer(X), Parse_tree = [(Z = X)]}
      | [X], {variable(X), Parse_tree = [(Z = X)]}
      | ['('], expr(Z, Parse_tree), [')']
25    ).
```

Notazione esplicita

```
expr(Z, Parse_tree, S0, S) :-
 term(Z, Parse_tree, S0, S).
expr(Z, Parse_tree, S0, S) :-
  term(Z1, P1, S0, S1),
  addterm([Z, Z1], P2, S1, S),
  Parse_tree = [P1, P2].

addterm([Z, Z1], Parse_tree, S0, S) :-
 S0 = ['+'|S1],
 expr(Z2, P1, S1, S),
 Parse_tree = [P1, (Z is Z1 + Z2)].
addterm([Z, Z1], Parse_tree, S0, S) :-
 S0 = ['-'|S1],
 expr(Z2, P1, S1, S),
 Parse_tree = [P1, (Z is Z1 - Z2)].

term(Z, Parse_tree, S0, S) :-
  factor(Z, Parse_tree, S0, S).
term(Z, Parse_tree, S0, S) :-
  factor(Z1, P1, S0, S1), multfactor([Z, Z1], P2, S1, S),
  Parse_tree = [P1, P2].

multfactor([Z, Z1], Parse_tree, S0, S) :-
  S0 = ['*'|S1], term(Z2, P1, S1, S), Parse_tree = [P1, (Z is Z1 * Z2)].
  S0 = ['/'|S1], term(Z2, P1, S1, S), Parse_tree = [P1, (Z is Z1 / Z2)].

factor(Z, Parse_tree, S0, S) :-
  S0 = [X|S],
  integer(X),
  Parse_tree = [(Z = X)].
factor(Z, Parse_tree, S0, S) :-
  S0 = [X|S],
  variable(X),
  Parse_tree = [(Z = X)].
factor(Z, Parse_tree, S0, S) :-
  S0 = ['('|S1], expr(Z, Parse_tree, S1, S2), S2 = [')'|S].
```

A.2 ExprTokenizer

Notazione implicita

```
start_tokenizer(Tokens) -->
  ( ['-', X], {is_digit(X)}, number_tail(Tokens, ['-', X])
  | ['('], start_tokenizer(Next_Tokens), {Tokens = ['('|Next_Tokens]}
  | [X], {is_digit(X)}, number_tail(Tokens, [X])
  | [X], {is_char(X)}, variable_tail(Tokens, [X])
  | [], {Tokens = []}
  ).

expr_tokenizer(Tokens) -->
  ( ['('], start_tokenizer(Next_Tokens), {Tokens = ['('|Next_Tokens]}
  | [')'], expr_tokenizer(Next_Tokens), {Tokens = [')'|Next_Tokens]}
```

```
    | [X], {is_operator(X), Tokens = [X|Next_Tokens]}, expr_tokenizer(Next_Tokens)
    | [X], {is_digit(X), Tokens = [X|Next_Tokens]}, number_tail(Next_Tokens, [X])
    | [X], {is_char(X), Tokens = [X|Next_Tokens]}, variable_tail(Next_Tokens, [X])
    | [], {Tokens = []}
    ).

number_tail(Tokens, Tmp) -->
    ( [X], {is_digit(X), concat(Tmp, [X], New_Tmp)}, number_tail(Tokens, New_Tmp)
    | [')'], {create_number(Tmp, Element), Tokens = [Element|Next_Tokens]},
      expr_tokenizer(Next_Tokens)
    | [X], {is_operator(X), create_number(Tmp, Element),
      Tokens = [Element|Next_Tokens]}, expr_tokenizer(Next_Tokens)
    | {create_number(Tmp, Element), Tokens = [Element]}
    ).

variable_tail(Tokens, Tmp) -->
    ( [C], {is_char(C), concat(Tmp, [C], New_Tmp)}, variable_tail(Tokens, New_Tmp)
    | [X], {is_digit(X), concat(Tmp, [X], New_Tmp)}, variable_tail(Tokens, New_Tmp)
    | ['_'], {concat(Tmp, ['_'], New_Tmp)}, variable_tail(Tokens, New_Tmp)
    | [')'], {create_atom(Tmp, Element), Tokens = [Element|Next_Tokens]},
      expr_tokenizer(Next_Tokens)
    | [X], {is_operator(X), create_atom(Tmp, Element),
      Tokens = [Element|Next_Tokens]}, expr_tokenizer(Next_Tokens)
    | {create_atom(Tmp, Element), Tokens = [Element]}
    ).
```

Notazione esplicita

```
  start_tokenizer(Tokens, S0, S) :-
    S0 = ['-', X|S1], is_digit(X), number_tail(Tokens, ['-', X], S1, S).
  start_tokenizer(Tokens, S0, S) :-
    S0 = ['('|S1], start_tokenizer(Next_Tokens, S1, S),
    Tokens = ['('|Next_Tokens].
  start_tokenizer(Tokens, S0, S) :-
    S0 = [X|S1], is_digit(X), number_tail(Tokens, [X], S1, S).
  start_tokenizer(Tokens, S0, S) :-
    S0 = [X|S1], is_char(X), variable_tail(Tokens, [X], S1, S).
  start_tokenizer(Tokens, S, S) :-
    Tokens = [].

  expr_tokenizer(Tokens, S0, S) :-
    S0 = ['('|S1], start_tokenizer(Next_Tokens, S1, S),
    Tokens = ['('|Next_Tokens].
  expr_tokenizer(Tokens, S0, S) :-
    S0 = [')'|S1], expr_tokenizer(Next_Tokens, S1, S),
    Tokens = [')'|Next_Tokens].
  expr_tokenizer(Tokens, S0, S) :-
    S0 = [X|S1], is_operator(X), Tokens = [X|Next_Tokens],
    expr_tokenizer(Next_Tokens, S1, S).
  expr_tokenizer(Tokens, S0, S) :-
    S0 = [X|S1], is_digit(X), Tokens = [X|Next_Tokens],
    number_tail(Next_Tokens, [X], S1, S).
  expr_tokenizer(Tokens, S0, S) :-
    S0 = [X|S1], is_char(X), Tokens = [X|Next_Tokens],
    variable_tail(Next_Tokens, [X], S1, S).
  expr_tokenizer(Tokens, S, S) :-
    Tokens = [].

  number_tail(Tokens, Tmp, S0, S) :-
```

```
    S0 = [X|S1], is_digit(X), concat(Tmp, [X], New_Tmp),
    number_tail(Tokens, New_Tmp, S1, S).
number_tail(Tokens, Tmp, S0, S) :-
    S0 = [')'|S1], create_number(Tmp, Element), Tokens = [Element|Next_Tokens],
    expr_tokenizer(Next_Tokens, S1, S).
number_tail(Tokens, Tmp, S0, S) :-
    S0 = [X|S1], is_operator(X), create_number(Tmp, Element),
    Tokens = [Element|Next_Tokens], expr_tokenizer(Next_Tokens, S1, S).
number_tail(Tokens, Tmp, S, S) :-
    create_number(Tmp, Element), Tokens = [Element].

variable_tail(Tokens, Tmp, S0, S) :-
    S0 = [C|S1], is_char(C), concat(Tmp, [C], New_Tmp),
    variable_tail(Tokens, New_Tmp, S1, S).
variable_tail(Tokens, Tmp, S0, S) :-
    S0 = [X|S1], is_digit(X), concat(Tmp, [X], New_Tmp),
    variable_tail(Tokens, New_Tmp, S1, S).
variable_tail(Tokens, Tmp, S0, S) :-
    S0 = ['_'|S1], concat(Tmp, ['_'], New_Tmp),
    variable_tail(Tokens, New_Tmp, S1, S).
variable_tail(Tokens, Tmp, S0, S) :-
    S0 = [')'|S1], create_atom(Tmp, Element), Tokens = [Element|Next_Tokens],
    expr_tokenizer(Next_Tokens, S1, S).
variable_tail(Tokens, Tmp, S0, S) :-
    S0 = [X|S1], is_operator(X), create_atom(Tmp, Element),
    Tokens = [Element|Next_Tokens], expr_tokenizer(Next_Tokens, S1, S).
variable_tail(Tokens, Tmp, S, S) :-
    create_atom(Tmp, Element), Tokens = [Element].
```

A.3 Translator

```
package dcg;

import java.lang.String;
import JSetL.*;

public class Translator
{

  static SolverClass t_solver = new SolverClass();  // solver per tokenizer
  static SolverClass p_solver = new SolverClass();  // solver per parser

  ExprTokenizer expr_tokenizer = new ExprTokenizer(t_solver);
  ExprParser expr_parser = new ExprParser(p_solver);

  public Constraint translate_expr(String s_expr, IntLVar result)
  throws Failure {
    LList l_expr = LList.empty();
    for(int i = 0; i < s_expr.length(); i++)
         l_expr = l_expr.insn(s_expr.charAt(i));
    l_expr.setName("l_expr").output();              //DEBUG

    LList token_list = new LList();
    t_solver.solve(expr_tokenizer.scan(token_list, l_expr));
    token_list.setName("token_list").output();      //DEBUG
```

```
25
    Constraint c = new Constraint();

    if (token_list.getSize() > 0) {
      LList parse_tree = new LList().setName("parse_tree");
30    p_solver.solve(expr_parser.parse(result, parse_tree, token_list));
      k.output();                                          //DEBUG

      // c = codeGeneration(parse_tree, result)
      c = parseTree_toConstraint(parse_tree);
35    result.setConstraint(c);
    }
    return c;
  }

40  private Constraint parseTree_toConstraint(LList parse_tree)
  throws Failure {
        Constraint c = new Constraint();
  SolverClass solverAux = new SolverClass();
        while (!parse_tree.isEmpty()) {
45      LList next = LList.empty();
      LList support = new LList(parse_tree).setName("support");
      while(!support.isEmpty()) {
        LList support2 = new LList();
        LVar x = new LVar();
50      solverAux.clearStore();
        solverAux.solve(support.eq(support2.ins1(x)));
        if (x.getValue() instanceof LList) {
          LList tmp = (LList)x.getValue();
          while(!tmp.isEmpty()) {
55          LVar y = new LVar();
            LList x2 = new LList();
            solverAux.clearStore();
            solverAux.solve(tmp.eq(x2.ins1(y)));
            next = next.ins1(y);
60          tmp = x2;
          } // end while
        }  // end if
        else if (x.getValue() instanceof Constraint){
            c.and((Constraint)x.getValue());
65      }  // end else_if
        support = support2;
      } // end while
      parse_tree = next;
    } // end while
70  return c;
  }

  public IntLVar expr(String s_expr) throws Failure {
            IntLVar result = new IntLVar();
75          translate_expr(s_expr, result);
            return result;
          }

}
```

Ringraziamenti

Varie persone mi hanno aiutato nel raggiungere questo mio traguardo e mi sembra doveroso dedicare a loro una pagina di questo lavoro.
Un primo ringraziamento va ai miei genitori per avermi permesso di proseguire lungo questa strada e a mia sorella per avermi sempre sostenuto.
Ringrazio inoltre il Prof. Gianfranco Rossi per essere stato sempre disponibile durante tutto il periodo di tirocinio e tesi e per aver dedicato così tanto tempo al mio lavoro nonostante i mille impegni.
Grazie a Roberto Amadini per avermi aiutato a muovere i primi passi con JSetL e a Fabio Biselli per i vari consigli relativi a LaTeX (oltre che per gli innumerevoli passaggi in macchina).
Grazie a Andrea Boscolo per la disponibilità ad ascoltare, sempre e comunque, qualsiasi problema di carattere informatico e non-informatico.
Grazie a Ilenia: il suo non è stato un aiuto tecnico ma sicuramente è stato un importantissimo supporto morale.
Grazie a Stefania, Tino, Tommaso, Daro, Michele, Gianni, Tsabo, Gioppa, Ruggero, Eleonora, Cello e in generale a tutti gli amici conosciuti durante questo periodo per essere stati presenti quando ne avevo bisogno e per avermi permesso di condividere quest'esperienza insieme a loro.