

Tokenizer e Parser BNF per tuProlog

Andrea Boccacci

8 novembre 2011

Indice

1	Introduzione	2
2	TokenizerLibrary	3
2.1	Tokens	3
2.2	Priorità dei tokens	4
2.3	Predicati della libreria	5
2.4	Formato di output	6
3	BNFParserLibrary	7
3.1	Formato delle produzioni grammaticali	7
3.1.1	Esempi di produzioni	8
3.2	Abstract Syntax Tree	8
3.2.1	Default AST	9
3.2.2	Potatura dei nodi	9
3.2.3	Mappatura dei nodi	10
3.3	Lookahead	12
3.4	LL{1} vs. LL{N}	13
3.5	Predicati della libreria	13
4	Java Parser	16
4.1	Esempi di parsing file java	16

Capitolo 1

Introduzione

Questo articolo descrive le librerie `alice.tuprolog.TokenizerLibrary` e `alice.tuprolog.BNFParserLibrary`, che servono per descrivere la sintassi di un linguaggio. La prima libreria implementa le funzionalità di un tokenizer, mentre la seconda quelle di un parser.

Capitolo 2

TokenizerLibrary

Questa libreria si occupa di implementare le funzionalità di un tokenizer.

2.1 Tokens

La libreria permette di specificare come devono essere costruiti e quali sono i tokens di un linguaggio, a questo scopo sono utilizzabili tre famiglie di token:

- *Skip Token* sono tokens che sono leciti all'interno della frase da riconoscere, ma non sono generati in output dal tokenizer.
- *Normal Token* sono tokens rappresentati da stringhe statiche.
- *Type Token* sono tokens appartenenti alla stessa classe descritta dal tipo specificato dall'utilizzatore.

Gli *skip tokens* devono essere definiti all'interno della teoria prolog utilizzando il seguente predicato:

`skiplist L.`

dove L è una lista prolog i cui elementi devono essere stringhe regex valide. Per quanto riguarda i *normal tokens* ci sono più modi per definirli. Il primo è simile a quanto abbiamo visto in precedenza:

`tokenlist T.`

T è una lista prolog i cui elementi sono stringhe che rappresentano i *normal tokens*

Un metodo alternativo è quello di definire un token alla volta utilizzando il predicato:

`token T.`

In questo caso *T* è una stringa che rappresenta il token.

Il token prodotto in output per default viene rappresentato con la stessa stringa che lo descrive, questo comportamento può essere cambiato includendo nella teoria:

`token T becomes NEWNAME.`

In questo modo il token generato sarà rinominato in *NEWNAME* dal tokenizer.

Per quanto riguarda i tipi di token, invece, bisogna definire le stringhe che vi appartengono, questo è possibile scrivendo una stringa regex che riconosca se i token appartengono alla classe voluta.

Bisogna quindi introdurre nella teoria prolog il fatto :

`type TYPENAME : REGEX.`

dove *TYPENAME* è il nome del tipo di token e *REGEX* è la stringa regex descritta in precedenza. L'output di questa famiglia di token è leggermente diverso dalla precedente, infatti è costituito da un predicato con arità 1, avente come funtore *TYPENAME* e come argomento il token riconosciuto (utilizzando il gergo tecnico il match della regex).

La libreria fornisce, inoltre, funzionalità per modificare l'output di default.

A questo scopo si può modificare il fatto precedente in

`type TYPENAME : REGEX as X becomes Y.`

oppure

`type type TYPENAME : REGEX as X becomes Y where {C}.` In questo modo si unifica il token di tipo riconosciuto nella variabile *X* e si restituisce in output il token costruito come *Y*, se serve introdurre codice prolog per la modifica o per un check, può essere scritto all'interno delle parentesi graffe, al posto di *C*.

2.2 Priorità dei tokens

Nella fase di riconoscimento è importante sapere quali siano le priorità usate nella libreria nel riconoscimento dei tokens al fine di non avere effetti indesiderati. I token appartenenti alla famiglia *skip* vengono valutati per primi, in seguito vengono valutati i *type* e solo alla fine i *normal tokens*. Questa scala

è modificata solo ne caso in cui un *normal token* è equivalente al match del *type token*, dove il primo acquista una priorità maggiore. Per esemplificare questo ultimo caso se all'interno della teoria prolog si introduce

```
type literal : '[a-z]+'.
```

```
tokenlist [prova].
```

e la frase su cui si invoca il tokenizer è *'provaa'* verrà generato un solo token *literal(provaa)* a causa della maggior priorità dei token *type*.

Se invece la frase è *'prova'*, verrà generato sempre un unico token ma in questo caso un *normal token* *prova*, perché il match del token di tipo è equivalente al token e le priorità si invertono.

Se dal primo esempio si volesse dare maggiore precedenza ai *normal tokens* piuttosto che rispetto ad un tipo di dato e generare al posto di un unico token, due tokens *prova* e *literal(a)* bisognerebbe modificare la teoria in questo modo:

```
type lit : '[a-z]+' as X becomes literal(X) where {tokenlist T,  
(member(E,T),atom_concat(E,_,X),!,fail,true)} .
```

```
tokenlist [prova].
```

con il codice prolog all'interno delle parentesi graffe, si fa infatti fallire la generazione di un unico token nel caso il suo prefisso sia un token appartenente alla *tokenlist*. E' consigliato scrivere i token all'interno della *tokenlist* in ordine di lunghezza, in caso contrario il riconoscimento potrebbe non essere esatto per due tokens distinti di cui uno sia prefisso nell'altro. La priorità all'interno della *tokenlist* va in base alla posizione.

2.3 Predicati della libreria

In seguito sono elencati i predicati utilizzabili dall'utente presenti nella libreria:

- `tokenize/2`
`tokenize(In,Tokens)` Invoca il tokenizer sulla stringa *In* fornita in ingresso e restituisce una lista di tokens, se il processo va a buon fine.
Template: `tokenize(+string,-list)`
- `file_tokenize/2`
`file_tokenize(File,Tokens)` Invoca il tokenizer sul *File* fornito in

ingresso e restituisce una lista di tokens, se il processo va a buon fine.

Template: `file_tokenize(+string,-list)`

2.4 Formato di output

Il tokenizer restituisce in output una lista di tokens, ogni token è un predicato con funtore *token* e arità 3, gli argomenti sono in ordine token riconosciuto (vedi 2.1), indice di riga, indice di colonna. In questo modo il parser potrà restituire un errore in caso di insuccesso, indicando la posizione del token non riconosciuto.

Capitolo 3

BNFParserLibrary

Questa libreria si occupa di fornire un supporto per il parsing e per la generazione dell' *Abstract Syntax Tree*. Come si deduce dal nome la sintassi del linguaggio deve essere specificata secondo il formato BNF, con alcune differenze legate a motivi implementativi che verranno trattate nel prossimo paragrafo. La libreria non gestisce grammatiche di tipo 0 e di tipo 1.

3.1 Formato delle produzioni grammaticali

Come scritto in precedenza le produzioni grammaticali devono essere in formato bnf. Per definire una produzione grammaticale basta includerla nella teoria, tenendo conto di queste specifiche:

- A destra vi deve essere un unico simbolo non terminale sotto forma di atomo prolog.
- A sinistra possono esservi invece un numero indefinito di simboli terminali e non.
- Un simbolo non terminale deve essere specificato all'interno di una lista prolog.
- Tra la parte destra e quella sinistra vi è il simbolo '::<=' che ha la stessa semantica del rispettivo BNF-EBNF
- Per separare i simboli terminali e non alla sinistra si usano le virgole prolog.

- Per specificare più produzioni può essere utilizzato il predicato *'or'* che ha la stessa semantica del simbolo *pipe* in BNF-EBNF. *NOTA:* è quindi un operatore meno prioritario rispetto alla virgola prolog.
- Possono essere utilizzate parentesi tonde all'interno delle produzioni.

3.1.1 Esempi di produzioni

Questa sezione mostra alcuni tipi di produzioni accettate dalla libreria.

`a ::= [a],a.`

`a ::= [].`

Questo è un tipico esempio di ricorsione a destra la grammatica prodotta è $\{a\}$ ovvero una sequenza di zero o più *'a'*. Se si volesse utilizzare un solo fatto prolog lo stesso risultato si otterrebbe con:

`a ::= [a],a or [].`

analoghe riflessioni possono essere fatte per la ricorsione a sinistra:

`a ::= a,[a] or [].`

Per definire invece una sequenza di una o più *'a'* si può scrivere:

`a ::= [a],b.`

`b ::= [a],b or [].`

oppure:

`a ::= [a],(a or []).`

Per quello che riguarda le grammatiche di tipo 2, invece, la grammatica

$$a^n b^m c^{n+m}$$

può essere scritta, ad esempio, nel seguente modo:

`a ::= [a],a,[c] or b.`

`b ::= [] or [b],b,[c].`

3.2 Abstract Syntax Tree

Il risultato della operazione di parsing è un Abstract Syntax Tree, questa libreria permette di costruire l'albero a partire da alcune regole specificate dall'utilizzatore.

3.2.1 Default AST

La libreria fornisce in output un albero di default i cui nodi sono costituiti da predicati prolog i cui funtori sono dati dai nomi delle produzioni e i cui argomenti sono simboli terminali prodotti oppure nodi legati alla produzione di un simbolo non terminale. Ad esempio date le regole che definiscono una grammatica per una serie di somme e sottrazioni con simboli terminali `'+', '-', 'num(X)'` e scopo `add` :

```
add ::= [num(_)], add_rest.
```

```
add_rest ::= [] or (['+'] or ['-']), add.
```

l'AST di default prodotto dalla frase `'1'` sarà:

```
add(num('1'), add_rest)).
```

Infatti la produzione `add` ha due figli, un numero e un simbolo non terminale `add_rest`, che a sua volta non ha figli quindi il predicato relativo al suo nodo ha arità 0.

Se invece la frase da riconoscere fosse `'1+2'` l'output prodotto sarà:

```
add(num('1'), add_rest('+', add(num('2'), add_rest)))
```

Notare che in questo caso il simbolo non terminale `add_rest` produce a sua volta un simbolo terminale `('+')` e un simbolo non terminale `add`.

Si può subito notare che questo AST, sebbene sia omnicomprensivo di tutti i passi per arrivare alla frase della grammatica, non sia comprensibile da un operatore umano, per questo scopo sono state disposte funzionalità per controllare la creazione e la crescita dell'AST prodotto.

3.2.2 Potatura dei nodi

La libreria permette uno stretto controllo sui nodi generati, in particolare offre tre scelte:

- Non generare il nodo se questo è vuoto.
- Non generare il nodo se questo ha un unico figlio.
- Non generare il nodo in nessun caso.

La prima scelta si applica introducendo nella teoria il fatto:

```
dont_gen_if_void X.
```

Si vedano ad esempio le righe 753 e 883 della teoria *JavaParser.pl*

Analogamente le altre scelte si dichiarano:

`dont_gen_single_node X.`

Si vedano ad esempio le righe 356 e 953 della teoria *JavaParser.pl*

`dont_gen_node X.`

Si vedano ad esempio le righe 364 e 469 della teoria *JavaParser.pl*

dove *X* è il nome del nodo.

Per le prime due scelte, inoltre, è possibile applicare gli effetti a tutto l'albero senza specificare quindi il nodo da tagliare inserendo rispettivamente i fatti:

`dont_gen_if_void_all.`

`dont_gen_single_node_all.`

Nota: per motivi implementativi queste ultime due funzionalità, al contrario di quelle precedenti che non hanno questa limitazione, tagliano solo i nodi prodotti da simboli non terminali inclusi nella teoria corrente. Ad esempio se si rinomina il funtore di un nodo prodotto da una data produzione quel nodo non subirà gli effetti del taglio.

Per quanto riguarda invece i risultati dell'esecuzione nel caso il nodo tagliato sia vuoto esso non viene generato e quindi non sarà incluso tra gli argomenti del nome padre, negli altri due casi, invece, il nodo non generato innesterà i suoi argomenti in quelli del nodo padre a partire dalla sua posizione.

3.2.3 Mappatura dei nodi

Un'altra funzione per modificare l'AST risiede nella mappatura dei nodi. Questa funzione permette di specificare un sotto-albero e modificarlo (mapping) in un altro. L'utilizzo corretto di questa funzionalità è legato alla comprensione del suo funzionamento. Si naviga l'albero di default generato (vedi 3.2.1) a partire dalle foglie, si prova a vedere se uno dei sotto-alberi specificato unifica in senso prolog con la porzione di albero attualmente navigato (*NOTA:* solo il primo predicato di mapping che unifica e non fallisce viene scelto, non ci saranno soluzioni alternative) e se questa operazione ha successo viene generato l'albero secondo le regole definite dall'utilizzatore.

Attenzione: da quel momento in poi i nodi di livello superiore vedranno e tenteranno di unificare l'albero modificato secondo la regola scelta. Per specificare una regola di mappatura bisogna introdurre all'interno della teoria:

`map SRC : DST.`

dove SRC è l'albero sorgente che verrà modificato e DST è l'albero risultato dalla modifica.

E' inoltre possibile utilizzare codice prolog per trasformazioni che lo richie-

dono:

```
map SRC : DST :- PROLOG.
```

In questo modo si possono anche far fallire alcune regole che altrimenti unificherebbero con l'albero sorgente, e produrrebbero l'albero risultato, eventualmente lasciando spazio a regole successive.

Esempi di mappatura

Prendendo in esame la seguente grammatica con scopo *add*:

```
add ::= [num(C)],add_rest.
```

```
add_rest ::= [] or (['+'] or ['-']),add.
```

il risultato di parsing della frase '1', come descritto in precedenza sarà:

```
add(num('1'),add_rest)
```

Se si vuole ottenere come risultato invece l'albero `num('1')` si hanno due strade, la prima dichiarare che *add_rest* e *add* non debbano essere rispettivamente vuoti e a nodo singolo, l'altro modo è quello di utilizzare una regola di mapping ad esempio:

```
map add(X,add_rest) : X.
```

modificando la frase in '1+2', otteniamo come AST di default:

```
add(num('1'),add_rest('+',add(num('2'),add_rest)))
```

se invece si volesse ottenere un AST del tipo `'+'(1,2)` oltre alla precedente regola di mapping si può introdurre:

```
map add(X,add_rest(OP,Y) : Z :- Z=.. [OP,X,Y] . in questo modo si ot-  
tiene un risultato che è associativo a destra per le operazioni, infatti aumen-  
tando il numero di operazioni ad esempio con la frase '1+1-2-3+4' si ottiene  
con solo le precedenti due regole:
```

```
'+'(num('1'), '-'(num('1'), '-'(num('2'), '+'(num('3'), num('4')))))
```

Se si volesse invece che la somma e la sottrazione siano associative a sinistra serve qualcosa di un po' più elaborato, ad esempio:

```
map add(X,add_rest(OP,Y)) : Z :- left_ass(X,OP,Y,Z) .
```

```
left_ass(X,OP,Y,Z) :- Y=.. [OP1,X1,Y1],left_ass(X,OP,X1,Z1),Z=.. [OP1,Z1,Y1] .
```

```
left_ass(X,OP,Y,Z) :- Z=.. [OP,X,Y] .
```

In questo modo si innesta l'operazione all'interno del sotto-albero generato ottenendo il risultato richiesto:

```
'+'('-'('-'('+'(num('1'),num('1')),num('2')),num('3')),num('4'))
```

Se si volesse usare un'unica lista contenente l'operazione si possono usare ad esempio queste regole:

```

map add(X,L) : [X|L].
map add_rest : [].
map add_rest(X,L) : [X|L].

```

ottenendo il risultato voluto:
`[num('1'), '+', num('1'), '-', num('2'), '-', num('3'), '+', num('4')]` Si
 faccia infine attenzione alla presenza di regole in conflitto ad esempio:

```

map num(X) : X.
map num(X) : number(X).

```

Verrà scelta solo la prima regola definita perché essa non può mai fallire facendo risultare la seconda regola inutile. Diverso sarebbe il caso se la prima regola avesse possibilità di fallire, la seconda regola in quel caso entrerebbe in campo, al fallimento della prima. Ad esempio:

```

map num(X) : N :- num_atom(N, X), N<2.
map num(X) : number(X).

```

In questo caso se il numero rappresentato è minore di 2 viene eseguita la seconda regola (la prima fallisce), altrimenti la prima regola.
 Per ulteriori esempi di predicati di mapping si veda la teoria `JavaParser.pl` ad esempio alle linee 320-327.

3.3 Lookahead

Nota: Questa funzionalità non è stata testata a fondo quindi il suo funzionamento potrebbe avere qualche problema.

Il lookahead è da intendersi come taglio di rami che sicuramente non generano la soluzione corretta ad esempio se la grammatica fosse LL1, una volta trovata la produzione che produce il prossimo token, sarebbe inutile verificare le altre. Se queste strade venissero lasciate aperte, l'esecuzione del parser potrebbe richiedere molto più tempo del dovuto in caso di errore. Questa funzionalità è utile solamente in quest'ultimo caso.

La libreria permette l'utilizzo di due tipi di lookahead.

- Lookahead globale che influenza tutte le produzioni grammaticali
- Lookahead di una singola produzione.

In ogni produzione per poter utilizzare questa funzionalità deve essere comunque definito uno dei due tipi di lookahead. Nel caso entrambi fossero definiti, solo il lookahead locale avrà effetto

Per definire il lookahead globale bisogna inserire nella teoria il fatto:

`global_lookahead NUM.`

dove *NUM* è il numero di token che serve produrre per scegliere univocamente la produzione. mentre per quanto riguarda il secondo tipo:

`lookahead PROD : NUM`

dove *PROD* è la produzione alla quale si vuole applicare il lookahead locale, mentre *NUM* è il numero di token che devono essere prodotti per determinare la produzione da scegliere.

3.4 LL{1} vs. LL{N}

Sebbene siano correttamente gestite grammatiche del tipo LL{N} è giusto fare un appunto sul loro utilizzo. Scrivendo una grammatica non LL{1} vi è infatti un *side effect* non trascurabile sul degrado delle prestazioni, infatti, meno una grammatica è LL{1} più spesso entrerà in gioco il processo di backtracking, che spesso è molto oneroso, specialmente se si erano riconosciuti parecchi tokens, o se la frase da riconoscere è piuttosto lunga.

Si consiglia pertanto di utilizzare una grammatica LL{1} nelle produzioni che avvengono più frequentemente e molto distanti dalla simboli terminali della gramantica, mentre è solo relativamente importante utilizzare questi accorgimenti in produzioni che sono parecchio vicine ai simboli terminali.

3.5 Predicati della libreria

In seguito sono elencati i predicati utilizzabili dall'utente presenti nella libreria:

- `parse/4`
`parse(Mode,Tokens,Scope,AST)` Invoca il parser sulla lista di tokens fornita in ingresso (con lo stesso formato di uscita dei predicati `tokenize/2` e `file.tokenize/2` della `TokenizerLibrary`) verificando se appartiene ad una frase con scopo *Scope*. *Mode* deve essere *right* se la grammatica definita nella teoria è ricorsiva a destra, *left* se invece è ricorsiva a sinistra. Non sono utilizzabili grammatiche con parti ricorsive a destra e altre ricorsive a sinistra. Il predicato non fallisce mai

restituisce in output *AST* l'albero generato (quello di default) in caso di successo, altrimenti fornisce un messaggio di errore del tipo:
`error(num_token_removed(NUM) ,last_token_removed(TOK) ,
line(ROW) ,column(COL))`
dove *NUM* è il numero di tokens consumati durante il processo, *TOK* è l'ultimo token rimosso, *ROW* è l'indice di riga al quale si trova, *COL* è l'indice di colonna.

Template: `parse(+atom ,+list ,+atom ,-Term)`

- `cut_tree/2`

`cut_tree(SRC ,AST)` Taglia i nodi dell'albero *SRC* come spiegato nel paragrafo 3.2.2 e restituisce l'albero risultato.

Template: `cut_tree(+Term ,-Term)`

- `ast_map/2`

`ast_map(SRC ,AST)` Modifica l'albero *SRC* secondo le regole di mapping come spiegato nel paragrafo 3.2.3 e restituisce l'albero risultato.

Template: `ast_map(+Term ,-Term)`

- `validate_grammar/1`

`validate_grammar(E)` Controlla che tutte le produzioni siano definite, restituendo o un messaggio di successo o uno di errore se qualche produzione non è definita. Il messaggio di errore è del tipo: `production_not_defined(TAR_PROD) ,in_production(SRC_PROD)` dove *TAR_PROD* è il nome della produzione non definita, mentre *SRC_PROD* è il nome della produzione in cui viene utilizzata. Questo predicato è molto utile in caso di errori di battitura.

Template: `validate_grammar(-List_or_atom)`

- `find_grammar_loop/2`

`find_grammar_loop(Mode,E)` Controlla che non vi siano loop nella grammatica definita che non farebbero terminare il processo di parsing, *Mode* può essere *right* se si cercano loop nelle grammatiche ricorsive a destra, mentre *left* se si cercano nelle grammatiche ricorsive a sinistra. Restituisce il messaggio *no_loops_found* in caso di loop non trovati, oppure un errore del tipo: `loop_found_in_production_list(L)` dove *L* è la lista di produzioni che verificano il loop.

Template: `find_grammar_loop(-List_or_atom)`

- `lookahead_parse/4`
`lookahead_parse(Mode, Tokens, Scope, AST)` Funzionamento identico al predicato `parse/4`, ma tenendo conto dei lookahead specificati come da paragrafo 3.3 . *Attenzione:* Questa funzionalità non è stata testata a fondo e potrebbe avere alcuni problemi.
Template: `lookahead_parse(+atom, +list, +atom, -Term)`

Capitolo 4

Java Parser

Un esempio di utilizzo di entrambe le librerie è fornito nella teoria `prolog JavaParser.pl` che implementa il parser del linguaggio java secondo la grammatica descritta nella Java Language Specifications. La teoria utilizza tutte le funzionalità descritte a parte quella di lookahead. Nel file è utilizzata una formattazione consigliata per la teoria. Le regole di mapping e di potatura dei nodi sono inserite appena dopo la produzione grammaticale a cui si riferiscono.

4.1 Esempi di parsing file java

Per utilizzare il parser bisogna caricare la teoria *JavaParser.pl*
Dato questo semplice esempio di classe java:

```
package unibo;
import java.util.*;
public abstract class Prova {
    public Prova(){}
}
```

otteniamo come risultato l'albero:

```
compilation_unit(package(unibo),import('java.util.*'),
type_declaration([class(modifiers([public,abstract]),identifier('Prova'),
```

```
class_body([constructor_declarator(
modifiers([public]),identifier('Prova'),formal_parameters([]),
method_body([])))]))
```

Possiamo subito notare che la produzione degli import non è contenuta in una lista, questo perchè l'import è unico, se ne file vi fossero specificati più import il risultato sarebbe stato la creazione di una lista di import. All'interno della classe è definito solo il costruttore senza parametri e senza corpo. Guardando l'albero generato questi elementi vengono messi in risalto. Complicando leggermente il codice:

```
public class Prova {
    private int integer;
    public Prova(int arg1){
        this.integer = arg1;
    }
}
```

otteniamo:

```
compilation_unit(type_declaration(
[class(modifiers([public]),identifier('Prova'),
class_body([
field_decl(modifiers([private]),type(int),identifier(integer)),
constructor_declarator(
modifiers([public]),identifier('Prova'),formal_parameters
([formal_parameter(type(int),variable_id(arg1))]),
method_body([statement(assign(
select(this,identifier(integer)),arg1))])])])])
```

Possiamo notare che è stato riconosciuto un campo di tipo intero denominato *integer*, il costruttore questa volta ha un parametro formale *arg1*, il corpo del costruttore è composto da uno statement che assegna il valore *arg1* al campo *integer*, il nodo *select* indica appunto la selezione di un campo o di un metodo appartenente alla classe.

Richiamando, invece, il tokenizer e il parser su un file java contenete:

```

public abstract class Prova{
    public void doJob(){
        int u = 1+2+System.out.println("hello world!");
    }
}

```

otteniamo in risultato l'albero:

```

compilation_unit(type_declaration([class(modifiers([public,abstract]),
identifier('Prova'),class_body(
[void_method_declarator(modifiers([public]),
identifier(doJob),formal_parameters([]),
method_body([local_variable_declaration(
type(int),vars([variable_decl(assign('identifier(u)',
plus(decimal('1'),plus(decimal('2'),
method_call('System.out.println',arguments([string('ciao'))]))))
]))]))]))))

```

possiamo subito notare che l'albero è fatto in modo che in un unico file possano essere definite più classi, infatti la dichiarazione dei tipi è contenuta in una lista, lo stesso vale per i modificatori. Per quello che riguarda i metodi sono divisi in metodi che non restituiscono un risultato e metodi che restituiscono un risultato. All'interno del metodo abbiamo una dichiarazione di variabile e un assegnamento. Possiamo infine notare che come previsto il parser controlla unicamente la sintassi del linguaggio. In questo esempio abbiamo un errore semantico per un'operazione tra tipi di dato non affini, questa operazione deve essere eseguita infatti eseguita in fase di checking.

Introducendo invece un errore nella classe, ad esempio:

```

public class Prova {
    private int integer;
    public Prova(int arg1){
        this.integer = arg1;
    }
}
//} ho cancellato questa parentesi errore!

```

Si ottiene l'errore:

```
error(num_token_removed(22),  
last_token_removed(rbrace),line(5),column(2))
```

Che dice che sono stati riconosciuti correttamente 22 tokens, e l'ultimo token rimosso è una parentesi graffa destra alla linea 5 colonna 2, segnalando quindi la parentesi commentata. Come scritto sopra è possibile provare con file molto più complessi. Si provi ad esempio con qualche file di tuProlog.