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

30 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, in particolare *scope* e *type checking*. Infine, nel capitolo 5, la generazione del codice assembly e la sua esecuzione.

[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

3.1 Definizione della Sintassi

Le regole sintattiche che definiscono ciò che costituisce un programma scritto "correttamente" si trovano nel file di grammatica di *FOOL.g4* nel package parser.

La serie di regole grammaticali di cui *FOOL.g4* è composto fungono da sostituzioni. Essi associano simboli non-terminali sia aa terminali (stringhe letterali), sia ad altri non-terminali, che a loro volta vengono associati ad altre sostituzioni, per rendere possibile, infine, l'associazione di un programma in FOOL a una combinazione legale di regole grammaticali che definiscono il linguaggio. È questo il compito del *lexer* e del *parser*, che vengono discussi successivamente, cioè della rappresentazione di una serie di stringhe letterali in regole grammaticali opportune.

3.2 Caratteristiche Grammaticali

La potenza espressiva della grammatica è fornita in gran parte dalla notazione sintattica accettata da *ANTLR*. Essa supporta quella usata nella scrittura di espressioni regolari, le regex, che utilizza i simboli per quantificare le istanze di un'espressione indicata, quali " ? ", " * ", " + " e " | ".

Questa espressività si è vista anche nell'abilità di allocare regole di precedenza alle espressioni grammatiche desiderate. Il caso esemplare in FOOL è la precedenza data alle operazioni di divisone e moltiplicazione sopra quelle di addizione e sottrazione tramite la distinzione tra i "factor" e i "term".

```
exp         : left=term ((PLUS | MINUS) right=exp)?
            ;
term        : left=factor ((TIMES | DIVISION) right=term)?
            ;
factor      : left=value ((EQ|GREATER|LESS|GREATEREQUAL|LESSEQUAL|OR|AND) right=factor)?
            ;
```

Inoltre, durante il corso dello sviluppo di questo progetto FOOL è stato reso ancor più espressivo con l'aggiunta di classi, di statements e della possibilità di utilizzare espressioni multiple nel corpo di funzioni e dei let in (anche mischiando statements ed exps).

Nella progettazione di queste capacità aggiuntive, decisioni riguardo le loro convenzioni sintattiche legali sono state prese tenendo conto di un bilancio tra espressività e l'attuabilità dell'implementazione. Un esempio di questo è il cambiamento alla sintassi delle dichiarazioni di funzioni, che adesso richiede i propri incapsulamento tra parentesi graffe. Questo è stato giudicato necessario per eliminare ogni ambiguità nei limiti delle funzioni.

Similarmente, la dichiarazione dei campi e delle funzioni nelle classi vanno scritti tra parentesi e parentesi graffe, rispettivamente. Se non vi sono dichiarati nessun campo o funzione, le parentesi tra cui sarebbero stati scritti non vanno scritte. Così rimuove il bisogno di scrivere parentesi vuote, influenzando la leggibilità.

Un aspetto importanto sviluppato nella grammatica di FOOL è l'introduzione delle statement, ovvero assegnamenti a variabili già esistenti, e l'abilità di utilizzarle liberamente con le espressioni sia nelle funzioni che nel corpo del programma. L'aggiunta di tale capacità era considerata del tutto essenziale al linguaggio FOOL, permettendo come minimo un funzionamento di base di manipolazione di variabili.

Permettere un utilizzo libero di diversi tipi di istruzioni, di fatto, causava delle difficoltà nella sua implementazione. Un esempio è l'ambiguità che creava nella localizzazione del valore di ritorno di una funzione, essendo quello ritornato dalla sua ultima riga di codice. Dato il modo in cui *Antlr* facilita l"accesso alle righe del codice FOOL, raggruppandole secondo le loro categorie (in questo caso "espressioni" o "statements") per determinare se l'ultima istruzione fosse un'espressione oppure una statement era facile nel caso una funzione potesse avere soltanto o espressioni oppure statements. Potento usare in modo libero un qualsiasi numero di espressioni e statements, invece, ha richiesto un approccio diverso, specialmente per determinare il valore di ritorno. Considerando i due tipi di comandi come un unica sequenza di istruzioni (Node) ha permesso di poter trascurare le due categorie e piuttosto controllare il tipo dell'ultima istruzione nella sequenza.

3.3 Controllo della Sintassi

Il codice fornito in input deve aderire alla sintassi definita. L'obiettivo di *Antlr* è quello di controllare che il programma sia corretto sintatticamente, in modo tale da avere un programma adeguato per la verifica semantica successiva. Per raggiungere tale scopo, si rappresenta il programma fornito come un albero AST (*abstract syntax tree*), il quale fungerà da rappresentazione strutturata del programma per facilitare la sua analisi.

Il primo passo in questo processo è la conversione del programma da formato testo in un stream dei *tokens*, compiuto dal *lexer*. Il flusso consiste di valori che corrispondono a termini categorizzati del linguaggio FOOL quali identificatori, interi o caratteri grammaticali. Questa rappresentazione del programma permette più facilmente al *parser* di compiere il suo compito, ovvero quello di applicare le regole grammaticali definite nel file *.g4* al flusso. Trasformando i *token* del flusso, cioè i terminali della grammatica FOOL,

in non-terminali che sono associati a tali *token*, vengono annotati creando una struttura dati di contesti grammaticali. Ciascuno di questi è un nodo nella struttura.

La radice di questo albero di contesti viene passato al *FOOLNodeVisitor* che visita ognuno dei suoi contenuti. Accompagnato ad ogni visita è la generazione di uno o più nodi del AST, che infine costituiranno l'AST finale, pronto per l'analisi semantica.

Da notare che questo processo fallisce in caso di input sintatticamente errato. Programmi che non aderiscono alla grammatica definita causeranno eccezioni della fase del parser, che fallirà ad associare con successo le regole ai suoi token.

4 Semantica

L'analisi semantica si occupa del controllo dello scope e del controllo dei tipi, che vengono eseguiti 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*, cioè una lista di hash tables nella classe Environment. Una hash table rappresenta uno scope nel programma, nel quale vengono inseriti gli identificatori di variabili, funzioni e classi associati alle proprie *STEntry*. I vari scope sono indicizzati tramite l'intero *nesting level*, che parte da 0 quando la Symbol Table è vuota ed incrementa di 1 ad ogni inserimento di hash table (quando si entra in nuovo scope). Gli oggetti STEntry contengono il nesting level, il tipo e l'*offset* dell'identificatore inserito. Con il nesting level si può accedere all'hash table dove l'identificatore è stato inserito, quindi lo scope in cui è stato dichiarato. Il tipo e l'offset sono usati per il controllo dei tipi e per la generazione di codice.

Se una stessa variabile, funzione o classe viene dichiarata più di una volta, l'errore viene riscontrato durante l'inserimento nella hashtable dello scope corrente in quanto già è presente una *entry* con lo stesso identificatore.

Un altro errore che il controllo con la Symbol Table riesce a catturare è l'utilizzo di identificatori non ancora dichiarati (esempio: invocazione di funzione non definita). Per controllare se l'identificatore è stato dichiarato, si cerca nella Symbol Table la sua entry partendo dallo scope più interno (l'ultimo nesting level). Non trovando nessuna entry significa un utilizzo errato dell'identificatore.

4.2 Tipi

Il controllo successivo che il compilatore esegue è quello sui tipi dei costrutti utilizzati nel programma di input. Come *checkSemantics*, il metodo *typeCheck* viene invocato sulla radice dell'AST così da avere una catena di invocazioni che arriva fino alle foglie, per poi risalire di nuovo alla radice la quale darà il risultato finale, cioè il tipo del programma. I tipi che ogni costrutto e il programma stesso possono assumere sono cinque.

- IntType
- BoolType
- ClassType
- VoidType
- ArrowType
- ErrorType

Ognuno implementa la classe astratta Type, che estende Node. In questo modo i tipi sono ancora nodi dell'AST però vengono separati dagli altri in quanto il loro scopo è solo quello di "etichettare" gli altri nodi.

ErrorType è un tipo speciale utilizzato nel caso ci siano errori in un qualunque punto durante la fase di controllo. Nel momento in cui si incontra un errore, un ErrorType viene creato e viene restituito al nodo chiamante di volta in volta fino alla radice. In questo modo come risultato del type checking si ha un ErrorType, il compilatore arresta l'esecuzione e mostra l'errore all'utente.

ArrowType, invece, è una composizione di tipi. È il tipo associato alle funzioni, in quanto una funzione ha un tipo composto dalla sequenza di tipi data dagli argomenti di input e dal tipo di ritorno che il valore da restituire deve avere.

Il punto fondamentale del controllo dei tipi del compilatore è il metodo *isSubType* nella classe FOOLlib. Questo metodo riceve in input due oggeti Type e controlla se il primo è sottotipo dell'altro. Le diverse regole di inferenza (che non riportiamo qui per brevità) sono quelle mostrate a lezione.

Poichè isSubType si occupa unicamente di controllare il sotto tipaggio di due nodi, ogni nodo nel proprio metodo typeCheck deve utilizzare isSubType per la verifica del soddisfacimento della regola di inferenza di interesse. Ad esempio il nodo che rappresenta una funzione (FunNode) dovrà controllare individualmente i parametri e le istruzioni nel corpo, ponendo particolarmente attenzione alla sua ultima istruzione verificando che sia un sotto tipo del suo tipo di ritorno. Oppure il nodo che rappresenta un costrutto *if then else* dovrà controllare se la condizione è un BoolType e se i nodi delle branch *then* ed *else* siano uno sottotipo dell'altro o comunque abbiano un tipo padre in comune.

4.2.1 isSubType

I tipi sono stati divisi in due classi: i tipi "primitivi", che non possono essere ereditati, di cui fanno parte IntType, BoolType, VoidType ed ErrorType, e i tipi ereditabili, i restanti. Questo significa che il metodo isSubType controlla se i tipi ricevuti in input siano parte dell'insieme dei tipi primitivi ed in tal caso controlla solo se siano lo stesso tipo, (nel caso del tipo speciale ErrorType, si restituisce l'errore prima ancora delle invocazioni di isSubType). Nel caso in cui sono passati in input due ClassType, si controlla se il nome è lo stesso (che sta a significare sono la stessa classe), in caso negativo si ottiene il ClassNode corrispondente alla prima classe per verificare se esso contiene il nome della classe padre. In tal caso si verifica se la classe padre è la seconda classe data in input. Si continua a risalire l'albero dell'ereditarietà in questo modo fino a trovare la classe estesa.

Per gli ArrowType, invece, il controllo del sotto tipaggio viene effettuato dal ClassNode stesso sui propri metodi e quelli della sua classe padre. Quindi isSubType non implemente di per se la verifica di sotto tipo dei metodi, ma le ClassNode invocano isSubType individualmente su tutti i paramentri e sul tipo di ritorno (come definito dalla regola di inferenza sull'overriding dei metodi).

Per approfondire sulle regole di inferenza utilizzate (sulle espressioni semplici, subtyping di classi, overriding di metodi etc.) consultare le slides sul type checking.

5 Generazione del Codice

L'ultima fase del compilatore è quella di generare codice assembly da eseguire tramite una Stack Virtual Machine (SVM). Ogni nodo definisce un metodo chiamato *codeGeneration* che si occupa di tradurre ogni nodo in codice eseguibile restituendo un stringa con il codice riguardante il costrutto che il nodo rappresenta. Ogni stringa generata dalle varie invocazioni del metodo vengono concatenate così da avere infine il codice completo, pronto per la SVM.

5.1 Codice assembly

Come per il codice FOOL, anche il codice assembly è stato definito in un file .g4 per *Antlr*, chiamato SVM.g4, nel package svm. La grammatica di questo codice definisce istruzioni che lavorano su uno stack, dove le istruzioni fondamentali alla base del funzionamento di tutte le altre sono *push* e *pop*. Analogamente a FOOL, il linguaggio assembly utilizzato è stato modificato ed esteso da un linguaggio pre-esistente dato dal professore. In particolare sono stati rimossi gli attributi ed azioni (vedere documentazione di *Antlr*), ed usato il *visitor* pattern per visitare il suo AST e definire il codice Java da eseguire per ogni nodo. In più, alcune istruzioni sono state aggiunte a quelle già definite: *new* (instanziazione oggetto), *lm* (load method), *cts* (copy top of stack, duplica la cima dello stack facendo un *pop* e due *push* dello stesso valore) e le istruzioni per eseguire i confronti numerici e booleani.

5.1.1 Virtual Machine

Il pattern visitor è stato implementato per generare oggetti AssemblyNode, dove ognuno di essi rappresenta un singolo comando assembly. Vengono inseriti in una lista, uno ad uno come vengono creati, così da rispecchiare l'ordine dei comandi dell'apposito file assembly. Un oggetto AssemblyNode contiene l'etichetta del comando che rappresenta, l'indice nel codice (la posizione in cui si trova) ed eventualmente i valori numerici o la label associata in base al tipo di comando.

Per eseguire questo codice (nella classe VirtualMachine) la lista viene "attraversata" prendendo uno ad uno gli oggetti AssemblyNode generati, controllando di che comando si tratti tramite l'etichetta ed eseguendo le operazioni opportune utilizzando i vari valori che AssemblyNode contiene, se necessario. L'esecuzione termina quando si raggiunge il

nodo con l'etichetta "HALT", non necessariamente l'ultimo nodo (funzioni e classi sono dichiarate dopo e raggiunte tramite labels, quindi appaiono dopo anche nella lista).

5.1.2 Istanziazione

Quando si crea una nuova instanza con il costruttore "new ClassName()", il codice generato (dal nodo ConstructorNode) corrisponde al push dei campi, al push del **numero** dei campi e all'istruzione *new*. Quest'ultima istruzione si occupa di instanziare l'oggetto, cioè di inserire nell'**heap** l'indirizzo della dispatch table (vedi dopo) e i campi. Presume che in cima allo stack ci sia l'indirizzo della dispatch table, il numero di campi e i parametri dati in input al costruttore, condizione che è sempre vera poichè ConstructorNode restituisce un blocco di codice che comprende tutte le istruzioni richieste da *new*.

5.1.3 Indirizzo dei metodi

Istruzione usata per il dispatch dinamico, carica in cima allo stack l'indirizzo di un metodo, ottenuto dalla dispatch table della sua classe. Presume che in cima allo stack ci sia l'indirizzo al metodo nella dispatch table, in questo modo può andare a prendere l'indirizzo alla label che indica l'inizio del corpo del metodo e inserirlo nello stack.

5.2 Heap

La memoria della virtual machine è gestita come uno stack, ma implementata in Java è, in realtà, un array di interi chiamato *memory*. Le istanze delle classi, invece, sono gestite con un heap. Questo si traduce nell'usare lo stesso array di interi però usando le prime posizioni dell'array (inoltre se l'ultima posizione dell'heap si incrocia con il puntatore della cima dello stack, significa che la memoria è esaurita).

In particolare lo heap è costituito dalla classe Heap e dalle sue "entry": HeapBlock. Con queste classi si gestiscono le allocazioni delle celle dell'array *memory*. Il funzionamento della classe Heap è quello di una *linked list* di oggetti HeapBlock, avendo sempre come testa della lista il prossimo HeapBlock libero. Un HeapBlock contiene la posizione dell'array *memory* che rappresenta, quindi allocare un segmento dell'array *memory* vuol dire prendere il prossimo HeapBlock libero (prossima posizione della memoria) e andare avanti di un elemento (quindi scorrere la lista) tante volte quante sono le posizioni dal allocare, aggiornando la testa della lista alla posizione seguente dell'ultima allocata.

5.3 Dispatch Table

Per gestire l'ereditarietà con overriding dei metodi è stata utilizzata la tecnica della *dispatch table* (definita nella classe DispatchTable), dove ad ogni classe viene associata la lista dei metodi della classe, con gli indirizzi in memoria che puntano al corpo dei metodi. Seguendo l'idea della Symbol Table di avere come entry una classe apposita, la dispatch table è popolata da oggetti DTEntry.

Una DTEntry è un contenitore per l'identificatore del metodo (il nome) e la sua label nel codice per SVM, la label indica l'inizio del codice del corpo del metodo nel file assembly. Perciò, per accedere ad un certo metodo si utilizza l'offset ottenuto durante le fasi precedenti dell'analisi semantica, il quale funziona da indice sulla lista dei metodi.

Per ogni classe, si aggiunge la sua dispatch table all'insieme di tutte le dispatch table, in modo tale da averle raggruppate in un unico punto per poi inserirle una dopo l'altra nel codice SVM. Al momento della generazione del codice, in sequenza vengono "trasformate" in stringhe della forma "nomeClasse_class: labelMetodo1 labelMetodo2" etc. Per accedere ad un certo metodo, in cima allo stack dovrà esserci l'indirizzo alla label "nomeClasse_class:" e ad esso viene aggiunto l'offset per accedere al metodo corretto. Ad esempio per accedere a "labelMetodo1" l'offset sarà 1, per "labelMetodo2" l'offset sarà 2. Ottenuto l'indirizzo corretto per un metodo specifico, l'istruzione successiva dovrà essere *lm* che si occuperà di inserire nello stack l'indirizzo alla label che delimita l'inizio del corpo del metodo sudetto, utilizzando la label del metodo della dispatch table.

5.4 Istruzioni Multiple

Come già detto, il compilatore sviluppato permette l'utilizzo di expressions e statements multipli nel corpo di una funzione o nella clausola "in" dei *let in*. Questa è stata una scelta di design degli autori per creare un linguaggio più espressivo e più simile ai moderni linguaggi di programmazione. Per aggiungere questa nuova caratteristica sono state affrontate alcune sfide sopratutto nella fase di generazione del codice.

Un problema in particolare è stato il gestire invocazioni multiple di funzioni o metodi, in quanto invocare funzioni più di una volta "sporcava" lo stack lasciando valori non più utilizzati che rendeno sbagliati i successivi utilizzi di offset. Questo ha richiesto una gestione dei comandi *pop* dinamica in base al numero di invocazioni in una funzione e ad un utilizzo più attento del comando *srv* per restituire (lasciare in seguito nello stack) il valore di ritorno corretto.