FoolOO

PROGETTO DI COMPILATORI E INTERPRETI

A.A. 2017/2018

Corso di laurea magistrale in Informatica

Marchesini Matteo
matteo.marchesini12@studio.unibo.it
N° Matricola: 856336

Pascali Andrea
andrea.pascali@studio.unibo.it
N° Matricola: 854835

Sanfelici Matteo
matteo.sanfelici@studio.unibo.it
N° Matricola: 856403

Introduzione

Questo progetto prende il nome di FoolOO, in quanto consiste in un compilatore per un linguaggio Object-Oriented sulla base di un linguaggio chiamato FOOL.

L'intero progetto è stato scritto in Java, con l'aggiunta di una libreria esterna chiamata ANTLR (versione 4.7). Lo sviluppo del linguaggio ha portato all'implementazione di 4 fasi:

- Analisi lessicale
- Analisi sintattica
- Analisi semantica
- Code generation

Nell'ordine, l'analisi lessicale e sintattica hanno visto la creazione nonché scrittura di una grammatica, partendo da quella esistente che ci è stata data dal professore. In questa fase il team ha dovuto aggiungere nuove regole sulla base di quanto richiesto dalle specifiche, in modo da rendere il linguaggio object-oriented.

Scritta la grammatica, con il tool ANTLR v4 è stato auto-generato codice necessario ad eseguire l'analisi lessicale e sintattica e per poter implementare la generazione dell'albero di sintassi astratta (AST). L'albero di sintassi astratta è un insieme di nodi in cui ciascuno identifica una tipologia di produzione possibile. Ognuna di queste produzione implementa l'interfaccia "Node", la quale presenta i metodi di typeCheck, codeGeneration ed un arraylist di errori semantici.

Ciascun metodo sarà poi implementato da ogni produzione secondo la grammatica definita.

Analisi lessicale

Nell'analisi lessicale sono state implementate tutti i token del lexer da utilizzare nella grammatica per ottenere una sintassi completa ed essenziale per un buon linguaggio object-oriented.

In particolare sono stati aggiunti token propri delle caratteristiche object-oriented, quali i termini *'class'* ed *'extends'* propri della definizione di una classe e dell'ereditarietà, e altri token, quali nuovii "type", e gli operatori logici.

Inoltre, il linguaggio FoolOO presenta un main, quindi avrà un token ad esso dedicato per poter definirlo.

I seguenti token vengono riconosciuti e accettati dal linguaggio come termini primitivi.

```
DOT     : '.';
SEMIC   : ';' ;
COLON   : ':' ;
COMMA   : ',' ;
EQ      : '==' ;
ASM     : '=' ;
PLUS    : '+' ;
MINUS   : '-' ;
TIMES   : '*' ;
DIV     : '/' ;
AND     : '&&' ;
OR      : '||' ;
GTEQ    : '>=' ;
LTEQ    : '<=' ;
NOT     : '!' ;
TRUE    : 'true' ;
FALSE   : 'false' ;
LPAR    : '(' ;
RPAR    : ')' ;
```

```
CLPAR   : '{' ;
CRPAR   : '}' ;
QLPAR   : '[' ;
QRPAR   : ']' ;
IF      : 'if' ;
THEN    : 'then' ;
ELSE    : 'else' ;
PRINT   : 'print' ;
LET     : 'let' ;
IN      : 'in' ;
INT     : 'int' ;
BOOL    : 'bool' ;
VOID    : 'void' ;
RETURN  : 'return' ;
CLASS   : 'class';
EXTENDS : 'extends';
NEW     : 'new';
NULL    : 'null';
MAIN    : 'main';
```

Analisi sintattica

L'analisi sintattica ha portato ala creazione di una grammatica ben articolata per poter scrivere correttamente un codice object-oriented.

In questa fase vi sono state numerose scelte progettuali raggiunte con la collaborazione di tutti i membri del team, in quanto la grammatica del linguaggio madre "Fool" è stata ampliata con innumerevoli regole legate sia alla necessità di avere un linguaggio orientato agli oggetti, e sia perché presentava dei limiti nelle funzionalità proposte.

Ciascun programma scritto in FoolOO dovrà presentare un main ed eventualmente delle dichiarazioni di classe. La dichiarazione di classe può presentare o meno l'estensione di un'altra classe, seguita dalla dichiarazione di variabili separate da una virgola. All'interno di una classe può esistere o meno la funzione costruttore e un insieme di funzioni di classe, ovvero i metodi.

```
start      : (decclass)* main ;

decclass   : CLASS ID (EXTENDS ID)? LPAR (vardec(COMMA vardec)*)?
                        RPAR CLPAR (funClass|funconstructor)+ CRPAR
           ;

main       : type MAIN LPAR RPAR CLPAR prog CRPAR ;
```

La funzione **main** presenta una sintassi di questo genere: *void main ( ) { … }.* Il type del main è sempre void. Al suo interno c'è tutto cuore del linguaggio, ovvero **prog**, che ammette *statements* o dichiarazioni all'interno della forma ***let*** *dec in* seguita o meno da statements.

```
prog       : (stms)?              #singleExp
           | let (stms)?          #letInExp
           ;

let        : LET (dec)+ IN
           ;

progFun    : (stms)?              #singleExpFun
           | letFun (stms)?       #letInExpFun
           ;

letFun     : LET (varasm)+ IN
           ;

dec        : varasm               #varAssignment
           | fun                  #funDeclaration
```

Dec a sua volta ammette una assegnamento o una dichiarazione di funzione.

In **varasm** sono possibili due tipi di assegnamento: assegnamento di un espressione oppure di uno statement. Di seguito possiamo osservare produzioni simili tra **funconstructor**, **fun** e **funclass**, utili a differenziare il costruttore da una qualsiasi funzione; inoltre la funclass è necessaria per poter differenziare i metodi di una classe da una funzione dichiarata nel main ( ).

La produzione **type** permette di stabilire quali tipi sono ammessi nel linguaggio. Come osservabile di seguito, è presente il tipo ID che identifica il tipo oggetto (classe).

```
vardec          : type ID;

varasm          : vardec ASM exp SEMIC      #expDecAsignment
                | vardec ASM stm            #stmDecAsignment
                ;

funconstructor  : ID LPAR (vardec(COMMA vardec)*)? RPAR CLPAR progFun
                  CRPAR ;

fun             : type ID LPAR (vardec(COMMA vardec)*)? RPAR CLPAR
                  progFun(ret)? CRPAR;

funClass        : type ID LPAR (vardec(COMMA vardec)*)? RPAR CLPAR
                  progFun(ret)? CRPAR;

ret             : RETURN exp SEMIC            #returnFunExp
                | RETURN stm                  #returnFunStms
                ;

type            : INT
                | BOOL
                | VOID
                | ID
                ;
```

All'interno di una funzione o di un metodo di una classe è possibile definire un ritorno, il quale può identificarsi in un'espressione o in uno statement. É stata definita una produzione propria del **return**.

La produzione **exp** esiste solo nel momento in cui esiste anche la sua parte destra, ovvero un'operazione di somma/differenza con un'altra produzione exp. Il termine sinistro di exp si traduce in **term**, una produzione che si comporta nel medesimo modo di exp, con la differenza che permette un'operazione di moltiplicazione/divisione. Infine il termine sinistro di term si traduce in factor, una produzione che permette di applicare gli operatori logici per eseguire un confronto tra due **value**. La produzione value può assumere diversi valori (vedi tabella), tra cui uno in particolare è

stm. Stm indica uno statement, la cui produzione può identificarsi in un assegnamento di un'altro statement, di un'espressione, in una chiamata di metodo, in una chiamata di funzione, in un assegnamento in base ad una condizione di if e infine nella stampa di un'espressione.

La produzione #callFunExp si traduce in **funExp** (chiamata di funzione), nella quale è necessario specificare l'id della funzione e i parametri.

```
exp     : (MINUS)? left=term ((PLUS|MINUS)right=exp)?
        ;

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

factor  : (NOT)? left=value ((EQ|GTEQ|LTEQ|AND|OR) factorRight)?
        ;

factorRight: (NOT)? right=value;

stm     : IF cond=exp THEN CLPAR
           thenBranch=stms CRPAR
           (ELSE CLPAR elseBranch=stms CRPAR)?   #stmIf
        | PRINT LPAR exp RPAR SEMIC              #stmPrint
        | funExp SEMIC                           #callFunExp
        | ID DOT funExp SEMIC                    #callMethod
        | ID ASM stm                             #stmAssignment
        | ID ASM exp SEMIC                       #stmValAssignment
        ;

funExp: ID LPAR (exp(COMMA exp)*)? RPAR ;

stms    : (stm)+
        ;

value   : INTEGER                                #intVal
        | ( TRUE | FALSE )                       #boolVal
        | LPAR exp RPAR                          #baseExp
        | IF cond=exp THEN CLPAR
           thenBranch=exp CRPAR (ELSE CLPAR
           elseBranch=exp CRPAR)?                #ifExp
        | stm                                    #stmsExp
        | ID                                     #varExp
        | NULL                                   #nullVal
        | NEW ID LPAR (exp (COMMA exp)* )? RPAR  #newClass
        | funExp                                 #funExpValue
        | ID DOT funExp                          #callMethodValue
```

Analisi semantica

L'analisi semantica si compone di due parti: scope checking e type checking.

Lo scope checking si identifica nella costruzione della symbol table, che a livello di codice consiste in un arraylist di hashmap. Ciascun hashmap identifica un livello, il cosiddetto "scope", il quale a seconda del nodo in cui ci troviamo può essere o meno popolato.

Infatti nella symbol table vengono aggiunte nuove entries quando avviene una dichiarazione, che può essere una dichiarazione di classe, di funzione, di un metodo all'interno della classe, o di un oggetto o variabile. Fatto ciò vengono analizzati gli statements, ovvero tutti i punti del codice in cui vengono utilizzate variabili, classi e metodi per verificarne l'esistenza e la correttezza.

In particolare, nel linguaggio FoolOO, sono presenti i seguenti errori semantici:

- "Id is not declared";
- "Method is not declared"
- "Field of class already declared";
- "Parameter of class already declared";
- "Class is already declared";
- "Method of class already declared";
- "Function already declared";
- "Parameter of function already declared";
- "Missing return";
- "Parameter already declared in constructor" ;
- "Function not declared";
- "Multiple declaration of variable x".

Terminata la fase di scope checking l'albero di sintassi astratta viene nuovamente visitato per eseguire la fase di type checking.

In questa fase viene utilizzata la symbol table creata e popolata precedentemente per poter stabilire il tipo di ciascuna espressione, ed eventualmente segnalare un errore in caso di tiraggio errato.

Di seguito tutte le regole di tipaggio utilizzate in FoolOO.

Integer value	$\Gamma \vdash e : int[intVal]$
Boolean value	$\Gamma \vdash false : bool[boolVal] \quad \Gamma \vdash true : bool[boolVal]$
Void value	$\Gamma \vdash e : void[voidVal]$
Object value	$\Gamma \vdash e : id[objectVal]$
Null value	$\Gamma \vdash x : null[nullVal]$
Variables	$\Gamma[x \mapsto T] \vdash x : T[varExp]$
Print	$\dfrac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{print}(e) : T}[stmPrint]$
If-Then-Else	$\dfrac{\Gamma \vdash cond : T \quad \Gamma \vdash e1 : T1 \quad T = bool \quad \Gamma \vdash e1 : T2 \quad T1 \prec T' \quad T2 \prec T'}{\Gamma \vdash \ \mathsf{if}\ cond\ \mathsf{then}\ e1\ \mathsf{else}\ e2\ : T'}$
Call method	$\dfrac{\Gamma(x) = C \quad \Gamma(C)(m) = T_1 \times \ldots \times T_n \mapsto T \quad (\Gamma \vdash e_i : T_i' \quad T_i' \prec T_i)^{i \in 0,\ldots,n}}{\Gamma \vdash \ x.m(e_1, \ldots, e_n) \ : \ T}[CallMethod]$
New class	$\dfrac{\Gamma \vdash \mathsf{new\ C} : (T_1, .., T_n) \to C \quad \Gamma \vdash (e_1 \to T_1', .., e_n \to T_n') \quad T_1' \prec T_1, .., T_n' \prec T_n}{\Gamma \vdash \mathsf{new\ C}(e_1, .., e_n) : C}[newClass]$
Exp (plus)	$\dfrac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}[exp]$
Exp (minus)	$\dfrac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 - e_2 : int}[exp]$
Factor (equals)	$\dfrac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = T_2}{\Gamma \vdash e_1 == e_2 : bool}[factor]$
Factor (LTEQ)	$\dfrac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = T_2}{\Gamma \vdash e_1 \leqq e_2 : \ \ bool}[factor]$
Factor (GTEQ)	$\dfrac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = T_2}{\Gamma \vdash e_1 \geqq e_2 : \ \ bool}[factor]$
Factor (AND)	$\dfrac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = T_2}{\Gamma \vdash e_1 \,\&\&\, e_2 : \ \ bool}[factor]$
Factor (OR)	$\dfrac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = T_2}{\Gamma \vdash e_1 \parallel e_2 : \ \ bool}[factor]$
Term (times)	$\dfrac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \times e_2 : int}[term]$

Term (div)	$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \div e_2 : int}[term]$
Fun declaration	$\frac{\Gamma \bullet [f \mapsto (T_1,..,T_n) \to T] \vdash decs : \Gamma' \quad \Gamma \bullet [f \mapsto (T_1,..,T_n) \to T, x_1 \mapsto T_1, \ldots, x_n \mapsto T_n] \vdash progFun : T' \quad T' = T}{\Gamma \vdash T\ f(T_1 x_1,..,T_n x_n) = progFun; \quad decs : [f \mapsto (T_1,..,T_n) \to T] \bullet \Gamma'}$
Function call	$\frac{\Gamma \vdash f(T_1 \times \ldots \times T_2) \to T \quad (\Gamma \vdash x_i : T_i' \quad T_i' \prec T_i)^{i \in 1,..,n}}{\Gamma \vdash f(x_1, \ldots, x_n) : \quad T}[FunCall]$
Assignment (Asm)	$\frac{\Gamma(x) = T_1 \quad \Gamma \vdash e : T_2 \quad T_2 \prec T_1}{\Gamma \vdash x = e : void}[Asm]$
Let in exp	$\frac{\Gamma \vdash e : T'' \quad \Gamma[x \mapsto T] \vdash exp : T' \quad T'' \prec T}{\Gamma \vdash \text{let} \quad T\ x = e \quad \text{in} \quad exp : T'}[LetInExp]$
Start	$\frac{\Gamma \vdash (decclass_1 : T_1, \ldots, decclass_n : T_n) \quad \Gamma \vdash main : void}{\Gamma \vdash decclass_1; \ldots; decclass_n; \quad main : void}$
	La notazione $\prec$ indica una relazione di sottotipaggio

Ciascuna delle regole di inferenza del tipo sopra menzionate ha un codice Java corrispondente nella fase di verifica del tipo del compilatore FoolOO; se qualcosa va storto, il compilatore genererà immediatamente un'eccezione di runtime (o alla fine della fase di verifica del tipo) e rifiuterà di continuare ulteriori fasi.

Non è presente una regola di type checking specifica per la dichiarazione di una classe, in quanto viene richiamato il type checking per ciascuna funzione presente al suo interno.

Code generation