

# FCompiler: A FOOL implementation

Giulia Cantini  
Matteo Del Vecchio  
Simone Preite

20 Luglio 2018

## Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Specifiche fornite . . . . .	2
1.1.1	Espressioni . . . . .	2
1.1.2	Comandi . . . . .	2
1.1.3	Object orientation . . . . .	2
1.1.4	Type check . . . . .	3
1.1.5	Il codice oggetto . . . . .	3
1.1.6	Opzionali . . . . .	4
<b>2</b>	<b>Struttura del progetto</b>	<b>4</b>
<b>3</b>	<b>Caratteristiche del linguaggio e grammatica</b>	<b>4</b>
<b>4</b>	<b>Analisi lessicale e sintattica (AST)</b>	<b>6</b>
4.1	Nodi degli operatori . . . . .	7
4.2	Nodi delle funzioni . . . . .	7
4.3	Nodi per la gestione delle classi . . . . .	7
<b>5</b>	<b>Analisi semantica</b>	<b>7</b>
5.1	Symbol table . . . . .	7
5.2	Type check . . . . .	8
<b>6</b>	<b>Generazione del codice</b>	<b>8</b>
6.1	Gestione degli offset, stack, heap . . . . .	8
6.2	Layout di un oggetto . . . . .	8
6.3	Grammatica SVM senza attributi e SVMVisitor . . . . .	8
6.4	Dynamic dispatch . . . . .	8
6.5	Garbage collector (???) . . . . .	8
<b>7</b>	<b>Conclusioni</b>	<b>8</b>

# 1 Introduzione

Il progetto consiste nell'implementazione di un compilatore per un linguaggio FOOL (Functional Object Oriented Language) che traduce in bytecode eseguibile 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  $\leq$ ,  $\geq$ ,  $\|$ ,  $\&\&$ ,  $/$ ,  $-$  e  $\text{not}$ .

### 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
- É possibile avere funzioni/metodi mutuamente ricorsive.

#### 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 (vedi lezioni)

- 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. Chi è interessato può scrivere il modulo relativo.

## 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
- **lib/** contiene la libreria (.jar) di ANTLRv4
- **testMain/** contiene le classi principali delle due rispettive configurazioni di run, Main.java per l'esecuzione standard e TestThread.java per l'esecuzione dei test su file .fool multipli (contenuti nella cartella FCompiler/code)
- **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.

## 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  $\leq$ ,  $\geq$ ,  $\parallel$ ,  $\&\&$ ,  $/$ ,  $-$  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 può estenderne un'altra (di cui diventerà 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>` questo perche' si e' eliminata la classe `Semantic Error`, che rappresentava solo una stringa e si e' 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`).
- e' stato aggiunto il metodo **getID()** che restituisce l'identificatore di un oggetto sotto forma di stringa.

#### 4.1 Nodi degli operatori

#### 4.2 Nodi delle funzioni

- `FunDecNode`
- `FunExpNode`

#### 4.3 Nodi per la gestione delle classi

- `BlockClassDecNode`
- `ClassMethodNode`
- `ClassFieldNode`
- `MethodDecNode`
- `ClassType`

### 5 Analisi semantica

#### 5.1 Symbol table

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; // serve per...
private boolean secondFunCheck; // serve per...
```

In piu' 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
- **public void updateClassEntry(ClassType type)**

## 5.2 Type check

# 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