UNIVERSITÀ DEGLI STUDI DI CATANIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Progetto di Linguaggi e Traduttori

A.A. 2015-2016

COMPILATORE PER IL LINGUAGGIO MINIC

Docente:
Vincenza Carchiolo

Candidati:

Marco Castrianni

Giuseppe Grasso

Introduzione

L'obiettivo del progetto è quello di implementare un compilatore per il linguaggio MiniC che traduca il codice MiniC in un Albero Sintattico Astratto (AST) rappresentato in forma parentetica.

Per l'implementazione del compilatore sono stati utilizzati il generatore di analizzatori lessicali JLex e il generatore di analizzatori sintattici/semantici CUP.

È stato implementato un compilatore a due passi:
- nella prima fase vengono eseguite l'analisi lessicale e sintattica del codice in ingresso;
- nella seconda fase viene generato il corrispondente l'Albero Sintattico Astratto;

Il compilatore è in grado di gestire errori sia a livello lessicale che sintattico, producendo un opportuno messaggio relativo al tipo di errore.

Specifiche della grammatica

La sintassi del MiniC è definita dalla grammatica context-free mostrata nella Tabella 1.

Tabella 1. Grammatica del MiniC

<program>::= { <stmt>* }
<stmt> ::= <simp> ; | <control> | ;
<simp> ::= <ident> <asop> <exp>
<control> ::= if (<exp>) <block> [else <block>] | while (<exp>) <block>
<block> ::= <stmt> | { <stmti>* }
<exp> ::= (<exp>) | <intconst> | <ident> | <unop> <exp> | <exp> <binop> <exp>
<ident> ::= [A-Z a-z][0-9A-Z a-z]*
<intconst> ::= [0-9][0-9]
<asop> ::= = | += | -= | *= | /= | %=
<binop> ::= + | - | * | / | % | < | <= | > | >= | == | != | && | ||
<unop> ::= ! | -

Tool di supporto

Al fine di ottenere il nostro obiettivo, sono stati utilizzati JLex e CUP.
JLex è un generatore di analizzatori lessicali per Java, sviluppato presso la Princeton University. Il download ed il manuale utente sono reperibili presso il link https://www.cs.princeton.edu/~appel/modern/java/JLex; è presente inoltre un file README che ci ha permesso di installare correttamente il tool.
Generare un analizzatore lessicale utilizzando JLex è di per sé semplice, basta infatti compilare il proprio file con estensione .lex utilizzando JLex.Main ed il risultato sarà una classe java contenente in codice dell'analizzatore. La parte più onerosa, che è stata oggetto del nostro studio, è ovviamente la scrittura del file di specifica per la nostra grammatica; sia il file di specifica, ***Yylex.lex***, sia il codice dell'analizzatore, ***Yylex.java***, sono reperibili nella cartella del progetto: ***MiniC/src/scanner/.***

CUP (acronimo di **Construction of Useful Parsers**) è un generatore di parser LALR(1) per Java. Tutto l'occorrente per l'installazione del tool è reperibile al link http://www2.cs.tum.edu/projekte/cup/index.php. In maniera analoga all'analizzatore lessicale, per generare un parser con CUP occorre partire da un file di specifica della grammatica in formato .cup; sarà poi il tool a generare il rispettivo codice Java. Anche in questo caso, oggetto del nostro lavoro è stato la scrittura del file di specifica; entrambi i file (***parser.cup*** e ***parser.java***) si trovano nella directory: ***MiniC/src/scanner/.***

Implementazione

Dopo aver scaricato i file sorgenti di JLex e CUP e studiato le relative documentazioni, a partire dalla grammatica in tabella è stato scritto il file Yylex.lex contenente le specifiche relative al linguaggio MiniC per la generazione del file Yylex.java.
Oltre alle regole che forniscono i vari token al parser, sono state inserite delle regole che permettono di ignorare le varie tipologie di spazi vuoti e i commenti multi linea e su singola linea.
Successivamente è stato scritto il file Parser.cup; le ambiguità presenti nella grammatica sono state risolte dalle varie specifiche di ***precedence*** definite nel file.
Per la generazione dell'albero sintattico astratto, abbiamo sfruttato il meccanismo offerto da CUP che permette di associare ad ogni produzione una specifica azione semantica. Sono state create ad hoc alcune classi java che modellano le varie tipologie di nodi che può avere l'albero semantico. In termini pratici, quello che viene fatto a seguito di una produzione della grammatica è creare una istanza della classa java relativa a ciò che è stato prodotto nella grammatica, con i relativi valori.

Ad esempio nella produzione:

exp ::= INTCONST:c {: RESULT = Exp.intconst(c); :}

l'azione semantica porterà ad avere un oggetto di tipo IntConst il cui valore (c) sarà quello che sarà contenuto nella tabella dei simboli, opportunamente riconosciuto e salvato dall'analizzatore lessicale mediante l'istruzione:

{NUM} { return new Symbol(sym.INTCONST, new Integer(yytext())); }*

Per facilitare la lettura del codice, abbiamo mantenuto sostanzialmente una certa relazione tra i nomi dei terminali/non terminali della grammatica e i nomi delle classi Java di supporto; ad esempio la classe ***Program*** rappresenta un insieme di Stmt, la quale modella una generica istruzione, o ancora la classe ***Identifier*** (che estende ***Exp***) rappresenta un identificatore.
L'azione semantica associata ad ogni produzione della grammatica permette di chiamare le classi relative alla costruzione dell'albero sintattico .

Dopo aver creato i vari nodi dell'albero, è stata scritta una classe java, ***ASTGenerator***, che permette di trasformare il parse tree ottenuto dall'analisi sintattica in un Abstract Syntax Tree; la classe è stata scritta (in linea di massima) seguendo il pattern Visitor. L'output di tale classe è un file in formato testuale contenente una rappresentazione parentetica dell' AST.

Come da specifica dell'elaborato si è gestita l'error recovery che permette al compilatore di continuare il parsing e produrre in uscita un messaggio d'errore che specifichi il tipo di errore e linea di occorrenza dell'errore nel file d'ingresso.
CUP supporta una modalità di error recovery di tipo ***panic mode*** dove è prevista la presenza di un simbolo speciale **error** che viene posto nello stack ogni volta che un errore sintattico accorre. Per default, quando si ha un errore, il parsing termina con una eccezione; oggetto del nostro lavoro è stato quello di inserire esplicitamente nella grammatica delle produzioni contenenti ***error*** che ci hanno permesso di associare delle azioni per segnalare opportunamente l'errore ed al contempo continuare il parsing; in particolare ciò è stato implementato mediante l'override del metodo del parser ***report_error().***