Progetto di compilatori e interpreti
A.A. 2017/2018

Anna Avena, mat. 0000842922,
Laura Cappelli, mat. 0000844166,
Daniele Rossi, mat. 0000855532.

Dipartimento di Informatica - Scienza e Ingegneria.
Università di Bologna.

Introduzione

Il progetto che abbiamo realizzato consiste in un'estensione del linguaggio FOOL in cui è stata aggiunta, fra le altre cose, la dichiarazione di classi e oggetti per renderlo un linguaggio Object-Oriented.

Di seguito verranno analizzate le 4 fasi che hanno costituito la costruzione del nostro compilatore, in particolare:

Nel primo capitolo viene trattata l'analisi lessicale in cui sono esaminati tutti i caratteri inseriti nel programma di input, viene verificato per ciascun token se questo è ammesso dalla nostra grammatica oppure se si tratta di un token illegale, in quest'ultimo caso viene lanciato un errore ed il processo termina immediatamente la sua esecuzione.

Nel secondo capitolo viene fatta l'analisi sintattica in cui si verifica che la costruzione dell'*abstract syntax tree* sia avvenuta con successo, ovvero che non ci siano delle malformazioni dovute alla mancanza di caratteri laddove la grammatica se li aspetta.
Un'esempio frequente è la mancanza di *semicolon* alla fine di uno *statement*.

Nel terzo capitolo viene spiegato il processo di analisi semantica in cui sono esplicitati i vari errori che si possono riscontrare durante la costruzione della *symbol table* oppure durante il *type checking* delle varie operazioni.

Nel quarto capitolo è definito come viene creato il codice assembly che è successivamente mandato ad *ExecuteVM* che esegue le istruzioni generate durante quest'ultima fase di generazione di codice.

1 Analisi lessicale

In questa fase vengono analizzati tutti i caratteri presenti nel programma di input, una volta che questi caratteri vengono divisi in token è compito del lexer analizzarli e una volta riconosciuti vengono passati al parser.
L'obiettivo di questa fase è la rilevazione di eventuali token illegali presenti nel

programma di input, in questo caso viene lanciata un'eccezione ed il processo termina la sua esecuzione.

Per il riconoscimento dei token ci si basa sulla grammatica definita in *FOOL.g4*.

1.1 FOOL Object Oriented

Di seguito sono riportate le regole lessicali aggiunte alla grammatica di base:

- Operatori *relazionali*:
  GE : '>=' ;
  LE : '<=' ;

- Operatori *logici*:
  NOT : ('not' — '!') ;
  AND : ('and' — '&&') ;
  OR : ('or' — '||') ;

- Operatori relativi alle *classi* e agli *oggetti*:
  CLASS : 'class' ;
  EXTENDS : 'extends' ;
  NEW : 'new' ;

- Altri operatori:
  DOT : '.' ;
  VOID : 'void' ;
  NULL : 'null' ;
  STDLIB : 'print' ;

Ogni volta che il lexer riconosce un determinato token lo invia al parser, che è meglio definito nel prossimo capitolo.

2 Analisi sintattica

Una volta che il lexer ha riconosciuto tutti i token, il parser li rileva e costruisce l'albero su cui viene fatta l'analisi sintattica.
L'obiettivo di questa fase è quello di segnalare se c'è stata una malformazione dell'albero sintattico astratto dovuta all'assenza di token previsti dalla grammatica ma non presenti nel programma di input.
Quando viene riscontrata una malformazione dell'AST viene lanciata un'eccezione e l'esecuzione del processo viene interrotta.

Di seguito viene fornito un breve esempio che rappresenta la costruzione dell'*abstract syntax tree*.

2.1 Abstract Syntax Tree

Per il seguente programma di input:

```
class Persona(int cod) {
    void getCod(){
        print(cod);
    };
};

let Persona x = new Persona(3); in x;
```

Viene generato questo albero di sintassi astratta:

```
Visualizing AST...
Start

Class: Persona
Par:cod
  IntType
 Fun: getCod
   VoidType
 Funbody
  Print
    Id: cod at nestlev 2
      STentry: nestlev 1
      STentry: type
        IntType
      STentry: offset 1
ProgLetIn
  Obj: x
    ClassType: Persona
    New obj of class Persona at nestlev 0
Class Persona
STentry: nestlev 0
STentry: type
  ClassType
cod
  IntType
getCod
 ArrowType
   ->VoidType
 -> Persona
STentry: offset 0
    Int:3
  Id: x at nestlev 0
    STentry: nestlev 0
    STentry: type
      ClassType: Persona
    STentry: offset 0
```

Dopo aver verificato la corretta costruzione dell'AST si procede con l'analisi semantica.

3 Analisi semantica

In questa fase si verifica se ci sono errori semantici come ad esempio:

- **Dichiarazioni multiple:** ciascuna variabile deve essere dichiarata al massimo una volta all'interno di uno *scope*.
- **Variabile non dichiarata:** devo assicurarmi che prima di utilizzare una variabile, questa deve essere stata dichiarata in quanto lo *scope* è statico.
- **Incongruenza di tipi:** il tipo dichiarato nella parte destra di un'assegnazione deve essere dello stesso tipo oppure sottotipo dichiarato nella parte sinistra.
- **Numero o tipo sbagliato di argomenti:** quando chiamo una funzione o un metodo devo assicurarmi di inserire il numero ed il tipo corretto dei parametri.

3.1 Scope checking

La *Symbol Table* è stata costruita secondo il metodo delle liste di *hashtable*.
In particolare si può accedere alla tabella dei simboli che si trova a *nesting level* 0 da qualunque parte del programma.
Se vi sono dichiarazioni di classi, la tabella dei simboli viene riempita con le informazioni riguardanti il nome della classe, i suoi campi con i relativi tipi ed i suoi metodi con gli eventuali parametri richiesti ed il tipo di ritorno.
Quando si accede ad un nuovo *scope* viene creata una nuova *hashtable* ad un *nesting level* successivo in cui vengono memorizzate le funzioni e le relative informazioni, all'uscita dello scope viene rimossa la tabella dei simboli dalla lista e viene decrementato il nesting level.

3.2 Type checking

Alcune operazioni sono possibili solo per valori di un certo tipo, l'obiettivo del *type checking* è proprio quello di controllare che le varie operazioni che si vogliono effettuare vengano usate con i tipi corretti.

Sono state applicate le regole di *type checking* viste a lezione, per le classi sono state applicate le seguenti regole:

- sia A una superclasse

```
class A() {
        T1 method1(T2 x) {
                ...
        }
}
```

- sia B una classe che estende A

class B extends A() {
T3 method1(T4 x) {
...
}
}

Valgono le seguenti regole di correttezza dei tipi:

T3 <: T1
T2 <: T4

4 Generazione del Bytecode

Dopo aver fatto l'analisi lessicale, sintattica e semantica, se non vi sono stati erorri, si procede con la generazione di codice oggetto.
L'obiettivo di questa fase è generare le istruzioni assembly che permetteranno di memorizzare ogni variabile, oggetto, metodo e funzione all'interno dell'indirizzo di memoria corretto.

Questa fase è composta da 3 parti:

- **Generazione di codice oggetto**
- **Visita del codice oggetto**
- **Esecuzione della Virtual Machine**

4.1 Generazione di codice oggetto

Per la generazione di codice di classi e oggetti abbiamo aggiunto ulteriori regole lessicali e sintattiche alla grammatica iniziale:

Added assembly instructions:

BRANCHGREATEREQ : 'bgeq' ; # jump to label if top >= next
ALLOC : 'alloc' ; # alloc an object in the heap
REMOVE : 'remove' ; # remove an object from heap
LOADF : 'lf' ; # load a field from the memory cell pointed by top
STOREF : 'sf' ; # store in the memory cell pointed by top the field next
JDT : 'jdt'; # jump to instruction pointed by dispatch table and store next instruction in ra
LOADIO : 'lio' ; # load object offset in the stack
LOADDP : 'ldp' ; # load dispatch pointer in the stack
STOREDP : 'sdp' ; # store top into dispatch pointer
LOADOP : 'lop' ; # load object pointer in the stack
STOREOP : 'sop' ; # store top into object poiter

4.2 Visita del codice oggetto

In questa fase ci si occupa di visitare le istruzioni presenti nel file "prova.fool.asm" creando l'albero sintattico e verificando la correttezza lessicale e sintattica del codice in input. Se ci sono errori viene lanciata un'eccezione e l'esecuzione del processo viene interrotta.
Per ogni istruzione visitata si riempie un'array di interi dove l'indice dell'array rappresenta l'i-esima istruzione visitata, ed il contenuto è il codice identificativo dell'istruzione definito nel parser.
Questo array si chiama *code* e verrà usato successivamente dalla VM.

4.3 Simple Virtual Machine

La SVM è una stack machine che utilizza il codice generato dal compilatore.
Il codice è contenuto nell'array *code* riempito precedentemente.
Durante l'esecuzione della VM si scansiona l'array *code* e per ogni codice trovato, lo si associa all'istruzione corrispondente.

I registri utilizzati dalla VM per operare sulla memoria sono:

- **IP:** instruction pointer → punta all'istruzione successiva da eseguire;
- **SP:** stack pointer → punta in cima allo stack;
- **FP:** frame pointer → punta al primo parametro della funzione;
- **HP:** heap pointer → punta alla coda dello heap;
- **DP:** dispatch pointer → è un array in cui gli indici rappresentano l'offset dell'oggetto ed il contenuto è l'indirizzo dove inizia il layout dell'oggetto nello heap;
- **OP:** object offset → rappresenta l'offset dell'oggetto;
- **RA:** return address → rappresenta l'indirizzo dell'istruzione successiva da eseguire quando termina l'esecuzione della funzione chiamata;
- **RV:** return value → memorizza il valore di ritorno di una funzione.

La memoria è gestita dalla VM, essa è costituita dallo ***heap*** e dallo ***stack***.
La memoria è composta da **10000** indirizzi.
Lo *heap* cresce dall'alto verso il basso, mentre lo *stack* cresce dal basso verso l'alto.

Sullo *heap* vengono memorizzati campi, oggetti e classi, mentre sullo *stack* si memorizzano variabili e funzioni.

4.4 Dispatch Table

Per il linguaggio OO è necessario gestire la rappresentazione in memoria degli oggetti, questo viene fatto mediante l'utilizzo della DT.
La DT è stata implementata come arraylist di arraylist, ciascun indice dell'arraylist esterno corrisponde all'offset di una classe ed ogni elemento è rappresentato da un

arraylist contenete i metodi della classe: l'indice di questo arraylist corrisponde all'offset del metodo, mentre l'elemento è l'i-esima istruzione dell'array *code* dove inizia il codice del metodo stesso.
I metodi sovrascritti si trovano allo stesso offset sia nella classe che nella superclasse.

Si consideri il seguente esempio:

```
class A() {
        T method1(T' x) {
                ...
        }
}

class B extends A() {
        T method1(T' x) {               // override
                ...
        }

        T method2(T' x) {
                ...
        }
}

class C extends A() {
        T method3(T' x) {
                ...
        }

        T method4(T' x) {
                ...
        }
}
```

	Offset 0 Class A	**Offset 1** Class B	**Offset 2** Class C
Offset 0	Pointer to method1( ) of A	Pointer to method1( ) of B	Pointer to method1( ) of A
Offset 1	-	Pointer to method2( ) of B	Pointer to method3( ) of C
Offset 2	-	-	Pointer to method4( ) of C