Implementazione di un compilatore per il linguaggio MiniC che generi istruzioni per la Java Virtual Machine.

Progetto di Compilatori e Interpreti - a.a. 2011/2012

Prof.ssa V. Carchiolo
Alessandro Nicolosi • Riccardo Pulvirenti • Giuseppe Ravidà
Website: http://code.google.com/p/minic-to-jasmin/

Indice

INTRODUZIONE

Il presente elaborato ha come obiettivo la realizzazione di un compilatore per il linguaggio **MiniC** che generi istruzioni per la Java Virtual Machine (JVM).

Del compilatore è stato sviluppato il front-end, mentre la traduzione del codice intermedio è stata lasciata a **Jasmin**, un back-end che produce byte-code Java; pertanto, il codice prodotto può essere interpretato perfettamente dalla Java Virtual Machine.

MiniC è un linguaggio puramente didattico, fortemente ispirato al C, che fornisce semplici costrutti per il controllo di flusso, tipi primitivi essenziali e due tipi strutturati: array e stringhe.

IMPLEMENTAZIONE

Il compilatore è stato realizzato in ambiente Eclipse Java EE IDE con l'ausilio di alcuni tools come **JFlex** e **CUP** per l'analisi lessicale e semantica.

Lo stesso IDE mette a disposizione **Ant**, un tool Java Based che permette di lanciare le fasi di Building e Running del codice in maniera semplificata similmente al comando `make` di Linux OS.

Struttura Elaborato

Il progetto, denominato `minic-to-jasmin`, è suddiviso in cartelle secondo la seguente struttura:

- `src`
    - `ast`
    - `aux`
    - `compiler`
    - `m2j`
    - `utils`

Il package `m2j` contiene i quattro files principali di tutto il progetto, ovvero la classe `Main.java` per eseguirlo, la classe `JasminCompiler.java` che rappresenta il front-end di tutto il compilatore, il file `minic2jasmin.flex` contenente le espressioni regolari per l'estrapolazione dei token utilizzati nel linguaggio e il file `minic2jasmin.cup` contenente la grammatica MiniC.

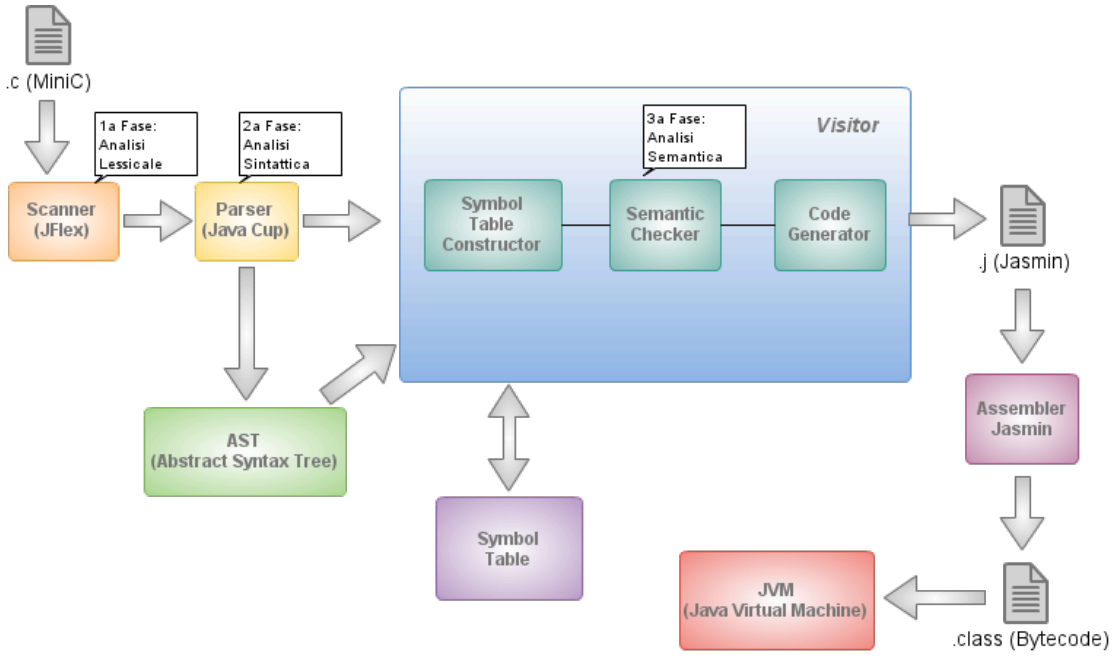
E' stata implementata inoltre una classe `GUI.java` che realizza una semplice interfaccia grafica per semplificare il processo di esecuzione del compilatore.

Nel package `ast` vi sono le classi utilizzate per la generazione dell'albero sintattico (AST). Esse rappresentano i vari nodi dell'albero, i quali verranno poi *visitati* durante le fasi di Type Checking e Generazione del codice JVM.

La visita dei nodi, è ottenuta utilizzando il pattern **Visitor**, che illustreremo nella sezioni successive. Le classi che lo implementano sono tutte contenute all'interno del package `compiler`.

Workflow

La figura sottostante, riporta uno schema di quelli che sono le fasi principali della compilazione da MiniC fino al codice per la Java Virtual Machine.



Tecniche utilizzate

Durante lo sviluppo del progetto si è fatto ricorso a diverse tecniche e strumenti per accelerare e semplificare la scrittura del codice e al tempo stesso rendere più robusta l'applicazione. Come accennato è stato utilizzato il pattern Visitor per la creazione della tabella dei simboli, analisi sintattica e generazione del codice, insieme a strumenti come JFlex e JCup. Analizziamo quindi nel dettaglio i singoli passi di progettazione descrivendo per ognuno specificatamente le tecniche utilizzate e i tool di supporto.

L'**analisi lessicale** (scanner) è ottenuta con il supporto di **JFlex**.

Per questa prima fase, vengono utilizzati i metodi `yyline` e `yycolumn` di JFlex che identificano rispettivamente numero di riga e di colonna di inizio e di fine di ogni token. Grazie a questa informazione l'emissione degli errori risulta molto precisa perchè permette di identificare porzioni specifiche di codice (e non semplicemente i numeri di riga).

I commenti annidati, su singola linea e il testo quotato, sono riconosciuti dallo scanner grazie all'utilizzo delle `start condition`; quando JFlex riconosce un commento attiva le regole specifiche della start condition e aggiunge lo stato su uno stack. Ad ogni chiusura di commento verrà tolto uno stato dalla pila fino a quando non si ritornerà allo stato iniziale. A questo punto l'analisi lessicale ripartirà con le regole normali.

L'**analisi sintattica** (parser) di MiniC è stata generata in modo automatico utilizzando **CUP**.

Al riconoscimento delle produzioni vengono svolti tutti i controlli semantici e si procede alla costruzione dell'albero sintattico astratto. Lo strumento essenziale in questa fase è la tabella dei simboli: essa associa ad ogni identificatore una serie di caratteristiche (tra cui il tipo per le variabili).

Symbol Table

La tabella dei simboli è gestita tramite la classe `SymbolTable` nel package `compiler`. Essa ha il compito di coordinare entrambi i simboli di variabili e funzioni, appoggiandosi alla classe `SymbolDesc` che descrive i dettagli per ogni simbolo.

La tabella dei simboli è composta da un `HashMap`:

```
HashMap<String,ArrayList<SymbolDesc> > storage = new
    HashMap<String,ArrayList<SymbolDesc> >();
```

Una generica entry nella mappa è fatta da una stringa che rappresenta l'ID del record e una lista di descrittori di simboli, che contengono alcune informazioni come il tipo del simbolo (`int`, `float`, etc.), la specie (variabile/funzione), il numero di blocco di codice (per gestire lo scope), una lista di parametri usati da una funzione e altri..

La classe `SymbolTable` espone due metodi principali per inserire elementi all'interno della collezione:

- `putVariable`
- `putFunction`

e alcuni metodi per estrapolare elementi dalla lista:

- `get`
- `getSpecific`
- `getVariableType`
- `getFunctionType`
- `getVarDesc`
- `getFuncDesc`

Il metodo `getSpecific` ritorna una lista di descrittori per uno specifico simbolo sia esso variabile o funzione.

I metodi `put[ ... ]` usano il `getSpecific` per controllare se un nuovo simbolo può essere aggiunto nella tabella. Per aggiungere una variabile, il metodo `putVariable` controlla se l'elemento esiste. In tal caso, il metodo controlla in che scope esiste in quanto è possibile avere variabili con stesso nome ma in scope differenti.

Degno di nota è il caso in cui sia possibile avere una funzione e una variabile con lo stesso nome. In questo caso avremo una singola entry per il simbolo con due descrittori: uno per la variabile e uno per la funzione:

Esempio:

```
extern "MinicLib/plootoh" float plootoh(int);

float main()
{
   int plootoh;
   int p;
   [...]

   return plootoh(p);
```

dove ogni descrittore è un'istanza della classe `SymbolDesc`:

ID	Descriptors					
plootoh	INT	null	IdType.VARIABLE	1	0	""
	FLOAT	[INT,0]	IdType.FUNCTION	0	-	MinicLib?/plootoh

In questa classe, ogni descrittore è composto dai seguenti attributi:

```
public class SymbolDesc{
    private IdType type;
    private IdType kind;
    private int nBlock;
    private ArrayList<IdType> paramList;
    private int dim;
    private int jvmVar;

   [...]
```

I metodi principali, `setVariableSymbol` e `setFunctionSymbol` creano rispettivamente un descrittore per una variabile o una funzione nella HashMap.

Pattern Visitor

Il Visitor è un design pattern comportamentale utilizzato in informatica nella programmazione orientata agli oggetti. Permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa. [1]

All'interno del package `compiler` è presente la classe astratta `Visitor`, la quale dichiara un metodo `visit` per ogni nodo dell'AST appartenente alla struttura di oggetti in modo che ogni oggetto della struttura possa invocare il metodo `visit` appropriato passando un riferimento a sé (`this`) come parametro.

Esempio:

```
public Object visit(BlockNode node)
{
      node.visitChildren(this);

      return null;
}
```

Questo permette al Visitor di identificare la classe che ha chiamato il metodo `visit`, eseguire il comportamento corrispondente e accedere all'oggetto attraverso la sua specifica interfaccia. [2]

Di seguito un estratto della classe Visitor:

```
public abstract class Visitor
{
    public abstract Object visit(AddNode node);
    public abstract Object visit(FloatNode node);
    public abstract Object visit(AndNode node);
    public abstract Object visit(ArgNode node);
    public abstract Object visit(AssignNode node);
```

Attraverso questo pattern si è semplificata notevolmente l'implementazione dei meccanismi di visita dell'AST. Nello specifico, le classi `SimbolTableConstructor`, `SemanticChecker` e `CodeGenerator` sono un esempio di Visitor utilizzati per le fasi di Symbol Table Creation, Type Checking e Code Generation.

Utilizzo del Pattern

La fase di **Type Checking** verifica i conflitti di tipi nelle assegnazioni e nelle chiamate a funzione. Tale compito è assegnato alla classe `SemanticChecker`.

Ogni qualvolta il visitor `SemanticChecker` incontra un nodo dell'AST, confezionerà un oggetto ausiliario (`NodeInfo` del package `aux`) che verrà ritornato ai metodi appositi per gestirne il check dei tipi.

Oltre al check dei tipi, la classe verifica la presenza di duplicati nella tabella dei simboli; ciò viene ottenuto tramite gli appositi metodi sopra citati della classe `SymbolTable`.

E' permessa la presenza di duplicati delle variabili purchè siano in scope differenti.

La fase di **Code Generation** genera un file contenente le istruzioni MiniC tradotte in Jasmin, un assembler per Java.

L'input di Jasmin è un semplice file di testo in cui è contenuta la descrizione delle classi Java per mezzo del set di istruzioni della Java Virtual Machine. In output sarà prodotto un file binario, contenente il byte-code di una classe Java.

I tipi delle variabili presenti in MiniC sono gestiti nel modo seguente: gli interi e i booleani sono trattati come valori primitivi interi, le stringhe sono degli oggetti il cui spazio in memoria viene allocato automaticamente a run-time, mentre per i vettori è necessario inserire appositi comandi assembler per allocarne la memoria necessaria (la loro dimensione può essere calcolata a run-time e posizionata sullo stack). I valori (primitivi o riferimenti ad oggetti) possono essere memorizzati in appositi registri.

L'assembler Java si basa su una macchina a pila: la maggior parte delle istruzioni richiedono di posizionare precedentemente sulla cima di uno stack i valori o i riferimenti necessari.

Ogni qualvolta il visitor `CodeGenerator` incontra un nodo dell'AST, confezionerà un oggetto ausiliario (`GenNodeInfo` del package `aux`) che verrà ritornato ai metodi appositi per gestirne la generazione delle rispettive istruzioni assembler.

Error Recovery

Durante le prime tre fasi (lessicale, sintattica e semantica), il compilatore segnala le situazioni di errore senza interrompere l'analisi del codice sorgente. Gli errori si possono suddividere in tre categorie:

1. **Errori lessicali:** commenti non chiusi, stringhe non chiuse, token non riconosciuti;
2. **Errori sintattici:** produzioni non riconosciute;
3. **Errori semantici:** conflitti di tipi nelle assegnazioni e nelle chiamate a funzione, utilizzo di funzioni mai dichiarate, utilizzo di identificatori non dichiarati, dichiarazioni multiple.

Per quanto riguarda gli errori sintattici, sono state aggiunte delle produzioni alla grammatica che, tramite l'utilizzo del token riservato **error** usato all'interno della grammatica, consente al parser di riprendere l'esecuzione dopo un eventuale errore.

Per default, un parser generato da CUP quando incontra un errore:

- Segnala chiamando la funzione `public void syntax_error(Symbol cur_token)` definita nella classe `java_cup.runtime.lr_parser` un errore sintattico scrivendo a schermo "*Syntax error*".
- Se tale errore non viene recuperato dal parser attraveso un opportuno simbolo `error`, il parser lo segnala richiamando la funzione `public void unrecovered_syntax_error(Symbol cur_token)`, anch'essa definita in `java_cup.runtime.lr_parser`. Tale funzione, dopo aver segnalato l'errore "*Couldn't repair and continue parse*" blocca l'esecuzione del parser.

Un'opportuna ridefinizione delle funzioni precedentemente elencate in parser code {: ... :}, permette di personalizzare la gestione degli errori.

Per ogni messaggio viene specificata la posizione della porzione di codice errato.

CONCLUSIONI

Durante lo sviluppo del compilatore sono state prese delle scelte atte a rendere il progetto il più modulare possibile e riutilizzabile anche in altri contesti.

Grazie all'utilizzo del pattern Visitor è infatti possibile far uso del compilatore per qualsiasi altro linguaggio, diverso da Jasmin, mantenendo inalterate le visite per la creazione della tabella dei simboli e del controllo sulla sintassi. Bisognerebbe in tal caso riscrivere soltanto il Visitor per la generazione del codice.

Il lavoro, grazie anche alla sua complessità, ha permesso al gruppo di estendere le proprie conoscenze sia in materia di compilatori e di programmazione che in ambito di gestione di progetti complessi e organizzazione del lavoro in gruppo.

Tale documentazione, insieme ad un piccolo wiki e al codice sorgente, è reperibile all'indirizzo: http://code.google.com/p/minic-to-jasmin/