Alma Mater Studiorum
Università di Bologna

Report

Progetto di Compilatori e Interpreti

Corso di Laurea Magistrale in Informatica

Giuseppe De Palma
0000854846

Andrew Memma
1900068459

25 novembre 2018

Indice

1 Introduzione

Il lavoro presentato riguarda un compilatore per un semplice linguaggio di programmazione ad oggetti: FOOL. Il progetto è parte dell'esame del corso "Compilatori e Intepreti" della Magistrale in Informatica, Università di Bologna. Il lavoro è stato svolto da 2 studenti del suddetto corso.

L'obiettivo di tale progetto è volto allo studio dello sviluppo e funzionamento dei compilatori. Partendo dal testo di un programma scritto in FOOL, il compilatore lo analizza sia per ricavarne le informazioni necessarie per generare codice eseguibile, sia per controllarne la correttezza. Si fa uso di diverse tecniche per cercare errori nei programmi (a tempo di compilazione). In particolare, il linguaggio si avvale di un semplice sistema di tipi e il compilatore esegue un controllo di tipi statico che permette di segnalare all'utente errori di tipo. Con questo controllo si ha il vantaggio di evitare i cosiddetti errori *untrapped* e una parte degli errori *trapped*, cioè l'insieme degli errori "proibiti".[1]

FOOL non è l'unico linguaggio utilizzato, infatti il compilatore deve generare del codice eseguibile (da un elaboratore) partendo da un programma scritto in FOOL. Questo nuovo codice è scritto in un linguaggio *assembly* per una *stack virtual machine* con registri (SVM), il quale offre istruzioni per alcuni calcoli aritmentici, per gestione di *stack* e *heap* e salti condizionati. In accordo con le estensioni fatte a FOOL, anche il linguaggio SVM è stato arricchito.

1.1 Outline

Nei capitoli successivi è presentato in modo approfondito il progetto e le sue caratteristiche principali. Nel capitolo successivo viene descritta la struttura del compilatore e come utilizzarlo e una breve analisi sulle diverse parti che lo costituiscono. Il capitolo 3 discute la grammatica del linguaggio e come da questa si ricavano le informazioni per proseguire con la compilazione. Nel capitolo 4 è discussa l'analisi semantica, di particolare importanza è la sezione del *type checking*. Prima di concludere è descritta, nel capitolo 5, la generazione del codice *svm*.

[1] Distinguiamo due generi di errore in un programma: errori *trapped*, ovvero quelli che provocano il fallimento della computazione, e gli errori *untrapped*, che non presentano sintomi immediatamente visibili, e sono perciò più insidiosi. Al primo tipo appartengono, ad esempio, la divisione per zero o un accesso proibito alla memoria; al secondo la lettura erronea di una porzione non significativa della memoria.

2 Descrizione del Progetto

2.1 Struttura

Il compilatore è stato sviluppato nel linguaggio di programmazione Java, utilizzando il *tool* Antlr nella sua versione 4.6 per la generazione automatica di *lexer* e *parser*. Avendo, quindi, definito la grammatica di FOOL (nel file *FOOL.g4*), il *tool* genera le varie classi che permettono di effettuare l'analisi lessicale e sintattica sui programmi scritti in FOOL. In questo modo la creazione di *tokens* e la creazione dell'albero astratto di sintassi (*Abstract Syntax Tree*) sono già gestite da Antlr, il quale fornisce due metodi per visitare ed utilizzare l'albero: i *visitor* e *listener* design patterns. Per questo progetto il *visitor pattern* è stato scelto per visitare l'albero ed ottenere il necessario per effettuare l'analisi semantica e la generazione di codice.

Il progetto è stato strutturato in diversi package, partendo dal codice fornito dal professore:

- ast: contiene le classi che rappresentano i nodi dell'albero visitato con il Visitor di *Antlr*. L'interfaccia Node rappresenta il concetto di nodo dell'albero astratto sintattico, in particolare dichiara tre metodi: *checkSemantics*, *typeCheck*, *codeGeneration*. Le classi usate per i vari tipi di nodo implementano questa interfaccia.
- codeexecution: si occupa della esecuzione del codice e della gestione della memoria. In particolare la classe *VirtualMachine* interpreta il codice (SVM) generato dalle istruzioni in FOOL.
- lib: contiene la libreria *Antlr 4.6*, necessaria per l'esecuzione del compilatore.
- parser: contiene *FOOL.g4* e le diverse classi create da *Antlr*.
- svm: contiene *SVM.g4* e le diverse classi create da *Antlr*.
- type: contiene le classi che rappresentano i tipi dei costrutti FOOL: *IntType*, *BoolType*, *ClassType*, *VoidType*.
- util: contiene in particolare la classe *Enviroment* che offre varie strutture dati di utilità e la *Symbol Table*, e la classe *FOOLlib* che oltre ad alcuni metodi utili per la generazione del codice, offre il metodo isSubType per il controllo dei tipi delle espressioni FOOL.

Il file "CompilerLauncher.java" è il punto d'entrata del compilatore.

2.2 Istruzioni Per l'Utilizzo

1. Importare il progetto in *Eclipse* oppure *IntelliJ*.

2. Se il progetto presenta problemi, aggiungere nel *Build Path* la libreria *Antlr* contenuta nella cartella *lib*. Da Eclipse, tasto destro sulla cartella del progetto, *Configure Build Path* e da lì aggiungere la libreria.

3. Il file "prova.fool" nella cartella *Tests* contiene il codice FOOL che il compilatore utilizza. Scrivere lì il proprio codice. Il compilatore legge il file "prova.fool" e, in seguito ai controlli di errori, crea il file "prova.fool.asm" con il codice SVM corrispondente.

4. In caso di errori in un qualunque punto del processo (errori lessicali, sintattici, di scope, di tipi), l'esecuzione verrà interrotta e verrà mostrato l'errore nel terminale.

5. Se nel codice FOOL vengono utilizzati comandi *print*, viene mostrata l'espressione da stampare nel terminale.

3 Sintassi

Punto di inizio dello sviluppo. Si discute FOOL. Si discute l'analisi sintattica, l'AST etc. Anche qui sarà breve.
Si discute anche l'implementazione dell'AST e le altre parti inerenti alla sintassi.

3.1 FOOL

4 Semantica

L'analisi semantica si occupa di controllo di scope e controllo di tipi, questo viene fatto mediante i metodi *checkSemantics* e *typeCheck*, ereditati dall'interfaccia *Node* da tutte le classi che rappresentano i nodi dell'AST.

4.1 Scope

Questo primo controllo, tramite *checkSemantics*, serve per accertarsi del corretto utilizzo di variabili, funzioni e classi. Ad esempio, se una variabile viene utilizzata ma non è mai stata dichiarata, l'errore viene incontrato in questa fase dell'esecuzione del compilatore.

Dopo aver visitato l'albero di sintassi e instanziati i nodi appropriati, viene invocato *checkSemantics* dal punto di entrata dell'albero (la radice), che sarà un nodo "*Prog*" (ProgExpNode, ProgLetInNode o ProgClassNode). Esso invocherà lo stesso metodo sul nodo figlio, il quale continuerà ad invocare sul proprio figlio e così via, fino alle foglie dell'albero.

In particolare, i controlli vengono fatti mediante una "*Symbol Table*"

4.2 Tipi

5 Generazione del Codice

Capitolo sulla code generation. Si discute come si trasforma il codice scritto nella grammatica ad alto livello di FOOL a svm. Come si gestisce lo stack, i pointers, i frames etc etc.

5.1 Heap

5.2 Dispatch Table

5.3 SVM

5.4 Virtual Machine

6 Conclusioni

Brevissimo capitolo conclusivo, si fa un piccolo riassunto di tutto il report e si tirano le somme.