FCompiler: A FOOL implementation

Giulia Cantini
Matteo Del Vecchio
Simone Preite

20 Luglio 2018

Indice

1 Introduzione

Il progetto consiste nell'implementazione di un compilatore per il linguaggio FOOL (Functional Object Oriented Language) che traduce in bytecode eseguibile per la SVM (Stack Virtual Machine).

bile da una stack machine SVM (Simplified Virtual Machine).

Di seguito si riportano le specifiche fornite per il progetto.

1.1 Specifiche fornite

1.1.1 Espressioni

Estensione delle espressioni con operatori $<=, >=, ||, \&\&, /, -$ e not.

1.1.2 Comandi

Esiste la categoria sintattica *stm*:

stm : id = exp (assegnamento) ;
| if exp then stms else stms (condizionale)

stms : (stm)+

1.1.3 Object orientation

È possibile dichiarare classi e sottoclassi al top level.
Le classi contengono

- campi (dichiarati nella classe o ereditati dalla super-classe)
- metodi (esplicitamente dichiarati nella classe o ereditati dalla super-classe).

Se in una sottoclasse viene dichiarato un metodo con il medesimo nome di un metodo della super-classe, tale metodo sovrascrive quello della super-classe. Non è possibile avere overriding di campi.
Quindi, significa che occorre estendere

dichiarazioni

class ID [extends ID] (...campi dichiarati come parametri...)

... metodi dichiarati come funzioni ...

espressioni

x.m(..) // x e‘ un oggetto, m e‘ un metodo

new C(..) // C e‘ un identificatore di classe

null

comandi con x.m(..), se il metodo ritorna void.
In particolare

- Gli oggetti nascono come istanza di classi
- I campi sono modificabili

- Il linguaggio ha il comando di assegnamento
- Non è possibile dichiarare funzioni annidate. Le funzioni non possono essere passate come parametri
- È possibile avere funzioni/metodi mutuamente ricorsivi.

1.1.4 Type check

Il compilatore deve comprendere un type-checker che controlli il corretto uso dei tipi. In particolare:

- Esiste il tipo void, oltre al tipo classe
- Si deve considerare una nozione di subtyping fra classi e tipi di funzioni. Il tipo di una funzione f1 è sottotipo del tipo di una funzione f2 se il tipo ritornato da f1 è sottotipo del tipo ritornato da f2, se hanno il medesimo numero di parametri, e se ogni tipo di parametro di f1 è sopratipo del corrisponde tipo di parametro di f2.
  Una classe C1 è sottotipo di una classe C2 se C1 estende C2 e se i campi e metodi che vengono sovrascritti sono sottotipi rispetto ai campi e metodi corrispondenti di C2. Inoltre, C1 è sottotipo di C2 se esiste una classe C3 sottotipo di C2 di cui C1 è sottotipo.
- Definire e implementare le regole di tipaggio per tutti i costrutti, in particolare per il condizionale.

1.1.5 Il codice oggetto

Il compilatore deve generare codice per un esecutore virtuale chiamato SVM (stack virtual machine) la cui sintassi è definita nel file SVM.g. Tale esecutore ha una memoria in cui gli indirizzi alti sono usati per uno stack. Uno stack pointer punta alla locazione successiva alla prossima locazione libera per lo stack (se la memoria ha indirizzi da 0 a MEMSIZE-1, lo stack pointer inizialmente punta a MEMSIZE). In questo modo, quando lo stack non è vuoto, lo stack pointer punta al top dello stack.

Il programma è collocato in una memoria separata puntata dall'instruction pointer (che punta alla prossima istruzione da eseguire). Gli altri registri della macchina virtuale sono: HP (heap pointer), RA (return address), RV (return value) e FP (frame pointer).
In particolare, HP serve per puntare alla prossima locazione disponibile dello heap; assumendo di usare gli indirizzi bassi per lo heap, HP contiene inizialmente il valore 0.

- Implementare tutte le operazioni per le nuove espressioni
- Implementare il dynamic dispatch ==> implementare e definire layout oggetti e classi

- La generazione di codice oggetto è fatta utilizzando una grammatica con attributi, vedere SVM.g4). Definire una grammatica SENZA attributi (come FOOL.g4) e definire una visita dell'albero sintattico per generare il codice.

1.1.6 Opzionali

La deallocazione degli oggetti nello heap (garbage collection) non è obbligatoria.

2 Struttura del progetto

Il codice sorgente del progetto è contenuto nella sottodirectory source/ che è così divisa:

- **ast/** contiene i nodi dell'albero di sintassi astratta generati dal parser
  - **ast/types/** in particolare, contiene quei nodi che rappresentano dei tipi all'interno della grammatica
- **grammars/** contiene i file .g4 per le grammatiche FOOL e SVM con i rispettivi parser e le implementazioni dei visitor per la visita dell'ast e del bytecode
- **lib/** contiene i jar delle libreria di ANTLRv4, di commons-cli per eseguire il progetto da riga di comando e la libreria YAML di supporto al testing
- **testMain/** contiene le classi principali per l'esecuzione del progetto, ovvero le due configurazioni di run, Main.java per l'esecuzione standard e TestSuite.java per l'esecuzione dei test contenuti nel file testSuite.yml in formato YAML e il CommandLineMain per l'esecuzione da riga di comando.
- **utils/** contiene le classi di supporto per la symbol table, il type checking e altre funzioni ausiliarie.

Il progetto è stato sviluppato usando IntelliJ IDEA, si consulti il file README.md per le istruzioni su come aprire ed eseguire il progetto con il suddetto IDE oppure utilizzando Eclipse o il terminale.

3 Caratteristiche del linguaggio e grammatica

Un tipico programma scritto in linguaggio FOOL è formato da un insieme di blocchi: ogni blocco può essere una dichiarazione di classe (al top level) oppure una sequenza di dichiarazioni introdotte dalla keyword *let*, seguite da un'altra sequenza di statement in cui le variabili dichiarate vengono usate, introdotta dalla keyword *in*.

```
class A (int x = 1, int y = 2) {
  void m() print(0);
}
```

```
class
```

Le espressioni sono state estese a comprendere gli operatori $<=, >=, ||, \&\&, /, -$ e not nella seguente maniera:

```
exp    : left=operand (operator=(EQ | LEQ | GEQ ) right=exp)? ;

operand:  left=term (operator=(PLUS | MINUS) right=operand)? ;

term   : left=factor (operator=(TIMES | DIV) right=term)? ;

factor : left=atom (operator=( OR | AND) right=factor)? ;

atom : (NOT)? (MINUS)? val=value ;

value  :
         INTEGER                                    #intVal
       | BOOLVAL                                    #boolVal
       | NULL                                       #nullVal
       | LPAR exp RPAR                              #baseExp
       | IF LPAR cond=exp RPAR THEN CLPAR thenBranch=exp CRPAR ELSE
    CLPAR elseBranch=exp CRPAR  #ifExp
       | var      #varExp
       | ID (LPAR (exp (COMMA exp)* )? RPAR )          #funExp
       | object=var DOT memberName=ID ( LPAR (exp (COMMA exp)* )?
    RPAR )?  #methodExp
       | NEW className=ID (LPAR RPAR)          #newExp
       ;
```

Sono stati aggiunti gli statement

```
stm : (var | object=var DOT fieldName=ID) ASM exp #varStmAssignment
  | IF cond=exp THEN CLPAR thenBranch=stms CRPAR (ELSE CLPAR
    elseBranch=stms CRPAR)?  #ifStm
  | object=var DOT memberName=ID ( LPAR (exp (COMMA exp)* )? RPAR )
    ?  #methodStm
  | PRINT LPAR exp (COMMA exp)* RPAR  #printStm
  | ID (LPAR (exp (COMMA exp)* )? RPAR )          #funStm
  ;

stms : ( stm )+  ;
```

uno statement può essere un assegnamento (semplice o a campo di un oggetto), un condizionale, una print o una chiamata di funzione/metodo.

Una dichiarazione di classe viene fatta nel modo seguente:

```
class hero {
    bool iAmBatman() return true;
};

class super extends hero(int x = 0, bool cond = true) {
    int fun() return 1;
    int soMuchFun(int a, bool b) return 1000;
};
```

dove le parentesi tonde delimitano i campi e le parentesi graffe i metodi. Una classe può contenere nessun campo ma deve contenere sempre almeno un metodo. Una classe puo' estenderne un'altra (di cui diventera' una sottoclasse) ereditandone campi e metodi, utilizzando la keyword *extends*.

```
classdec : CLASS className=ID ( EXTENDS superName=ID )? (LPAR
    varasm (COMMA varasm)* RPAR)? CLPAR (fundec SEMIC)+ CRPAR ;
```

Un oggetto può essere istanziato utilizzando la keyword *new*. Il costruttore deve essere richiamato senza argomenti, mentre i valori dei campi sono stati già inizializzati durante la dichiarazione della classe e potranno essere modificati successivamente tramite assegnamento.

Ulteriori caratteristiche del linguaggio sono le seguenti:

- Tutte le variabili devono essere inizializzate al momento della dichiarazione (così come vale per i campi di una classe).
- Una variabile può avere tipo intero, booleano o classe (se è un oggetto).
- Una funzione può avere tipo di ritorno void
- È stato inserito il tipo null che è sottotipo di classe, per consentirne l'assegnamento ad un oggetto.
- Una funzione può opzionalmente avere un blocco *let in* all'interno del suo body, che è formato da statement oppure statement ed eventualmente un'espressione preceduta dalla keyword *return* nel caso in cui il suo tipo di ritorno sia diverso da void.

4 Analisi lessicale e sintattica (AST)

L'interfaccia Node, implementata da tutti i nodi dell'ast è stata modificata nel seguente modo:

- il metodo **checkSemantics()** ha come tipo di ritorno HashSet<String> invece di ArrayList<Semantic Error>; l'oggetto di tipo String rappresenta la stringa che contiene l'errore semantico e si è optato per una gestione degli errori con HashSet per facilitare il controllo sull'inserimento di duplicati.
- il metodo **typeCheck()** adesso solleva un'eccezione quando viene rilevato un errore di tipo (vedi file source/utils/TypeCheckException.java).
- è stato aggiunto il metodo **getID()** che restituisce l'identificatore di un oggetto sotto forma di stringa.

4.1 Nodi delle funzioni

- **FunDecNode** rappresenta la dichiarazione di una funzione, in esso sono memorizzati il nome della funzione, il suo tipo di ritorno, la lista dei parametri, la lista delle dichiarazioni e il corpo della funzione.
  Inoltre il nodo memorizza l'entry della symbol table associata alla funzione stessa.
- **FunExpNode** rappresenta la chiamata di una funzione, in esso sono memorizzati l'id della funzione, gli argomenti, il nesting level e una variabile di supporto classID per gestire la chiamata di metodo che avviene senza dot notation. Inoltre il nodo memorizza l'entry della symbol table associata alla funzione stessa.

4.2 Nodi per la gestione delle classi

- **BlockClassDecNode** rappresenta la dichiarazione di una classe, e contiene oltre ai metodi implementati dell' interfaccia Node, le utility getMethods() e getFields()
- **ClassMethodNode** rappresenta la chiamata di metodo (o.m()), è sottoclasse di FunExpNode
- **ClassFieldNode** rappresenta il campo di una classe
- **MethodDecNode** rappresenta la dichiarazione di metodo
- **ClassType** rappresenta il tipo classe, memorizza l'id, l'id del supertipo, la lista dei campi e dei metodi. Contiene getter e setter per gestire questi campi.

5 Analisi semantica

5.1 Symbol table

La costruzione della symbol table avviene visitando l'ast e richiamando su un ogni nodo il metodo **checkSemantics()**, che a partire dal primo nodo ProgNode (che rappresenta la start rule della grammatica) viene richiamata due volte. Le due visite successive servono per la gestione delle dichiarazioni di classi e di funzioni/metodi. Nel caso delle dichiarazioni di classi, la prima passata

viene utilizzata per raccogliere tutte le dichiarazioni, mentre la seconda serve per gestire campi, metodi, oggetti e risolvere le relazioni di sottotipaggio tra classi. Nel caso di funzioni/metodi, la doppia visita consente di implementare il meccanismo di mutua ricorsione, ovvero consente che nel corpo di una funzione possa essere chiamata un'altra funzione non ancora definita. Lo stesso vale per i metodi.

La tabella dei simboli viene gestita utilizzando la classe Environment (contenuta nella sottodirectory utils) e i metodi di supporto che contiene.
La classe contiene i seguenti campi:

```
private ArrayList<HashMap<String, SymbolTableEntry>> symTable;
private int nestingLevel;
private int offset;
private boolean secondCheck; // questa e la successiva gestiscono
    le passate della checkSemantic
private boolean secondFunCheck;
private String definingClass = null; // indica il nome della classe
     da cui stiamo aggiornando la st
```

In più contiene (oltre a getter e setter per i campi) anche i metodi:

- **public int decreaseOffset()**
- **public int increaseOffset()**
- **public SymbolTableEntry getActiveDec(String id)** scorre la symbol table alla ricerca della entry corrispondente alla dichiarazione al momento attiva nell'ambiente per l'oggetto rappresentato da id.
- **public void pushScope()** aggiunge una nuova tabella hash alla lista che rappresenta la symbol table, con nesting level incrementato.
- **public void popScope()** rimuove dalla symbol table la hash corrispondente al nesting level corrente, e poi lo decrementa.
- **public SymbolTableEntry getClassEntry(String classID)** cerca nella symbol table la entry della classe rappresentata da classID

La tabella dei simboli viene riempita utilizzando la classe SymbolTableEntry che possiede i seguenti campi:

```
private int nestingLevel;
private int offset;
private Node type;  // nel caso degli oggetti, indica il tipo
    dinamico
private String className; // usata nel caso di entry che
    rappresentano classi
private Node staticType; // usata nel caso di classi
```

5.2 Type check

I tipi su cui si basa il sistema di tipaggio di FOOL sono i seguenti:

- BoolType
- ClassType
- FunType
- IntType
- NullType
- VoidType

6 Generazione del codice

6.1 Gestione degli offset, stack, heap

6.2 Layout di un oggetto

6.3 Grammatica SVM senza attributi e SVMVisitor

6.4 Dynamic dispatch

6.5 Garbage collector (???)

7 Conclusioni