UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI
MATEMATICA e INFORMATICA

Corso di Laurea in Informatica

Tesi di Laurea

Progettazione e sviluppo di un preprocessore Java per la libreria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Lorenzo Furini

Anno Accademico 2013/2014

Ai miei genitori,

Antonio e Rosetta

Indice

Introduzione

Il *preprocessing* [4] è una fase dell'elaborazione del codice sorgente che precede la compilazione.
Il programma che esegue questa elaborazione viene detto *precompilatore o preprocessore*.
Il risultato di questa operazione è ancora codice sorgente, ovvero la precompilazione non include nessuno dei passaggi necessari alla generazione del codice macchina, che rimane un compito del compilatore.
Un preprocessore opera sul codice sorgente, prima di qualsiasi parsing, in base a regole definite dallo sviluppatore dette direttive.
Un preprocessore può essere di due tipi:

- *preprocessore lessicale* che sostituisce sequenze di caratteri trasformate in token con altre sequenze di caratteri.
  L'esempio più comune è il *preprocessore C*, che prende le istruzioni che iniziano con '#' come direttive e svolge principalmente inclusioni di pezzi di codice sorgente e sostituzioni di stringhe con altre stringhe.

- *preprocessore sintattico* che opera sul linguaggio utilizzato agendo sull'albero di sintassi.
  Un esempio è il *preprocessore* del linguaggio *Lisp*, che utilizza le direttive per estendere il linguaggio da una semplice programmazione funzionale ad una programmazione Object Oriented.

Una soluzione interessante è quella in [6], in cui le direttive del preprocessore sono inserite nei commenti Java.
Questo permette di mantenere la piena compatibilità con la catena di sviluppo standard di Java e rendere il loro utilizzo trasparente in tutti gli IDE Java.

In questa tesi affrontiamo il problema di realizzare un preprocessore, essenzialmente di tipo lessicale, per il linguaggio Java, allo scopo di facilitare l'utilizzo delle funzionalità offerte dalla libreria `JSetL`.

`JSetL` [1][2][3][5] è una libreria Java che offre costrutti tipici della programmazione dichiarativa, utilizzabili nello sviluppo di programmi Object Oriented:

- Variabili logiche
- Liste ed Insiemi parzialmente specificati
- Unificazione
- Risolutore di vincoli
- Vincoli implementati dall'utente
- Non-determinismo

L'utilizzo delle funzionalità fornite da `JSetL` richiede il rispetto di diverse convenzioni di programmazione.
Questo infatti è un problema comune per l'integrazione dei costrutti della programmazione logica tramite libreria: l'uso di una notazione particolarmente scomoda e la necessità di seguire precisi schemi di programmazione che implementino le funzionalità desiderate.
L'obbiettivo della tesi è quello di integrare il linguaggio Java con costrutti sintattici in grado di semplificare il codice, rendendolo più conciso e quindi più comprensibile e manutenibile, per facilitare l'utilizzo di `JSetL` all'interno di un programma Java.
Questo avverrà attraverso l'uso di opportuni commenti strutturati trattati da un programma di *preprocessing* creato ad-hoc, il tutto senza dover estendere il linguaggio Java stesso.

Il **lavoro di tesi** è stato così organizzato:

Il **capitolo 1** presenta la libreria `JSetL`, soffermandosi in particolare sulle funzionalità utilizzate e sul concetto di *non-determinismo*.

Il **capitolo 2** presenta i costrutti del *linguaggio esteso*: costanti collezione, costrutti per la gestione del non-determinismo e costrutti per il calcolo delle soluzioni.

Il **capitolo 3** mostra alcuni esempi di programmi Java + `JSetL` che utilizzano i costrutti presentati nel capitolo precedente, con la relativa traduzione.

Il **capitolo 4** presenta lo strumento Java che esegue la traduzione, oggetto di questa tesi, completo della sua effettiva implementazione.
Viene inoltre mostrata l'architettura software in cui esso opera.

Il **capitolo 5** presenta `Apache ANT`, lo strumento Java utilizzato per l'automatizzazione del lavoro di traduzione e mostra l'implementazione `XML` dello script `ANT` in uso nel nostro programma.

Il **capitolo 6** contiene le conclusioni ed eventuali sviluppi futuri.

Capitolo 1

La libreria JSetL

In questo capitolo presentiamo `JSetL`, una libreria java che implementa le funzionalità tipiche della programmazione logica all'interno di un contesto object-oriented, come *variabili logiche, liste ed insiemi logici anche parzialmente specificati, unificazione e risoluzione di vincoli, non-determinismo ed implementazione di nuovi vincoli definiti dall'utente.*
La presentazione si limita alle funzionalità di `JSetL` utilizzate in questo lavoro.

1.1 Variabili logiche

1.1.1 Classe `LVar`

Una *variabile logica* [1][2][3] è un'istanza della classe `LVar`. Possiamo associare ad una `LVar` un valore ed un nome esterno. Quando ad una *variabile logica* viene associato un valore, la variabile è detta *bound*, in caso contrario sarà *unbound*. Si puo assegnare un valore ad una *variabile logica* soltanto attraverso vincoli chiamati sulla stessa.

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 valore.

Metodi
Analizziamo alcuni metodi di nostro interesse appartenenti a questa classe:

`LVar setName(String extName)`
Assegna alla *variabile logica* d'invocazione il nome esterno `extName` e ritorna la variabile stessa.

`Constraint eq(Object o)`
Restituisce il vincolo $this = o$ che unifica la *variabile logica* d'invocazione con l'oggetto `o`.
Diciamo che due *variabili logiche* `x` e `y` sono *equivalenti* se sono state unificate con successo con il vincolo `x.eq(y)`.

`Constraint neq(Object o)`
Restituisce il vincolo $this \neq o$ che richiede alla *variabile logica* d'invocazione di essere diversa dall'oggetto `o`.

`Constraint in(LSet s)`
Restituisce il vincolo $this \in s$ che richiede l'*appartenenza* della *variabile logica* d'invocazione all'*insieme logico* `s`. Quando questo vincolo viene risolto, unifica *non-deterministicamente* il valore della *variabile logica* ad ogni elemento dell'*insieme logico* `s`. Il vincolo ha successo se almeno una unificazione ha successo.

`Constraint nin(LSet s)`
Restituisce il vincolo $this \notin s$ che richiede la *non appartenenza* della *variabile logica* d'invocazione all'*insieme logico* `s`.

Esiste una specializzazione della classe `LVar` che serve per rappresentare i numeri interi, la classe `IntLVar` [1][2][3].
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. Specializzando la classe `LVar`, mantiene tutti i suoi metodi e ne mette altri a disposizione.

1.2 Collezioni logiche

1.2.1 Classe `LList`

Una *lista logica* `l` è una speciale variabile logica il cui valore è una coppia `<elems,rest>`, dove `elems` è una lista $[e_0, ..., e_n]$, con $n \geq 0$ e `rest` è una lista che può essere *vuota* o *unbound* che rappresenta la rimanenza di `l`.
Quando `rest` è una lista *unbound* oppure `elems` contiene elementi *unbound*, `l` si dice *parzialmente specificata*.
Una *lista logica* è un'istanza della classe `LList`, la quale estende la classe `LCollection`.
Anche se non estende direttamente la classe `LVar`, la classe `LList` mutua da essa molti metodi.

Costruttori

`LList()`
`LList(String extName)`
Crea una *lista logica unbound*; `extName`, se specificato, viene assegnato come nome esterno alla lista, in caso contrario le viene assegnato automaticamente il nome esterno di default.

`LList(LList l)`
`LList(String extName, LList l)`
Crea una *lista logica bound* assegnandole `l` come valore.
È l'equivalente di creare una *lista logica unbound* `x` ed unificarla con `l` con il vincolo `x.eq(l)`.

Metodi
Analizziamo alcuni metodi di nostro interesse appartenenti a questa classe:

`LList setName(String extName)`
Assegna alla *lista logica* d'invocazione il nome esterno `extName` e ritorna la lista stessa.

`LList empty()`
Ritorna la *lista logica vuota*.

`LList ins(Object o)`
Ritorna la *lista logica* d'invocazione, con l'oggetto `o` in prima posizione.

`Constraint eq(LList l)`
Restituisce il vincolo $this = l$ che unifica la *lista logica* d'invocazione con `l`.

`Constraint neq(LList l)`
Restituisce il vincolo $this \neq l$ che richiede alla *lista logica* d'invocazione di essere diversa da `l`.

1.2.2 Classe `LSet`

Un *insieme logico* `s` è una speciale variabile logica il cui valore è una coppia `<elems,rest>`, dove `elems` è un insieme $\{e_0, ..., e_n\}$, con $n \geq 0$, e `rest` è un insieme che può essere *vuoto* o *unbound* e rappresenta la rimanenza di `s`.
Quando `rest` è un *insieme logico unbound* oppure `elems` contiene elementi *unbound*, `s` si dice *parzialmente specificato*.
Un *insieme logico* è un'istanza della classe `LSet`, la quale estende la classe `LCollection`. Anche se non estende direttamente la classe `LVar`, la classe `LSet` mutua da essa molti metodi.
Insiemi e liste logiche sono simili sotto molti aspetti; la principale differenza è che, mentre per quanto riguarda le liste, l'ordinamento e la ripetizione degli elementi che la compongono ha importanza, per gli insiemi non ne ha.
I metodi a disposizione degli oggetti `LSet` sono praticamente gli stessi degli `LList`, ma applicati ad oggetti `LSet`.

Costruttori

`LSet()`
`LSet(String extName)`
Crea un *insieme logico unbound*; `extName`, se specificato, viene assegnato come nome esterno all'insieme, in caso contrario gli viene assegnato automaticamente il nome esterno di default.

`LSet(LSet s)`
`LSet(String extName, LSet s)`
Crea un *insieme logico bound* assegnandogli `s` come valore.
È l'equivalente di creare un *insieme logico unbound* `x` ed unificarlo con `s`, attraverso il vincolo `x.eq(s)`.

Metodi
Analizziamo alcuni metodi di nostro interesse appartenenti a questa classe:

`LSet setName(String extName)`
Assegna all'*insieme logico* d'invocazione il nome esterno `extName` e ritorna l'insieme stesso.

`LSet empty()`
Ritorna l'*insieme logico vuoto*.

`LSet ins(Object o)`
Ritorna l'*insieme logico* d'invocazione, con l'oggetto `o` in prima posizione.

`Constraint eq(LSet s)`
Restituisce il vincolo $this = s$ che unifica l'*insieme logico* d'invocazione con `s`.

`Constraint neq(LSet s)`
Restituisce il vincolo $this \neq s$ che richiede all'*insieme logico* di invocazione di essere diverso da `s`.

1.3 Vincoli

I *vincoli*, istanze della classe `Constraint` [1][2][3], sono creati dai costruttori di questa classe o generati da metodi di altre classi, ad esempio `eq(LVar)`.
Sono relazioni tra gli *oggetti logici* definiti precedentemente, che vengono gestite e risolte da un *risolutore di vincoli*.
Possono interessare anche *oggetti logici unbound*, il che ci permette di risolvere *vincoli* che interessano *liste e insiemi parzialmente specificati*. Essi possono essere di due tipi:

- `Vincoli atomici`, quindi della forma $o_1.op(o_2, ..., o_n)$ con $o_1, ..., o_n$ oggetti logici e `op` un metodo che ritorni un oggetto `Constraint`.

- `Vincoli composti`, quindi della forma $v_1.and(v_2)$ oppure $v_1.or(v_2)$ con v_1, v_2 vincoli.

I *vincoli* sono risolti utilizzando un *risolutore di vincoli*, che in `JSetL` è un'istanza della classe `SolverClass` [1][2][3].
Questa classe fornisce i metodi per aggiungere *vincoli* al *constraint store(cs)*, provare a soddisfarli e trovare tutte le possibili soluzioni ad essi.

I *vincoli* memorizzati nel *cs* sono considerati in *and logico* tra di loro.

Costruttori

`SolverClass()`
Crea un oggetto di tipo `SolverClass`; il suo *constraint store* è inizialmente *vuoto*.

Metodi
Analizziamo alcuni metodi di nostro interesse appartenenti a questa classe:

`void add(Constraint c)`
Aggiunge il *vincolo* `c` al *constraint store*, del *risolutore* d'invocazione.

`boolean check(Constraint c)`
Se `C` è il *constraint store* del *risolutore* d'invocazione, controlla se il *vincolo* $C \wedge c$ è *risolvibile*, se lo è ritorna `true`, altrimenti `false`.
Se il *vincolo* è *risolvibile*, esso viene risolto e il vincolo $C \wedge c$ e gli *oggetti logici* che lo compongono vengono semplificati, se è invece *irrisolvibile*, il *vincolo* e gli *oggetti logici* rimangono immutati.

`void solve(Constraint c)`
Come `check(Constraint c)`, ma se il *vincolo* $C \wedge c$ è *irrisolvibile*, viene sollevata una eccezione di fallimento.

`boolean nextSolution()`
Se chiamato dopo una `check`, una `solve` o un'altra `nextSolution`, prova a calcolare un'altra soluzione per i *vincoli nel constraint store*, se esiste ritorna `true`, altrimenti ritorna `false`.

1.4 La classe NewConstraintsClass

`JSetL` permette all'utente di definire nuovi vincoli, anche *non-deterministici.*
I *vincoli definiti dall'utente* vengono implementati come metodi all'interno di una classe che estende la `NewConstraintsClass` [1][2][3].
Una volta che l'oggetto della nuova classe è stato creato, si possono usare i *vincoli* che offre come si farebbe con quelli predefiniti.
Il costruttore della nuova classe ha un parametro, il *risolutore di vincoli*, che utilizzeremo per aggiungere al *constraint store* attraverso il metodo `add(c)`,

per poi risolvere attraverso i metodi `check(c)` o `solve(c)`, i *vincoli definiti dall'utente* come quelli predefiniti.

```
public class MyOps extends NewConstraintsClass {
        public MyOps(SolverClass currentSolver) {
                super(currentSolver);
        }
        // metodi che implementano i nuovi vincoli definiti dall'utente
}
```

Vogliamo, ad esempio, implementare i *vincoli* `op1(v1,v2)` e `op2(v3)` con `v1,v2,v3` rispettivamente di tipo `t1,t2,t3`; creiamo quindi i metodi, che restituiscano i *vincoli*, da inserire nel *constraint store*, del *risolutore di vincoli*. I seguenti metodi, servono solo per aggiungere i *vincoli* `op1` e `op2` al *constraint store*, in cui il nome del *vincolo* è identificato da una stringa, specificata come parametro del costruttore. Queste definizioni formano l'*interfaccia utente*.

```
...
public Constraint op1(t1 v1, t2 v2) {
        return new Constraint("op1", v1, v2);
}
public Constraint op2(t3 v3) {
        return new Constraint("op2", v3);
}
...
```

Dopo queste definizioni è necessario implementare i metodi veri e propri, di tipo `private void`, che specificano, di fatto, le operazioni che vogliamo eseguire nei *vincoli*, i quali avranno, come unico parametro, un oggetto di tipo `Constraint`.
Iniziamo con l'acquisizione dei parametri, che vengono recuperati dal *vincolo* stesso tramite il metodo `getArg(int k)`, dove `k` è l'indice del parametro.

```
...
private void op1(Constraint c) {
// acquisizione parametri vincolo op1
        t1 v1 = (t1)c.getArg(1);
        t2 v2 = (t2)c.getArg(2);

// implementazione vincolo op1
```

```
...
}
private void op2(Constraint c) {
// acquisizione parametri vincolo op2
        t3 v3 = (t3)c.getArg(1);

// implementazione vincolo op2
...
}
...
```

Il metodo `user_code`, anch'esso obbligatoriamente con un unico parametro di tipo `Constraint`, ha il compito di associare il *vincolo* richiamato dal *risolutore*, al metodo della classe, che deriva `NewConstraintsClass`, che lo implementa. Questa definizione forma l'*interfaccia solver*.

```
protected void user_code(Constraint c)
throw Failure, NotDefConstraintException {
        if (c.getName()=="op1")
                op1(c);
        else if (c.getName()=="op2")
                op2(c);
        else throw new NotDefConstraintException();
}
```

Definire nuovi vincoli in `JSetL` richiede che ogni risultato da essi calcolato debba essere restituito attraverso i suoi parametri.
L'uso di *oggetti logici unbound* come argomenti, variabili, liste e insiemi, risolve in modo molto semplice questo problema.
Un *oggetto logico* specificato come parametro, può essere utilizzato sia come input che come output, in base al fatto che sia *bound* o meno alla chiamata del metodo, perciò, un *oggetto logico bound* può soltanto agire da input, un *oggetto logico unbound* può soltanto agire da output e un *oggetto logico parzialmente specificato* può agire sia da input che da output.

Una volta che l'oggetto della classe `MyOps` è stato creato, i metodi che la compongono sono *vincoli* a tutti gli effetti, possono quindi essere aggiunti al *constraint store* e risolti dal *risolutore di vincoli* come quelli predefiniti della libreria.
In questo esempio

```
...
MyOps o = new MyOps(solver);
solver.solve(o.op1(v1,v2));
...
```

viene creato un oggetto di tipo `MyOps`, a cui assegniamo il *risolutore di vincoli* `solver`, quindi viene aggiunto al *constraint store* il *vincolo* `op1`, definito nella classe `MyOps`, e infine risolto.

1.5 Sfruttare il non-determinismo

Una *computazione deterministica* è tale se dato uno stato e un dato in ingresso, posso andare in un unico nuovo stato. Intuitivamente, in una *computazione non deterministica* dato uno stato e un dato in ingresso posso andare a finire in un insieme di stati; in altre parole ogni stato può avere più di una transizione per ogni dato in ingresso. Ma vediamo questi concetti in modo formale:

`C` è una *computazione* [4] 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`, ... ,`S_i`, ... }` una *computazione*:

- `C` si dice *finita* se e solo se

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

  quindi S_k esiste ed è unico, ed è l'ultimo stato di esecuzione.

- `C` si dice *deterministica* se e solo se

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

  dove il simbolo $\rightarrow$ viene detto *passo di computazione*, quindi preso un qualsiasi stato di computazione S_i ed applicando un passo, la computazione va nel singolo stato S_{i+1}.

- In contrapposizione, `C` si dice *non-deterministica* se e solo se

  $$\forall i . \exists S_{i+1} \in \{S_{i1}, \ldots, S_{ik}\} : S_i \rightarrow S_{i+1}$$

> quindi preso un qualsiasi *stato di computazione* S_i ed applicando un singolo passo, la *computazione* va in al più k stati.

L' uso di *oggetti logici* è fondamentale per sfruttare il *non-determinismo*.
Infatti, se un oggetto è coinvolto in una computazione *non-deterministica*, è necessario ripristinare lo stato che aveva prima di entrare nell'ultimo *punto di scelta* incontrato, in modo tale che la computazione possa fare *backtracking* e provare un percorso alternativo in caso di fallimento.
In `JSetL`, questo è ottenuto permettendo al *risolutore di vincoli* di salvare automaticamente, e successivamente ripristinare, lo stato di tutti gli *oggetti logici* coinvolti nella *computazione*.
Siccome un *oggetto logico* è caratterizzato dal fatto che il suo valore, se esiste, non può essere modificato attraverso *side effects*, ma soltanto utilizzando *vincoli* che lo riguardino, è necessario utilizzare solo *oggetti logici*, in particolare *variabili logiche*, per gli oggetti coinvolti in una *computazione non-deterministica*.

Per implementare il concetto di *non-determinismo*, i programmi che utilizzano la libreria `JSetL`, sfruttano meccanismi di *backtracking* attraverso *punti di scelta*, in grado di salvare lo stato della *computazione* e degli *oggetti logici* coinvolti, e provare un percorso alternativo in caso di fallimento dell'esecuzione.
I *vincoli* in `JSetL` sono risolti in modo *non deterministico*, l'ordine di risoluzione non ha importanza e il calcolo delle stesse viene eseguito *non-deterministicamente*, grazie all'utilizzo di *punti di scelta* e facendo *backtracking*.
Attraverso la creazione di *vincoli definiti dall'utente*, di cui abbiamo discusso nel paragrafo precedente, è possibile sfruttare le funzionalità offerte da `JSetL` per implementare efficacemente il *non-determinismo*.

`int getAlternative()`
Ritorna un intero associato al *vincolo* d'invocazione, che viene usato per contare le soluzioni alternative calcolate in modo *non-deterministico*.

`void fail()`
Forza il fallimento del *vincolo* d'invocazione. Questo causa il *backtracking* del *risolutore* al più vicino *punto di scelta*; se non ne esistono, viene sollevata un'eccezione di fallimento.

`void addChoicePoint(Constraint c)`
Aggiunge un *punto di scelta* nel *risolutore di vincoli* d'invocazione; questo

permette di fare *backtracking* e riconsiderare il *vincolo* `c` se si verifica un fallimento.
Quando si effettua un *backtracking*, lo stato del *vincolo* `c`, cioè il valore degli *oggetti logici* coinvolti, viene ripristinato com'era prima che la risoluzione entrasse nel *punto di scelta*.
L'indice associato al vincolo `c`, che rappresenta il numero di alternative *non-deterministiche*, viene incrementato di uno.

Segue una implementazione *non-deterministica* del *vincolo abs*, che calcola il valore assoluto di un numero intero:

```
private void abs(Constraint c) throws Failure {
        IntLVar x = (IntLVar)c.getArg(1);
        IntLVar y = (IntLVar)c.getArg(2);
        switch(c.getAlternative()) {
        case 0: // x >= 0 e x = y
                Solver.addChoicePoint(c);
                Solver.add(x.ge(0).and(x.eq(y)));
        break;
        case 1: // x < 0 e x = 0 - y
                Solver.add(x.lt(0).and(x.eq(new IntLVar(0).sub(y))));
        break;
        }
}
```

Assumendo di aver implementato questo *vincolo* come un metodo, all'interno di una classe di nome `MyOps`, che estende `NewConstraintsClass`, un esempio d'uso del *vincolo* è il seguente:

```
IntLVar x = new IntLVar("x");
IntLVar y = new IntLVar("y",3);
Solver.check(MyOps.abs(x,y));
do {
        x.output();
} while(Solver.nextSolution(y));
```

che genera il seguente output a video:

x = 3
x = -3

Capitolo 2

Linguaggio esteso

Con *linguaggio esteso* si intende l'introduzione di una concisa e più conveniente notazione sintattica, che faciliti l'utilizzo delle funzionalità offerte dalla libreria `JSetL` nei programmi Java, basata sull'uso di speciali commenti che chiameremo *direttive* `JSetL`.
Il codice creato è ancora Java e quindi sintatticamente trattabile dal compilatore, ma prima dell'effettiva compilazione ed esecuzione, le classi che usano il *linguaggio esteso* vengono tradotte dal *preprocessore* in classi formate da codice Java+`JSetL`.
In questo capitolo saranno analizzati i costrutti facenti parte del *linguaggio esteso*, che saranno quindi tradotti dal *preprocessore* in fase di precompilazione.

2.1 Costanti collezione

Le costanti collezione sono gli *oggetti logici* fondamentali di `JSetL`, ma definiti in modo molto più immediato ed intuitivo per l'utente finale.
Possono essere usate ovunque possa apparire un *oggetto logico* di tipo `LList` o `LSet`.
Le forme sintattiche accettate in input dal *preprocessore* sono le seguenti:

Liste

- $/*[e_0, ..., e_n]*/$
- $/*[e_0, ..., e_n|r]*/$ $(n \geq 0)$
- $/*[e_0, ..., e_n]*/$"extname"

- $/*[e_0, ..., e_n|r]*/$"extname" $(n \geq 0)$

$e_0, ..., e_n$ sono gli elementi che compongono la *lista logica*; se non ci sono elementi verrà creata la lista *vuota*.
`r` è una *lista logica unbound* che indica il *resto*.
`extname` è il nome esterno della *lista logica*; se presente, verrà visualizzata in output con questo nome.

Insiemi

- $/*\{e_0, ..., e_n\}*/$
- $/*\{e_0, ..., e_n|r\}*/$ $(n \geq 0)$
- $/*\{e_0, ..., e_n\}*/$"extname"
- $/*\{e_0, ..., e_n|r\}*/$"extname" $(n \geq 0)$

$e_0, ..., e_n$ sono gli elementi che compongono l'*insieme logico*; se non ci sono elementi verrà creato l'insieme *vuoto*.
`r` è un *insieme logico unbound* che indica il *resto*.
`extname` è il nome esterno dell'*insieme logico*; se presente, verrà visualizzato in output con questo nome.

Mostriamo qui alcuni esempi di utilizzo delle *Costanti collezione*.

Liste

```
LList x = new LList(/*[1,2,3|r]*/''x'');
```
Crea una nuova *lista logica parzialmente definita* di nome x, con 1, 2, 3 come elementi di testa, con una *lista logica* `r` come resto. Il suo nome esterno è "x".

```
LList z = new LList(/*[4,5]*/''z'');
```
Crea una nuova *lista logica* di nome z, con 4, 5 come elementi. Il suo nome esterno è "z".

```
LList y = new LList(/*[]*/''y'');
```
Crea una nuova *lista logica vuota* di nome y. Il suo nome esterno è "y".

```
solver.add(x.eq(/*[''a'',''b'',''c''|r2]*/''l1''));
solver.add(x.neq(/*[''d'',1,8]*/''l2''));
```

Applicazione dei *vincoli* `eq, neq`. `x` è un oggetto di tipo `LList`.

```
solver.add(x.in(/*[''io'',''tu'',''egli''|r3]*/));
solver.add(x.nin(/*[1,2,''ciao'']*/''l3''));
```
Applicazione dei *vincoli* `in, nin`. `x` è un oggetto di tipo `LVar`.

Insiemi

```
LSet x = new LSet(/*{1,2,3|r}*/''x'');
```
Crea un nuovo *insieme logico parzialmente definito* di nome x, con 1, 2, 3 come elementi di testa, con una *lista logica* `r` come resto. Il suo nome esterno è "x".

```
LSet z = new LSet(/*{4,5}*/''z'');
```
Crea una nuovo *insieme logico* di nome z, con 4, 5 come elementi. Il suo nome esterno è "z".

```
LSet y = new LSet(/*{}*/''y'');
```
Crea una nuovo *insieme logico vuoto* di nome y. Il suo nome esterno è "y".

```
solver.add(x.eq(/*{'a','b','c'|r2}*/''s1''));
solver.add(x.neq(/*{'d',1,8}*/''s2''));
```
Applicazione dei *vincoli* `eq, neq`. `x` è un oggetto di tipo `LSet`.

```
solver.add(x.in(/*{''io'',''tu'',''egli''|r3}*/''s3''));
solver.add(x.nin(/*{1,2,''ciao''}*/));
```
Applicazione dei *vincoli* `in, nin`. `x` è un oggetto di tipo `LVar`.

Limitazioni e note di utilizzo

Mentre `JSetL` prevede l'uso di *strutture logiche* annidate, nel lavoro di traduzione delle *costanti collezione* implementato da questa versione del *preprocessore* ciò non viene permesso, ed in caso l'utente ne facesse uso verrà lanciata un'eccezione e il processo di traduzione terminerà.

Per quanto riguarda la traduzione delle costanti collezione come parametri dei vincoli, in questa versione del preprocessore è stata implementata solamente per `eq, neq, in, nin`, ma si può facilmente integrare con i vincoli rimanenti.

Le istruzioni che hanno come parametro una *collezione logica*, che a sua volta è il *resto* di un'altra *collezione logica* definita precedentemente, devono essere precedute dalla direttiva per il preprocessore //%, come in questo esempio:

```
...
solver.add(L1.eq(/*['+'|L2]*/));
//%solver.add(expr(L2, Remain));
...
```

2.2 Costrutti per la creazione di nuovi vincoli

Come abbiamo visto nel capitolo precedente, in `JSetL` è possibile creare nuovi *vincoli definiti dall'utente*.
Il linguaggio esteso introduce nuovi costrutti sintattici per facilitare la definizione di nuovi vincoli in `JSetL`.

- E' necessario definire una classe, la cui effettiva implementazione deve essere preceduta dalla direttiva
  `//%New Constraint Class`

- Il costruttore della classe, con un singolo parametro, serve per inizializzare il *risolutore di vincoli* attualmente in uso nel programma; deve essere preceduto dalla definizione di un oggetto `private` di tipo `SolverClass`, il cui nome deve coincidere con il *risolutore di vincoli* inizializzato nel costruttore

```
//%New Constraint Class
public class MyOps {
        private SolverClass solver;
        public MyOps(SolverClass s) {
                solver = s;
        }
        // metodi che implementano i nuovi vincoli definiti dall'utente
}
```

- I *vincoli definiti dall'utente* sono metodi pubblici di questa classe e dovranno essere preceduti dalla direttiva
  `//%Constraint`
  altrimenti verrà sollevata un'eccezione e il processo di traduzione terminerà.
  Essi dovranno restituire un oggetto di tipo `Constraint` che sarà
  `Constraint.ok` se la risoluzione del *vincolo* è andata a buon fine,
  `Constraint.fail` se, percorrendo tutti i possibili percorsi di unificazione, la risoluzione non è andata a buon fine,
  `Constraint.notSolved` se la risoluzione del vincolo non può essere completata e viene momentaneamente ritardata.

```
//%Constraint
public Constraint op1(t1 v1, t2 v2) {
        // implementazione vincolo op1
}
//%Constraint
public Constraint op2(t3 v3) {
        // implementazione vincolo op2
}
...
//%Constraint
public Constraint opn(t1 v4, t4 v5, t2 v6) {
        // implementazione vincolo opn
}
```

Nel seguente esempio, il vincolo `number`, implementato attraverso l'apposito metodo, restituisce `Constraint.ok` se la *variabile logica* `n` è *bound* ed il suo valore è un intero, altrimenti restituisce `Constraint.fail`.

```
//%New Constraint Class
public class MyOps {
        private SolverClass solver;
        public MyOps(SolverClass s) {
                solver = s;
        }
        //%Constraint
        public Constraint number(LVar n) {
                if (n.getValue() instanceof Integer)
                        return Constraint.ok;
                else
                        return Constraint.fail;
```

```
        }
        //%Constraint
        ...
}
```

2.3 Costrutti per la gestione del non-determinismo

La gestione dei vincoli *non-deterministici* avviene soprattutto grazie alla creazione e gestione dei *punti di scelta*, esplorazione delle alternative e uso del *backtracking*.

2.3.1 either - orelse

Il *linguaggio esteso* prevede un costrutto per esprimere il *non-determinismo*: `either - orelse`.
Se $S_1, ..., S_n$ con $n \geq 2$, sono sequenze di statement da eseguire in modo *non-deterministico*,

```
//%either{S1}
//%orelse{S2}
...
//%orelse{Sn}
```

implica una scelta *non-deterministica* tra $S_1, ..., S_n$.
L'interpretazione logica di questo costrutto è $S_1 \vee ... \vee S_n$, mentre la sua interpretazione computazionale è l'esplorazione, tramite *backtracking*, di ogni possibile alternativa, finchè una di esse viene risolta con successo.
Lo statement S_1 viene eseguito; se l'esecuzione fallisce, entra in gioco il *backtracking* e viene ripristinato lo stato degli *oggetti logici* coinvolti nella *computazione* all'entrata in S_1, viene cioè creato un *punto di scelta* prima di eseguire S_1, viene quindi eseguito lo statement S_2 e così via fino all'esaurimento di tutti gli statements; se viene riscontrato un fallimento durante l'esecuzione di S_n, l'esecuzione del costrutto fallisce definitivamente.
Il costrutto `either - orelse` può essere utilizzato soltanto all'interno di *nuovi vincoli definiti dall'utente.*

Seguono alcuni esempi di utilizzo.

Questo vincolo definito dall'utente assegna *non-deterministicamente* ad una *variabile logica* gli interi `1,2,3`.

```
//%Constraint
public Constraint ndAssign(IntLVar x) {
        //%either
        {solver.add(x.eq(1));}
        //%orelse
        {solver.add(x.eq(2));}
        //%orelse
        {solver.add(x.eq(3));}
        return Constraint.ok;
}
```

Questo vincolo definito dall'utente calcola *non-deterministicamente* il valore assoluto di un numero intero.

```
//%Constraint
public Constraint ndAbs(IntLVar x, IntLVar y) {
        //%either
        {solver.add(x.ge(0).and(x.eq(y)));}
        //%orelse
        {solver.add(x.lt(0).and(x.eq(new IntLVar(0).sub(y))));}
        return Constraint.ok;
}
```

2.4 Costrutti per il calcolo delle soluzioni

La risoluzione di vincoli *non-deterministici*, propri di `JSetL` o creati dall'utente, può comportare la generazione di più soluzioni per lo stesso insieme di vincoli.

2.4.1 findAll

Il *linguaggio esteso* prevede un costrutto per il calcolo delle soluzioni dei *vincoli* contenuti nel *constraint store*, sfruttando il backtracking: `findAll`.
La sintassi di questo costrutto ha queste forme:

1.
```
//%findAll(c){ corpo }
```

dove `c` è il singolo *vincolo* da risolvere utilizzando un *risolutore* temporaneo, `corpo` è la sequenza di statement da eseguire per tutte le soluzioni di `c`.

2.
```
//%findAll(solver){ corpo }
```

dove `solver` è il *risolutore di vincoli* utilizzato, `corpo` è la sequenza di statement da eseguire per tutte le soluzioni dei vincoli contenuti nel *constraint store*.

Per svolgere il compito, il *risolutore di vincoli* fa *backtracking* tornando al *punto di scelta* creato all'inizio del costrutto per trovare, se esiste, la prossima soluzione.

Seguono alcuni esempi di utilizzo.

Calcolare tutte le permutazioni degli elementi di un *insieme logico* `s` di interi. Il problema può essere considerato come l'unificazione dell'*insieme logico* `s` con un altro *insieme logico parzialmente specificato* di n = |s| *variabili logiche*, tale che $s = x_1, ..., x_n$.
Ogni soluzione di questo *vincolo* garantisce l'assegnamento di valori interi distinti alle variabili $x_1, ..., x_n$ e quindi una possibile permutazione degli interi in `s`.

```
public void allPermutations(LSet s) {
        int n = s.getSize();
        LSet r = LSet.mkLSet(n);
        //%findAll(r.eq(s))
        {
                r.printElems(' ');
                System.out.println();
        }
}
```

Un ulteriore esempio d'uso di questo costrutto, stampa in output tutte le possibili partizioni di una stringa `lin` in due sottostringhe `lout1` e `lout2`. Per ottenere questo risultato utilizziamo il vincolo *non-deterministico* `concat` definito dall'utente.

```
public void allSolutions(LList lin) {
        LList lout1 = new LList();
        LList lout2 = new LList();
        solver.add(concat(lout1,lout2,lin));
        //%findAll(solver)
        {
                lout1.output();
                lout2.output();
        }
}
```

Capitolo 3

Traduzione

In questo capitolo vengono prese in esame le principali funzionalità di traduzione del *preprocessore* `JSetL`, soffermandoci in particolare sul codice generato per ogni specifico input.
In sostanza, andremo a vedere come il *preprocessore* interpreta alcune porzioni di codice sorgente, dette *direttive*.
Faremo infine alcuni esempi di traduzione di interi programmi Java.

3.1 Costanti collezione

Vediamo la traduzione delle *costanti collezioni*, definite nel capitolo 2.1.

Liste

- `/*`$[e_0, ..., e_n]$`*/''extname''`

  la cui *traduzione* è la seguente:

  `LList.empty().ins(`e_n`). ... .ins(`e_0`).setName(''extname'')`

- `/*`$[e_0, ..., e_n|r]$`*/''extname''`

  la cui *traduzione* è la seguente:

  `LList r = new LList(''r'');`

`r.ins(`e_n`). ... .ins(`e_0`).setName(''extname'')`

Sia gli elementi che compongono la *lista logica* che il suo nome esterno sono opzionali, può quindi trattarsi, rispettivamente, di una lista *vuota* o senza nome esterno.

Insiemi

- /*{$e_0, ..., e_n$}*/''extname''

  la cui *traduzione* è la seguente:

  `LSet.empty().ins(`e_n`). ... .ins(`e_0`).setName(''extname'')`

- /*{$e_0, ..., e_n|r$}*/''extname''

  la cui *traduzione* è la seguente:

  ```
  LSet r = new LSet(''r'');
  r.ins(e_n). ... .ins(e_0).setName(''extname'')
  ```

Sia gli elementi che compongono l'*insieme logico* che il suo nome esterno sono opzionali, può quindi trattarsi, rispettivamente, di un insieme *vuoto* o senza nome esterno.

3.1.1 Traduzione degli esempi del capitolo 2

Vediamo la traduzione delle istruzioni in linguaggio esteso del capitolo 2.1.1.

Liste

- `LList x = new LList(`/*$[1, 2, 3|r]$*/`''x'');`

  la cui *traduzione* è la seguente:

  ```
  LList r = new LList(''r'');
  LList x = new LList(r.ins(3).ins(2).ins(1).setName(''x''));
  ```

- `LList z = new LList(/*`$[4,5]$`*/''z'');`

  la cui *traduzione* è la seguente:

  ```
  LList z = new LList(LList.empty().ins(5).ins(4).setName(''z''));
  ```

- `LList y = new LList(/*`$[]$`*/''y'');`

  la cui *traduzione* è la seguente:

  ```
  LList y = new LList(LList.empty().setName(''y''));
  ```

- `solver.add(x.eq(/*`$['a','b','c|r2]$`*/''l1''));`

  la cui *traduzione* è la seguente:

  ```
  LList r2 = new LList(''r2'');
  solver.add(x.eq(r2.ins('c').ins('b').ins('a').setName(''l1'')));
  ```

- `solver.add(x.neq(/*`$['d',1,8]$`*/''l2''));`

  la cui *traduzione* è la seguente:

  ```
  solver.add(x.neq(LList.empty().ins(8).ins(1).ins('d').setName(''l2'')));
  ```

- `solver.add(x.in(/*`$[''io'',''tu'',''egli''|r3]$`*/));`

  la cui *traduzione* è la seguente:

  ```
  LList r3 = new LList(''r3'');
  solver.add(x.in(r3.ins(''egli'').ins(''tu'').ins(''io'')));
  ```

- `solver.add(x.nin(/*`$[1,2,''ciao'']$`*/''l3''));`

  la cui *traduzione* è la seguente:

  ```
  solver.add(x.nin(LList.empty().ins(''ciao'').ins(2).ins(1).setName(''l3'')));
  ```

Insiemi

- `LSet x = new LSet(/*{1, 2, 3|r}*/''x'');`

  la cui *traduzione* è la seguente:

  ```
  LSet r = new LSet(''r'');
  LSet x = new LSet(r.ins(3).ins(2).ins(1).setName(''x''));
  ```

- `LSet z = new LSet(/*{4, 5}*/''z'');`

  la cui *traduzione* è la seguente:

  ```
  LSet z = new LSet(LSet.empty().ins(5).ins(4).setName(''z''));
  ```

- `LSet y = new LSet(/*{}*/''y'');`

  la cui *traduzione* è la seguente:

  ```
  LSet y = new LSet(LSet.empty().setName(''y''));
  ```

- `solver.add(x.eq(/*{'a','b','c'|r2}*/''s1''));`

  la cui *traduzione* è la seguente:

  ```
  LSet r2 = new LSet(''r2'');
  solver.add(x.eq(r2.ins('c').ins('b').ins('a').setName(''s1'')));
  ```

- `solver.add(x.neq(/*{'d', 1, 8}*/''s2''));`

  la cui *traduzione* è la seguente:

  ```
  solver.add(x.neq(LSet.empty().ins(8).ins(1).ins('d').setName(''s2'')));
  ```

- `solver.add(x.in(/*{"io","tu","egli"|r3}*/''s3''));`

  la cui *traduzione* è la seguente:

```
LSet r3 = new LSet(''r3'');
solver.add(x.in(r3.ins(''egli'').ins(''tu'').ins(''io'').setName(''s3'')));
```

- `solver.add(x.nin(/*{1, 2,'' ciao''}*/));`

  la cui *traduzione* è la seguente:

  `solver.add(x.nin(LSet.empty().ins(''ciao'').ins(2).ins(1)));`

3.2 Nuovi vincoli

Vediamo la traduzione dei costrutti per la creazione di *nuovi vincoli definiti dall'utente*, definiti nel capitolo 2.2.

Definizione della classe che implementa i *nuovi vincoli definiti dall'utente*.

```
//%New Constraint Class
public class MyOps {
        private SolverClass solver;
        public MyOps(SolverClass s) {
                solver = s;
        }
        // metodi che implementano i nuovi vincoli definiti dall'utente
}
```

la cui *traduzione* è la seguente:

```
public class MyOps extends NewConstraintsClass {
        public MyOps(SolverClass currentSolver) {
                super(currentSolver);
        }
        // metodi che implementano i nuovi vincoli definiti dall'utente
}
```

Definizione dei metodi della classe `MyOps` che implementano i *nuovi vincoli definiti dall'utente*.

```
//%Constraint
public Constraint op1(t1 v1, t2 v2) {
        // implementazione vincolo op1
}
//%Constraint
public Constraint op2(t3 v3) {
        // implementazione vincolo op2
}
...
//%Constraint
public Constraint opn(t1 v4, t4 v5, t2 v6) {
        // implementazione vincolo opn
}
```

la cui *traduzione* avviene in tre fasi:

La prima fase crea i metodi che servono per aggiungere i *vincoli* `op1`, `op2`, ... ,`opn` al *constraint store*; crea quindi l'*interfaccia utente*.

```
public Constraint op1(t1 v1, t2 v2) {
        return new Constraint("op1", v1, v2);
}
public Constraint op2(t3 v3) {
        return new Constraint("op2", v3);
}
...
public Constraint opn(t1 v4, t4 v5, t2 v6) {
        return new Constraint("opn", v4, v5, v6);
```

La seconda fase crea lo `user_code`, che ha il compito di associare il *vincolo* richiamato dal *risolutore*, al metodo della classe `MyOps` che lo implementa; crea quindi l'*interfaccia solver*.

```
protected void user_code(Constraint c)
throw Failure, NotDefConstraintException {
        if (c.getName()=="op1")
                op1(c);
        else if (c.getName()=="op2")
                op2(c);
        ...
        else if (c.getName()=="opn")
                opn(c);
```

```
        else throw new NotDefConstraintException();
}
```

La terza e ultima fase genera i metodi veri e propri, di tipo `private void`, che implementano i nuovi *vincoli*, i quali avranno, come unico parametro, un oggetto di tipo `Constraint`.
L' implementazione di ogni metodo inizia con l'acquisizione dei suoi parametri, che vengono recuperati dal *vincolo* stesso tramite il metodo `getArg(int k)`, dove `k` è l'indice del parametro.

```
private void op1(Constraint c) throws Failure {
        t1 v1 = (t1)c.getArg(1);
        t2 v2 = (t2)c.getArg(2);

        // implementazione del vincolo op1
}
private void op2(Constraint c) throws Failure {
        t3 v3 = (t3)c.getArg(1);

        // implementazione del vincolo op2
}
...
private void opn(Constraint c) throws Failure {
        t1 v4 = (t1)c.getArg(1);
        t4 v5 = (t4)c.getArg(2);
        t2 v6 = (t2)c.getArg(3);

        // implementazione del vincolo opn
}
```

Infine, la traduzione degli oggetti `Constraint`, restituiti dai *nuovi vincoli definiti dall'utente*:

- `return Constraint.ok` viene tradotto in `return`
- `return Constraint.fail` viene tradotto in `c.fail()`
- `return Constraint.notSolved` viene tradotto in `c.notSolved()`

3.3 Costrutti per il non-determinismo

Vediamo la *traduzione* dei costrutti definiti nei capitoli 2.3 e 2.4.

3.3.1 either - orelse

```
//%either {stmt_1;}
//%orelse {stmt_2;}
...
//%orelse {stmt_n;}
```

la cui *traduzione* è la seguente:

```
switch(c.getAlternative()) {
        case 0: {
        Solver.addChoicePoint(c);
        stmt_1;
        break;
        }
        case 1: {
        Solver.addChoicePoint(c);
        stmt_2;
        break;
        }
        ...
        case n: {
        stmt_n;
        break;
        }
}
```

`Solver` è il *risolutore di vincoli* creato dal costruttore della classe, `c` è il *vincolo* che fa da parametro al metodo che utilizza questo costrutto, $stmt_1$, ..., $stmt_n$ sono sequenze di statement da eseguire *non-deterministicamente*.

Esempi di traduzione di questo costrutto, definito nel capitolo 2.3:

Il seguente vincolo definito dall'utente assegna *non-deterministicamente* ad una *variabile logica* gli interi `1,2,3`.

```
//%Constraint
public Constraint ndAssign(IntLVar x) {
...
        //%either
        {solver.add(x.eq(1));}
```

```
        //%orelse
        {solver.add(x.eq(2));}
        //%orelse
        {solver.add(x.eq(3));}
...
}
```

la cui *traduzione* è la seguente:

```
private void ndAssign(Constraint c) throws Failure {
...
switch(c.getAlternative()) {
        case 0: {
        Solver.addChoicePoint(c);
        Solver.add(x.eq(1));
        break;
        }
        case 1: {
        Solver.addChoicePoint(c);
        Solver.add(x.eq(2));
        break;
        }
        case 2: {
        Solver.add(x.eq(3));
        break;
        }
}
...
}
```

Il seguente vincolo definito dall'utente calcola *non-deterministicamente* il valore assoluto di un numero intero.

```
//%Constraint
public Constraint ndAbs(IntLVar x, IntLVar y) {
...
        //%either
        {solver.add(x.ge(0).and(x.eq(y)));}
        //%orelse
        {solver.add(x.lt(0).and(x.eq(new IntLVar(0).sub(y))));}
...
}
```

la cui *traduzione* è la seguente:

```
private void ndAbs(Constraint c) throws Failure {
...
switch(c.getAlternative()) {
        case 0: {
        Solver.addChoicePoint(c);
        Solver.add(x.ge(0).and(x.eq(y)));
        break;
        }
        case 1: {
        Solver.add(x.lt(0).and(x.eq(new IntLVar(0).sub(y))));
        break;
        }
}
...
}
```

3.3.2 findAll

```
//%findAll(c) { corpo }
```

la cui *traduzione* è la seguente:

```
SolverClass aux = new SolverClass();
aux.solve(c);
do {
   corpo
} while (aux.nextSolution());
```

dove `c` è il singolo *vincolo* da risolvere utilizzando un *risolutore* temporaneo, `corpo` è la sequenza di statement da eseguire per tutte le soluzioni di `c`.

```
//%findAll(solver) { corpo }
```

la cui *traduzione* è la seguente:

```
solver.solve();
do {
   corpo
} while (solver.nextSolution());
```

dove `solver` è il *risolutore di vincoli* utilizzato, `corpo` è la sequenza di statement da eseguire per tutte le soluzioni dei vincoli contenuti nel *constraint store*.

Esempi di traduzione di questo costrutto, definito nel capitolo 2.4:

Il seguente metodo calcola tutte le permutazioni degli elementi di un *insieme logico* `s` di interi.

```
public void allPermutations(LSet s) throws Failure {
        int n = s.getSize();
        LSet r = LSet.mkLSet(n);
        //%findAll(r.eq(s))
        {
                r.printElems(' ');
                System.out.println();
        }
}
```

la cui *traduzione* è la seguente:

```
public void allPermutations(LSet s) throws Failure {
        int n = s.getSize();
        LSet r = LSet.mkLSet(n);
        SolverClass aux = new SolverClass();
        aux.solve(r.eq(s));
        do {
                r.printElems(' ');
                System.out.println();
        } while (aux.nextSolution());
}
```

Il seguente metodo stampa in output le soluzioni *non-deterministiche* generate dal `vincolo concat`, grazie al *risolutore di vincoli* `solver` creato dall'utente.

```
public void allSolutions(LList lin) throws Failure {
        LList lout1 = new LList();
        LList lout2 = new LList();
        solver.add(concat(lout1,lout2,lin));
        //%findAll(solver)
        {
                lout1.output();
                lout2.output();
        }
}
```

la cui *traduzione* è la seguente:

```
public void allSolutions(LList lin) throws Failure {
        LList lout1 = new LList();
        LList lout2 = new LList();
        solver.add(concat(lout1,lout2,lin));
        solver.solve();
        do {
                lout1.output();
                lout2.output();
        } while (solver.nextSolution());
}
```

3.4 Esempi di programmi completi

Vengono qui mostrati alcuni esempi di programmi completi con relative classi main, sviluppati in linguaggio Java esteso, e la loro *traduzione* effettuata dal *preprocessore*.
Possiamo notare il risparmio di codice e la relativa semplicità d'implementazione, grazie all'utilizzo delle *direttive per il preprocessore*.

Esempio 1: Concatenazione di liste logiche

Implementazione del *vincolo* per effettuare la concatenazione degli elementi di due *liste logiche*, e la stampa di ogni possibile soluzione di questa operazione.

```
import JSetL.*;
```

```
//%New Constraint Class
public class Ops {
        private SolverClass solver;
        public Ops(SolverClass s) {
                solver = s;
        }
        //%Constraint
        public Constraint concat(LList l1, LList l2, LList l3) {
                //%either
                {
                        solver.add(l1.eq(/*[]*/));
                        solver.add(l2.eq(l3));
                }
                //%orelse
                {
                        LVar x = new LVar();
                        solver.add(l1.eq(/*[x|l1new]*/));
                        solver.add(l3.eq(/*[x|l3new]*/));
                        //%solver.add(concat(l1new,l2,l3new));
                }
                return Constraint.ok;
        }
        public void allSolutions(LList lin)
        throws Failure {
                LList lout1 = new LList();
                LList lout2 = new LList();
                solver.add(concat(lout1,lout2,lin));
                //%findAll(solver)
                {
                        lout1.output();
                        lout2.output();
                }
        }

}
```

TRADUZIONE DEL PREPROCESSORE

```
package processati;
import JSetL.*;

public class Ops extends NewConstraintsClass {
        public Ops (SolverClass currentSolver) {
        super(currentSolver);
        }
```

```
public Constraint concat(LList l1, LList l2, LList l3) {
        return new Constraint("concat", l1, l2, l3);
}

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

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);


        switch(c.getAlternative()) {

        case 0: {

        Solver.addChoicePoint(c);
        Solver.add(l1.eq(LList.empty()));
        Solver.add(l2.eq(l3));

        break;
        }
        case 1: {

        LVar x = new LVar();
        LList l1new = new LList();
        LList l3new = new LList();
        Solver.add(l1.eq(l1new.ins(x)));
        Solver.add(l3.eq(l3new.ins(x)));
        Solver.add(concat(l1new,l2,l3new));

        break;
        }

        }

        return;

}

public void allSolutions(LList lin)
throws Failure {
        LList lout1 = new LList();
        LList lout2 = new LList();
        solver.add(concat(lout1,lout2,lin));
```

```
        solver.solve();
        do {
                lout1.output();
                lout2.output();
        } while (solver.nextSolution());
}

}
```

Implementazione del metodo main, per effettuare alcune prove.

```
import JSetL.*;

public class Main_di_Ops {
        private static SolverClass s = new SolverClass();
        public static void main (String[] args)
        throws NotDefConstraintException, Failure {
            LList l1 = new LList(/*["pippo","pluto"]*/);
            LList l2 = new LList(/*["paperino","topolino"]*/);
            LList l = new LList("l");
            LList l3 = new LList("l3");
            LList l4 = new LList("l4");
            Ops O = new Ops(s);
            s.check(O.concat(l1, l2, l));
            O.allSolutions(l);
        }
}
```

TRADUZIONE DEL PREPROCESSORE

```
package processati;
import JSetL.*;
public class Main_di_Ops_main {
    private static SolverClass s = new SolverClass();
    public static void main (String[] args)
    throws NotDefConstraintException, Failure {
        LList l1 = LList.empty().ins("pluto").ins("pippo");
        LList l2 = LList.empty().ins("topolino").ins("paperino");
        LList l = new LList("l");
        LList l3 = new LList("l3");
```

```
        LList l4 = new LList("l4");
        Ops O = new Ops(s);
        s.check(O.concat(l1, l2, l));
        O.allSolutions(l);
    }
}
```

Esempio 2: Parser per espressioni matematiche

Implementazione dei *vincoli* per effettuare il controllo di correttezza sintattica di un'espressione matematica.

```
import JSetL.*;

//%New Constraint Class
public class ExprParser {
        private SolverClass solver;
        public ExprParser(SolverClass s) {
                solver = s;
        }
        //%Constraint
        public Constraint expr (LList L, LList Remain) {
                //%either
                {// expr(L, Remain) :- num(L, Remain)
                        solver.add(num(L, Remain));
                }
                //%orelse
                {// expr(L, Remain) :-
                 // num(L, L1), L1 = [+|L2], expr(L2, Remain)
                        LList L1 = new LList();
                        solver.add(num(L,L1));
                        solver.add(L1.eq(/*['+'|L2]*/));
                        //%solver.add(expr(L2, Remain));
                }
                //%orelse
                {// expr(L, Remain) :-
                 // num(L, L1), L1 = [-|L2], expr(L2, Remain)
                        LList L1 = new LList();
                        solver.add(num(L,L1));
                        solver.add(L1.eq(/*['-'|L2]*/));
                        //%solver.add(expr(L2, Remain));
                }
```

```
                return Constraint.ok;
        }
        //%Constraint
        public Constraint num(LList L, LList Remain) {
                LVar D = new LVar();
                solver.add(L.eq(Remain.ins(D)));
                solver.add(number(D));
                return Constraint.ok;
        }
        //%Constraint
        public Constraint number(LVar n) {
                if (n.getValue() instanceof Integer)
                        return Constraint.ok;
                else
                        return Constraint.fail;
        }
}
```

TRADUZIONE DEL PREPROCESSORE

```
package processati;
import JSetL.*;

public class ExprParser extends NewConstraintsClass {
        public ExprParser (SolverClass currentSolver) {
        super(currentSolver);
        }

public Constraint expr(LList L, LList Remain) {
        return new Constraint("expr", L, Remain);
}

public Constraint num(LList L, LList Remain) {
        return new Constraint("num", L, Remain);
}

public Constraint number(LVar n) {
        return new Constraint("number", n);
}

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

```
}

private void expr(Constraint c) throws Failure {
        LList L = (LList)c.getArg(1);
        LList Remain = (LList)c.getArg(2);


        switch(c.getAlternative()) {

        case 0: {

        Solver.addChoicePoint(c);
        // expr(L, Remain) :- num(L, Remain)
        Solver.add(num(L, Remain));

        break;
        }
        case 1: {

        Solver.addChoicePoint(c);
        // expr(L, Remain) :-
        // num(L, L1), L1 = [+|L2], expr(L2, Remain)
        LList L1 = new LList(), L2 = new LList();
        Solver.add(num(L,L1));
        Solver.add(L1.eq(L2.ins('+')));
        Solver.add(expr(L2, Remain));

        break;
        }
        case 2: {

        // expr(L, Remain) :-
        // num(L, L1), L1 = [-|L2], expr(L2, Remain)
        LList L1 = new LList(), L2 = new LList();
        Solver.add(num(L,L1));
        Solver.add(L1.eq(L2.ins('-')));
        Solver.add(expr(L2, Remain));

        break;
        }

        }

        return;

}

private void num(Constraint c) throws Failure {
        LList L = (LList)c.getArg(1);
```

```
        LList Remain = (LList)c.getArg(2);

        LVar D = new LVar();
        Solver.add(L.eq(Remain.ins(D)));
        Solver.add(number(D));
        return;

}

private void number(Constraint c) throws Failure {
        LVar n = (LVar)c.getArg(1);

        if (n.getValue() instanceof Integer)
            return;
        else
            c.fail();

}

}
```

Implementazione del metodo main, per effettuare alcune prove.

```
import JSetL.*;

public class Main_di_ExprParser {
    private static SolverClass solver = new SolverClass();
    public static void main(String[] args) throws Failure {
       ExprParser sampleParser = new ExprParser(solver);
       // l0 = [3] true
       LList l0 = new LList(/*[3]*/ "l0");
       l0.output();
       System.out.println(solver.check(sampleParser.expr(
       l0,LList.empty())));
       // l1 = [5,+,3] true
       System.out.println();
       solver.clearStore();
       LList l1 = new LList(/*[5,'+',3]*/ "l1");
       l1.output();
       System.out.println(solver.check(sampleParser.expr(
       l1,LList.empty())));
       // l2 = [5,+,3,-,2] true
       System.out.println();
       solver.clearStore();
```

```
        LList l2 = new LList(/*[5,'+',3,'-',2]*/ "l2");
        l2.output();
        System.out.println(solver.check(sampleParser.expr(
        l2,LList.empty())));
        // l3 = [5,+,3,-] false
        System.out.println();
        solver.clearStore();
        LList l3 = new LList(/*[5,'+',3,'-']*/ "l3");
        l3.output();
        System.out.println(solver.check(sampleParser.expr(
        l3,LList.empty())));
        // l4 = [-] false
        System.out.println();
        solver.clearStore();
        LList l4 = new LList(/*['-']*/ "l4");
        l4.output();
        LList R4 = new LList("R4");
        System.out.println(solver.check(sampleParser.expr(l4,R4)));
    }
}
```

TRADUZIONE DEL PREPROCESSORE

```
package processati;
import JSetL.*;
public class Main_di_ExprParser_main {
    private static SolverClass solver = new SolverClass();
    public static void main(String[] args) throws Failure {
        ExprParser sampleParser = new ExprParser(solver);
        // l0 = [3] true
        LList l0 = LList.empty().ins(3).setName("l0");
        l0.output();
        System.out.println(solver.check(sampleParser.expr(
        l0,LList.empty())));
        // l1 = [5,+,3] true
        System.out.println();
        solver.clearStore();
        LList l1 = LList.empty().ins(3).ins('+').ins(5).setName(
        "l1");
        l1.output();
        System.out.println(solver.check(sampleParser.expr(
        l1,LList.empty())));
        // l2 = [5,+,3,-,2] true
        System.out.println();
        solver.clearStore();
        LList l2 = LList.empty().ins(2).ins('-').ins(3).ins(
```

```
        '+').ins(5).setName("l2");
        l2.output();
        System.out.println(solver.check(sampleParser.expr(
        l2,LList.empty())));
        // l3 = [5,+,3,-] false
        System.out.println();
        solver.clearStore();
        LList l3 = LList.empty().ins('-').ins(3).ins(
        '+').ins(5).setName("l3");
        l3.output();
        System.out.println(solver.check(sampleParser.expr(
        l3,LList.empty())));
        // l4 = [-] false
        System.out.println();
        solver.clearStore();
        LList l4 = LList.empty().ins('-').setName("l4");
        l4.output();
        LList R4 = new LList("R4");
        System.out.println(solver.check(sampleParser.expr(l4,R4)));
    }

}
```

Capitolo 4

Implementazione del preprocessore JSetL

In questo capitolo analizzeremo il funzionamento del *preprocessore*, sia l'architettura che lo caratterizza, compresi input e output del programma, che l'implementazione in dettaglio di ogni modulo di cui è composto.

4.1 Architettura del preprocessore

L'architettura del *preprocessore* include le seguenti componenti:

- **programma del preprocessore**: Il programma del *preprocessore* è costituito da 3 classi, contenute nel *default package*:
  1. classe **Preprocessore**: Acquisisce in input i file da tradurre e chiama i metodi della classe *Servizi*, in base al tipo di classe contenuta nel file attualmente in traduzione.
  2. classe **Servizi**: Contiene i metodi necessari all'acquisizione dal file in input, traduzione e scrittura in output delle singole istruzioni o di interi costrutti appartenenti al *linguaggio esteso*.
  3. classe **PreprocEcc**: Implementa la gestione delle eccezioni.
- **programma utente da tradurre**: Il programma utente da tradurre, input del *preprocessore*, è costituito, in generale, da più classi contenute nel *default package*.
- **programma utente tradotto**: Il programma utente tradotto, output del preprocessore, sarà costituito dalle stesse classi del **programma utente da tradurre** ma poste nel *package processati*.

- **libreria JSetL**: La libreria JSetL, contenuta nel *package JSetL*.
- **script build.xml**: Lo script *XML* per l'automatizzazione del processo di traduzione.

All'avvio da parte dell'utente dello script *build.xml*, il metodo `main` della classe *Preprocessore*, dopo aver preso in input i file Java da tradurre, elabora i dati in essi contenuti grazie alle funzionalità offerte dalla classe *Servizi* e scrive i risultati nei file di output corrispondenti, creati nel *package processati*.

Vediamo le classi *Preprocessore* e *Servizi* più in dettaglio:

4.2 Classe Preprocessore

La classe *Preprocessore* ha il compito di acquisire in input i file da tradurre e chiamare i metodi della classe *Servizi* per l'esecuzione della traduzione e la scrittura in output.

L'acquisizione dei file da tradurre inizia attraverso la lettura in input della stringa contenente i path dei file separati da '$', i quali vengono suddivisi in tokens grazie al metodo `leggi_tokens`; quindi, un token alla volta, viene isolato il file corrispondente e ne viene acquisito il codice in esso contenuto, grazie al metodo `leggi_file`, per la successiva traduzione, che verrà eseguita dai metodi della classe *Servizi*.

Per iniziare l'analisi del file isolato viene, per prima cosa, ricavato il nome della classe in esso contenuta, quindi si procede al rilevamento della categoria a cui appartiene, operazione necessaria per poter tradurre la classe nel modo appropriato.
Le categorie possibili sono: *classi contenenti vincoli definiti dall'utente*, *classi contenenti il metodo main*, *classi ausiliarie*.
Per quanto riguarda la traduzione delle classi appartenenti alla prima categoria (le *classi contenenti vincoli definiti dall'utente*, riconosciute grazie alla presenza della direttiva `//%New Constraint Class`) viene inizialmente ricavato il nome del *risolutore di vincoli* in uso nella classe, quindi viene creato il file, nella cartella *processati*, che ha come nome lo stesso della classe da tradurre e che conterrà la classe tradotta.
Si procede, quindi, all'acquisizione dei dati riguardanti i *vincoli*, definiti sottoforma di metodi della classe in traduzione, attraverso i metodi della classe *Servizi*, `acquisisci_nome_funzione`, `acquisisci_corpo_funzione`

e `acquisisci_parametri`, chiamati per ogni *nuovo vincolo* della classe; una volta terminata l'acquisizione dei dati, si può passare alla scrittura dei dati iniziali della classe tradotta con `scrivi_parametri_constraint` e `scrivi_user_code`. Per ogni *nuovo vincolo* rilevato, vengono quindi chiamati `scrivi_getArg` e `scrivi_corpo_funzione`, che procederanno alla effettiva traduzione, attraverso il metodo `jsetlwork`, e alla scrittura sul file di output, dell'intestazione e del corpo dei vincoli tradotti.
Per effettuare la traduzione delle altre due categorie di classi, *contenenti il metodo main* e *ausiliarie*, viene creato il file di output nella cartella *processati* come di consueto e, per effettuare la traduzione e la scrittura sul file corrispondente in output, viene semplicemente chiamato il metodo `jsetlwork` sull'intera stringa in input.

Sebbene la modalità principale del preprocessore è la traduzione di interi programmi Java, esiste la possibilità della traduzione di un singolo file Java attraverso l'uso di un oggetto `JFileChooser`, evitando così l'uso dello script *build.xml*; questa modalità è stata implementata principalmente per favorire il debug, da parte dell'utente, durante l'implementazione di un programma Java in *linguaggio esteso*.

4.3 Classe Servizi

Questa classe contiene i metodi di cui il programma *preprocessore* si serve per effettuare la traduzione.
Questi metodi possono essere suddivisi in due categorie: metodi per l'acquisizione dati dai file di input e metodi per l'elaborazione dati e successiva scrittura dati sui file di output.

4.3.1 Acquisizione dati

acquisisci_nome_funzione

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, per prima cosa acquisisce il nome del *vincolo* da tradurre, effettua un controllo sulla presenza della direttiva `//%Constraint` che sta ad indicare la definizione di un *vincolo definito dall'utente*, e infine, prepara gli indici per la successiva acquisizione del corpo dello stesso.

Questa funzione prende in input la stringa del file in traduzione, l'array contenente i nomi di tutti i *vincoli definiti dall'utente* finora acquisiti e il nome della classe da tradurre, e restituisce in output il nome del *vincolo* acquisito.

acquisisci_corpo_funzione

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, per prima cosa controlla la presenza del costrutto `either - orelse`, in tal caso viene acquisito il codice precedente, postcedente il costrutto e gli statements che lo compongono, che saranno eseguiti *non-deterministicamente*.
Se il costrutto `either - orelse` non viene rilevato, viene semplicemente acquisito indistintamente tutto il corpo della funzione.
Questa funzione prende in input solamente la stringa del file in traduzione e restituisce i dati del corpo del *vincolo* in traduzione.

acquisisci_parametri

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, acquisisce i parametri del *vincolo* in traduzione, sia il loro tipo che il loro nome.
Questa funzione prende in input solamente la stringa del file in traduzione e restituisce nome e tipo dei parametri del *vincolo* in traduzione.

leggi_file

Questa funzione acquisisce, un carattere alla volta, il file in traduzione, attraverso uno stream di input su di esso.
Questa funzione prende in input il file da tradurre e restituisce una stringa con il contenuto del file.

leggi_tokens

Questa funzione crea un oggetto `StringTokenizer`, che viene riempito con tanti *token* quanti sono i file in input da tradurre presenti in `args[0]`, sono quindi passati ad un array di stringhe.
Questa funzione prende in input la stringa contenente i file da tradurre divisi da '$' e restituisce l'array di tokens.

isola_comando

Questa funzione acquisisce dalla stringa in input la prossima istruzione, dalla fine di quella precedente al prossimo ';' e, nel caso in cui in traduzione ci sia una *classe che implementi vincoli definiti dall'utente*, permette l'utilizzo di un *risolutore di vincoli* con un nome che non sia di default.
Questa funzione prende in input la stringa del file in traduzione, lo stream ad essa associato, il *risolutore di vincoli* utilizzato nella classe, informazione sfruttata soltanto nel caso di *classi che implementano vincoli definiti dall'utente*, e restituisce l'istruzione acquisita.

crea_file_output

Questa funzione crea un file nella cartella *processati*, con lo stesso nome del file in input e ne viene creato lo stream di output per la scrittura.
Questa funzione prende in input il file in traduzione e il nome della classe in essa contenuto, e restituisce lo stream di output del file tradotto.

4.3.2 Elaborazione e Scrittura dati

scrivi_parametri_constraint

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, scrive la parte iniziale della traduzione nel file in output, import dei pacchetti, definizione della classe, costruttore che inizializza il *risolutore di vincoli* e le intestazioni dei *vincoli*, i quali ritornano un oggetto `Constraint`, con relativi tipo e nome dei parametri.
Questa funzione prende in input lo stream di output della traduzione, il nome della classe in traduzione e gli array che contengono i dati dei *vincoli* di quella classe.

scrivi_user_code

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, scrive il metodo `user_code`, che serve per chiamare il metodo associato al nome del *vincolo* nel *constraint store*, agisce quindi come router dei metodi.
Questa funzione prende in input lo stream di output della traduzione e i nomi dei *vincoli* acquisiti.

scrivi_getArg

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, acquisito il nome dei *vincoli* da tradurre e tipo e nome dei loro parametri grazie ai metodi `acquisisci_nome_funzione` e `acquisisci_parametri`, scrive questi dati nello stream di output della traduzione.
Questa funzione prende in input lo stream di output della traduzione e tutti i dati concernenti le definizioni dei *vincoli*, il loro nome, e tipo e nome dei loro parametri.

scrivi_corpo_funzione

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, acquisito il corpo dei *vincoli* da tradurre grazie al metodo `acquisisci_corpo_funzione`, traduce in modo opportuno questi dati grazie al metodo `jsetlwork` e successivamente li scrive nello stream di output della traduzione.
Questa funzione prende in input lo stream di output della traduzione, il *risolutore di vincoli* utilizzato nella classe, informazione sfruttata soltanto nel caso di classi che implementano *vincoli definiti dall'utente*, e tutti i dati concernenti il corpo dei *vincoli* in traduzione.

jsetlwork

Questa funzione, una volta isolata la prossima istruzione grazie a `isola_comando`, chiama la giusta funzione per la sua traduzione, quindi viene isolata la prossima istruzione e così via fino al termine della stringa del file in traduzione.
Questa funzione prende in input la stringa del file in traduzione, il *risolutore di vincoli* utilizzato nella classe, informazione sfruttata soltanto nel caso di traduzione di *classi che implementano vincoli definiti dall'utente*, lo stream di output e un parametro di sistema.

nomeclasse

Questa funzione traduce la definizione della classe contenuta nel file in traduzione e la scrive in output.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita e lo stream di output.

nomemetodo

Questa funzione traduce le definizioni dei metodi contenuti nel file in traduzione e le scrive in output.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita e lo stream di output.

comando_semplice

Questa funzione si limita a scrivere sullo stream di output l'istruzione acquisita, non essendo stata ritenuta necessaria nessuna traduzione.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita e lo stream di output.

comando_return

Questa funzione, chiamata dal *preprocessore* soltanto in caso di traduzione di *classi contenenti vincoli definiti dall'utente*, traduce le istruzioni `return` dei metodi della classe contenuta nel file in traduzione.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita e lo stream di output.

comando_findAll

Questa funzione traduce la direttiva `//%findAll` acquisendo per prima cosa i suoi parametri: il *vincolo* da risolvere e il *risolutore*, utilizzato soltanto nel caso in cui stessimo traducendo una *classe che implementi vincoli definiti dall'utente*.
Viene, infine, acquisito il corpo del costrutto e scritta la sua traduzione sullo stream di output, in base ai parametri acquisiti.
Questa funzione prende in input la stringa del file in traduzione, lo stream di output e il *risolutore di vincoli* utilizzato.

inverti_parametri

Questa funzione ausiliaria inverte l'ordine dei parametri che compongono la definizione di una nuova *lista logica* o nuovo *insieme logico*, per poter effettuare il loro inserimento nell'oggetto nel giusto ordine.
Questa funzione prende in input l'array dei parametri e restituisce lo stesso array, con gli elementi invertiti di posizione.

comando_lista

Questa funzione traduce le istruzioni che creano *liste logiche non vuote*.
Per prima cosa acquisisce l'intera sottostringa da tradurre, successivamente acquisisce i suoi elementi, a cui vengono applicati la `inverti_parametri`, e infine vengono acquisiti, se sono presenti, *resto* e *nome esterno*; viene quindi scritto, nello stream di output, la traduzione dell'istruzione in base ai parametri specificati.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita, lo stream di output e un parametro di sistema.

comando_insieme

Questa funzione traduce le istruzioni che creano *insiemi logici non vuoti*.
Per prima cosa acquisisce l'intera sottostringa da tradurre, successivamente acquisisce i suoi elementi, a cui vengono applicati la `inverti_parametri`, e infine vengono acquisiti, se sono presenti, *resto* e *nome esterno*; viene quindi scritto, nello stream di output, la traduzione dell'istruzione in base ai parametri specificati.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita, lo stream di output e un parametro di sistema.

comando_listavuota

Questa funzione traduce le istruzioni che creano *liste logiche vuote*.
Per prima cosa acquisisce l'intera sottostringa da tradurre, infine viene acquisito, se presente, il *nome esterno*; viene quindi scritto, nello stream di output, la traduzione dell'istruzione in base ai parametri specificati.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita, lo stream di output e un parametro di sistema.

comando_insiemevuoto

Questa funzione traduce le istruzioni che creano *insiemi logici vuoti*.
Per prima cosa acquisisce l'intera sottostringa da tradurre, infine viene acquisito, se presente, il *nome esterno*; viene quindi scritto, nello stream di output, la traduzione dell'istruzione in base ai parametri specificati.
Questa funzione prende in input la stringa del file in traduzione, l'istruzione acquisita, lo stream di output e un parametro di sistema.

Capitolo 5

Automatizzazione della traduzione con Apache ANT

L'utilizzo di un software per l'automatizzazione del processo di build per il preprocessore è di centrale importanza, poichè fornisce gli strumenti per automatizzare l'acquisizione dei file in input, l'esecuzione della traduzione e la scrittura dei file in output, operazioni che andrebbero altrimenti eseguite manualmente, con spreco di tempo e risorse, sia per l'utente che per il sistema.
In questo capitolo introdurremo il software Apache ANT, utilizzato nell'automatizzazione del processo di build, quindi analizzeremo lo script ANT che utilizziamo per automatizzare l'utilizzo del preprocessore.
Infine mostreremo un esempio completo di traduzione automatica.

L'automatizzazione del processo di build interessa, in generale, un'ampia varietà di operazioni:

- Compilazione del codice sorgente
- Creazione e cancellazione di files e cartelle
- Esecuzione di file *class*
- Creazione di insiemi di files
- Passaggio di parametri a file *class*

I vantaggi dell'automatizzazione del processo di build sono molteplici:

- Aumento della qualità del prodotto software
- Accelerazione del processo di compilazione e link

- Eliminazione di task ridondanti
- Minimizzazione di inutili processi di build

5.1 Apache ANT

Apache ANT [7] è un software per l'automatizzazione del processo di *build*, ed è un progetto open source scritto in Java.
Una importante caratteristica di *Apache ANT* è la portabilità.
Ad esempio, una differenza comune tra le varie piattaforme è il modo in cui è specificato il path delle directory: *UNIX* usa il forward slash (/) per delimitare i componenti di un path, mentre *WINDOWS* usa il backslash (\).
La sintassi *ANT* lascia libero il programmatore di usare una convenzione qualsiasi e converte successivamente il tutto nella forma più appropriata alla piattaforma in uso.
Il funzionamento di *Apache ANT* si basa sullo script *build.xml*, che si trova all'interno della cartella del progetto, allo stesso livello della cartella *src*.

Ogni build file è composto da `target` in cui sono elencati i `task`, le istruzioni da eseguire. Nel progetto possono essere definite delle `properties`, coppie $< nome, valore >$ immodificabili nel resto del progetto.
I `target` possono dipendere da altri `target`, i quali devono essere eseguiti prima di quello corrente; il `target` di default è il primo ad essere eseguito, purchè non dipenda da altri.
La struttura di uno script ANT è la seguente:

```
<?xml version="1.0" encoding="UTF-8"?>

<project default="target_n" name="nome_progetto">

        <target name="target_n" depends="target_k">
                ...
        </target>

        .
        .
        .

        <target name="target_3" depends="target_2">
                ...
        </target>
```

```
        <target name="target_2" depends="target_1">
                ...
        </target>

        <target name="target_1">
                ...
        </target>

</project>
```

Quando in un programma Java è presente uno script ANT, il principale metodo di build è l'esecuzione di questo script tramite l'opzione Run As Ant Build.

5.2 Build file per il preprocessore JSetL

Lo script build.xml, di nome Preprocessore, che utilizziamo nel nostro programma, è composto da quattro `target`, ognuno dei quali presenta il proprio nome e il `target` da cui dipende.
Secondo la gerarchia specificata dallo script, l'ordine di esecuzione dei `target` è il seguente: *Init* → *Compila* → *EseguiPrc* → *Esegui*.

Sono qui descritte le funzionalità di ogni *target* dello script *build.xml*.

```
<target name="Init">
        <delete dir="src\processati"/>
        <mkdir dir="src\processati"/>
</target>
```

Questo *target*, di nome `Init`, cancella la cartella *processati* con tutti i file al suo interno, prodotto di eventuali traduzioni precedenti, e crea una cartella *processati* vuota.

```
<target name="Compila" depends="Init">
    <javac encoding="ISO-8859-1" includeantruntime="false"
     srcdir="src" destdir="src\processati">
        <include name="Preprocessore.java"/>
    </javac>
</target>
```

Questo *target*, di nome `Compila`, compila i file di sistema *Preprocessore.java*, *Servizi.java*, *PreprocEcc.java* contenuti in *src*, tramite il *task* `javac`, e mette i file *class* risultanti nella cartella *processati*.

```
<target name="EseguiPrc" depends="Compila">
        <fileset dir="src" id="src.files">
                <include name="*.java"/>
                <exclude name="Preprocessore.java"/>
                <exclude name="Servizi.java"/>
                <exclude name="PreprocEcc.java"/>
        </fileset>
        <pathconvert pathsep="$" property="javafiles"
        refid="src.files"/>
        <java classname="Preprocessore"
        classpath="src\processati" >
                <arg value="${javafiles}"/>
        </java>
</target>
```

Questo *target*, di nome `EseguiPrc`, per prima cosa crea un `fileset` Java, che comprende tutti i files che si trovano in *src* tranne quelli di sistema, al quale viene dato il nome di `src.files`; comprende quindi tutti i file che dovranno essere presi in input dal *preprocessore* per essere tradotti. Viene poi creata una stringa contenente i path dei file contenuti in `src.files`, separati dal carattere ''$'', il cui nome è `javafiles`.
Infine, viene eseguito il metodo `main` della classe `Preprocessore`, il cui file è stato precedentemente compilato nel *target* `Compila`, a cui viene dato in input la stringa `javafiles` attraverso il *task* `arg`; ciò genera, in `src\processati`, i file Java contenenti le classi tradotte dal *preprocessore*.

```
<target name="Esegui" depends="EseguiPrc">
        <javac sourcepath="src" encoding="ISO-8859-1"
        includeantruntime="false" srcdir="src\processati"
        destdir="src" listfiles="true">
                <include name="*.java"/>
        </javac>
        <fileset dir="src\processati" id="cmp.files">
                <include name="*_main.class"/>
        </fileset>
        <pathconvert pathsep=" " property="classfiles"
        refid="cmp.files">
                <chainedmapper>
                    <flattenmapper/>
                    <globmapper from="*.class" to="*"/>
                </chainedmapper>
```

```
        </pathconvert>
        <java classname="processati.${classfiles}"
        classpath="src" failonerror="true"/>
        <fileset dir="src\JSetL" id="cls.files">
                <include name="*.class"/>
        </fileset>
        <delete>
                <fileset refid="cls.files"/>
        </delete>
</target>
```

Questo *target*, di nome `Esegui`, per prima cosa compila tutti i file Java contenuti in src\processati, cioè gli output del processo di traduzione eseguito nel *target* `EseguiPrc`, e mette i file compilati nella stessa cartella.
Viene quindi isolata, tra i prodotti di `javac`, la singola classe che contiene il metodo *main*, cioè l'unica classe eseguibile del progetto tradotto, a cui è stato convenientemente aggiunto "_main" al nome; il file acquisito prende il nome di `cmp.files`. Viene quindi creata una stringa contenente il path del file class isolato dal *task* precedente.
Viene cambiata questa stringa, e quindi il path che contiene, modificando "*.class" in "*", riuscendo così ad avere l'esatto nome della classe *main*, dentro alla *property* `classfiles`.
Infine, viene eseguito il metodo *main* del progetto, ubicato all'interno della classe isolata nella *property* `classfiles`, chiamando il *task* `java` sulla stessa, eliminando poi i prodotti della compilazione di `JSetL`.
Questo *target*, avendo compiti di esecuzione, può essere terminato forzatamente dallo script in caso di errori a runtime.

5.3 Esempio completo

Nel seguente esempio, la situazione iniziale, prima che l'utente esegua lo script *build.xml*, è la seguente:

- **programma del preprocessore**: Il programma del *preprocessore* è costituito dalle classi Preprocessore, Servizi, PreprocEcc, contenute nel *default package*.
- **programma utente da tradurre**: Il programma utente da tradurre, input del *preprocessore*, è costituito dalle classi ExprParser, MyOps, Main_di_MyOps, Aux, contenute nel *default package*.
- **programma utente tradotto**: Il programma utente tradotto, output del preprocessore, è inesistente.

- **libreria JSetL**: La libreria JSetL, contenuta nel *package JSetL*.
- **script build.xml**: Lo script *XML* per l'automatizzazione del processo di traduzione.

Al momento dell'esecuzione, da parte dell'utente, dello script *build.xml*, viene creato il *package processati* grazie al `target Init`, che conterrà il **programma utente tradotto**.

Viene compilato il **programma del preprocessore**, grazie al `target Compila`.

Nel `target EseguiPrc` viene eseguito il metodo `main` della classe Preprocessore, compilata precedentemente, a cui viene passato, attraverso il suo parametro args[0], la seguente stringa:
"filepath ExprParser**$**filepath MyOps**$**filepath Main_di_MyOps**$**filepath Aux"
dove filepath indica il path completo del file all'interno del file system, contenente la classe specificata.
Viene quindi passato, al metodo *main* della classe Preprocessore, la stringa contenente i filepath delle classi del **programma utente da tradurre**, separati dal carattere "**$**".

A questo punto il **programma utente tradotto**, inizialmente inesistente, è stato creato e si trova nel *package processati*.
Esso contiene le seguenti classi: ExprParser, MyOps, Main_di_MyOps_main, Aux.

Nel `target Esegui`, infine, saranno compilate le classi del **programma utente tradotto** ed eseguito il metodo `main` della classe Main_di_MyOps_main, essendo la classe *main* del programma tradotto.

Capitolo 6

Conclusioni

Il lavoro svolto è consistito nell'implementazione di un *preprocessore* per il linguaggio Java, specificatamente progettato per facilitare lo sviluppo di programmi che utilizzano la libreria `JSetL`.
E' stata scelta una implementazione ad-hoc poichè, sebbene esistano strumenti simili [6], si tratta principalmente di semplici preprocessori lessicali di tipo *general purpose* e, come tali, difficilmente adattabili ad una versione di *preprocessore* che realizzi le funzionalità che vogliamo implementare.
Per queste motivazioni si è preferito sviluppare un *preprocessore* Java ad-hoc di tipo *special purpose*.

Per quanto riguarda gli sviluppi futuri, uno di essi potrebbe essere la realizzazione di una versione di *preprocessore* che utilizzi una tecnica più avanzata per la traduzione del codice Java esteso e riesca a creare le classi aggiuntive, come quelle necessarie alla definizione di nuovi vincoli, in modo automatico a partire dalle direttive `JSetL` poste nelle classi principali.
Un ulteriore sviluppo prevede di rendere più completo il traduttore delle costanti collezione, permettendo anche collezioni logiche annidate, ed estendere l'utilizzo del *preprocessore* alle espressioni aritmetiche presenti nei vincoli `JSetL`.
Quest'ultime possibilità richiederebbero però l'utilizzo di una tecnica di traduzione più sofisticata all'interno del *preprocessore*.
Un altro significativo sviluppo potrebbe essere l'implementazione, nel *preprocessore*, di ulteriori costrutti per lo sfruttamento del *non-determinismo* o per il calcolo delle soluzioni dei *vincoli* nel *constraint store*.

Bibliografia

[1] Gianfranco Rossi, Federico Bergenti
Nondeterministic Programming in Java with JSetL
Fundamenta Informaticae XXI 2001, IOS Press.

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

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

[6] **Java Comment Preprocessor**
`http://code.google.com/p/java-comment-preprocessor`

[7] **Apache ANT**
`http://ant.apache.org/manual/index.html`

Ringraziamenti

Mi sembra doveroso dedicare una pagina di questo lavoro alle persone che hanno contribuito al raggiungimento della mia laurea.

Un primo ringraziamento va alla mia famiglia, per avermi permesso di portare a termine il corso di studi senza mai farmi mancare nulla.

Un sentito ringraziamento al Prof. Gianfranco Rossi, per avermi seguito durante il periodo di tirocinio e stesura della tesi, e per aver dedicato così tanto tempo al mio lavoro nonostante i mille impegni quotidiani.

Desidero inoltre ringraziare tutti i miei compagni di corso, con i quali ho condiviso un lungo cammino fatto di momenti divertenti e momenti meno felici, ma grazie ai quali ho trovato sempre la forza di continuare il percorso intrapreso.
Un grazie speciale va a Ravo,Rasti,Mirko,Gianni,Fede,Daniel,Cavo,Long e a tutti gli amici conosciuti durante questo periodo, per essere stati presenti quando ne avevo bisogno e per avermi permesso di condividere con voi quest'esperienza.

A tutte queste persone un grazie di cuore.