

Relazione Tokenizer e Parser BNF per tuProlog

Andrea Boccacci

8 novembre 2011

Indice

1	Introduzione	2
2	Tokenizer	3
3	BNF Parser	5
4	Java Parser	7
4.1	Esempi di parsing file java	8
5	Possibili miglioramenti	11

Capitolo 1

Introduzione

Questo progetto ha come scopo quello di emulare le funzionalità di JavaCC e includerle in tuProlog. Il risultato è stato lo sviluppo di due librerie, una che si occupa del tokenizing, l'altra che si occupa del parsing. Per testare il risultato ottenuto ho inoltre scritto una teoria prolog che utilizza le suddette librerie per costruire un parser del linguaggio java.

L'utilizzo delle funzionalità della libreria è descritto nel *manuale d'uso* in questa relazione tratterò dei problemi affrontati, delle scelte effettuate, dei possibili miglioramenti della libreria e dei risultati ottenuti.

Capitolo 2

Tokenizer

Per ottenere buone prestazioni in questa fase ho evitato la conversione del testo in ingresso in codici ascii che occupava buona parte del tempo di esecuzione, ho quindi deciso di lavorare direttamente sulle stringhe implementando due predicati in java:

- La *string_append/3*, che lavora come la *atom_concat/3* ma direttamente con le stringhe quindi particolarmente più veloce soprattutto per conversioni di grossi file
- la *string_match/3*, che si occupa di restituire il match di una stringa regex su una stringa, entrambe fornite in ingresso, solo se esso si trova in posizione iniziale. Ho fornito inoltre una seconda implementazione dello stesso predicato, ma con arità 4 la cui unica differenza che restituisce anche la posizione del primo match della stringa regex.

L'utilizzo delle stringhe regex migliora di molto le prestazioni, inoltre fornisce un valido supporto all'utilizzatore per specificare anche tokens molto complessi. Ad esempio nella teoria del parser del linguaggio java ho fornito la regex che riconosceva il commento multilinea.

sono stati pensati tre tipi di tokens, gli skip, i token e i tipi di token, le loro priorità e il loro utilizzo sono descritti nel manuale d'uso, al relativo paragrafo.

Il tokenizer riconosce l'indice di riga e di colonna del token, tuttavia l'implementazione dell'indice di riga è stata effettuata ad-hoc sul token relativo alla *newline*, purtroppo penso non si possa fare in altro modo. Per quanto riguarda invece l'indice di colonna, è utilizzato il predicato *atom_length/2*

che restituisce la lunghezza di un atomo e non è stato un problema implementare la funzionalità. Sono state inoltre messe a disposizione dell'utilizzatore delle funzioni per controllare l'output del token, esse sono descritte nel manuale d'uso.

Capitolo 3

BNF Parser

Ispirandomi alle specifiche JavaCC ho trovato comodo implementare un parser che riuscisse a gestire grammatiche scritte secondo il formato BNF. Tuttavia, sul piano implementativo, sono riuscito a pensare una buona rappresentazione di default per l'AST solo su grammatiche di tipo 2 e di tipo 3, per cui il supporto della libreria è limitato a questi due tipi. Ho scelto inoltre di delegare la trasformazione dell'albero a due predicati costruiti appositamente per le due funzionalità in modo tale che possano essere utilizzati anche indipendentemente dal tipo di parser utilizzato, ma soprattutto per snellire il processo di parsing e i tempi di risposta in caso di errore.

La gestione degli errori si è rivelata molto utile sul piano pratico, la sua implementazione, però, non è stata particolarmente semplice. Ho infatti dovuto utilizzare i predicati *assert* e *retract* per tener traccia dell'ultimo token correttamente riconosciuto e restituire l'errore in caso di fallimento, non ho trovato altri modi per implementare questa funzionalità a causa del processo di backtracking, che è necessario utilizzare quando il parser non è LL1.

Ho inoltre pensato che fosse opportuno che la libreria trattasse anche grammatiche ricorsive a sinistra anche se non sono trattate in JavaCC. Questo permette di gestire in maniera più semplice l'associatività a sinistra degli operatori.

L'implementazione è simile al parsing a destra, con l'unica differenza che la produzione dei token avviene da destra verso sinistra.

Ho inoltre introdotto due predicati utili durante la stesura delle grammatiche, *validate_grammar/1* e *find_grammar_loop/2* che sono descritte nel ma-

nuale d'uso. Per un errore di valutazione iniziale ho preferito non implementare il lookahead, tuttavia la scelta è risultata sbagliata, infatti non ho tenuto conto nella valutazione iniziale del caso in cui la frase da riconoscere non appartenga alla grammatica e la grammatica sia al tempo stesso molto complessa. In questo caso infatti una guida dettata dal lookahead sarebbe stata molto utile al fine di tagliare rami che non contengono soluzioni. Non avendo eliminato questi rami, il processo di backtracking viene quindi richiamato tantissime volte, facendo crollare drasticamente le prestazioni in caso di errore. Nel tempo rimanente ho quindi implementato una proposta di soluzione a questo problema, ma non l'ho testata a fondo, per cui potrebbe avere qualche problema. Per questo motivo ho preferito fornire due tipi di predicati per il parsing *lookahead_parse/4* e *parse/4*, di cui solo il primo tiene conto del lookahead.

Il processo di riconoscimento è piuttosto veloce invece nel caso la frase da riconoscere sia appartenente alla grammatica specificata.

Capitolo 4

Java Parser

Obiettivo di questa parte di progetto è quello di creare un parser per il linguaggio Java versione 1.6. Il file contenente la teoria è *JavaParser.pl*, per il funzionamento basta caricarlo in tuProlog.

L'utilizzo delle librerie sviluppate e delle funzioni si è rivelato molto utile all'atto della stesura di questo esempio di utilizzo, la gestione degli errori si è rivelata uno strumento efficace nel fornire un aiuto nel cercare eventuali errori nella stesura della grammatica. La *validate_grammar/1* è stata utilissima nel rilevare errori di battitura nella stesura della grammatica, e la *find_grammar_loop/2* nella ricerca di eventuali loop che non facevano terminare l'esecuzione.

La grammatica fornita nella Java language specifications è stata modificata tenendo conto della priorità degli operatori e in un errore nel riconoscimento delle espressioni del tipo *instanceof*, la grammatica da loro proposta permetteva l'utilizzo della parola chiave *instanceof*, solo alla fine dello statement relativo ad un'espressione. Questo non sempre è vero, infatti dopo una espressione *instanceof* potrebbero esserci ulteriori espressioni. Un esempio è fornito da:

```
t instanceof Number && ((Number)t).isReal()
```

valida in java, dove si nota chiaramente che l'espressione *instanceof* è seguita da un operatore e un'altra espressione booleana.

Sono state riscritte quelle parti di grammatica che risultavano ricorsive a sinistra per renderle compatibili con il resto della grammatica che risultava ricorsiva a destra.

Le funzionalità di trasformazione dell'AST si son rivelate di buon livello e

hanno permesso la generazione di un albero non troppo complesso. Come esempio di utilizzo della libreria è abbastanza esaustivo e mi ha permesso di sfruttare quasi tutte le funzionalità offerte. Parlando di prestazioni, l'esempio fornito riesce a riconoscere anche file *java* piuttosto grandi in un tempo relativamente piccolo (200 righe in circa 30 secondi), ma questo dipende anche dalla struttura del file, in caso di errore invece i tempi di risposta sono abbastanza elevati a causa del problema descritto sopra. Implementando le specifiche di lookahead, questo problema verrà probabilmente risolto.

4.1 Esempi di parsing file java

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.

Capitolo 5

Possibili miglioramenti

Sono possibili ancora alcuni miglioramenti alla libreria:

- Trovare un modo migliore per gestire le newline nel tokenizer.
- Permettere all'utilizzatore di specificare la priorità dei tipi di token.
- Testare il lookahead e fornire una implementazione funzionante, se quella fornita avesse qualche problema.
- Supporto per grammatiche di tipo 1.
- Supporto per specifiche EBNF.