Alma Mater Studiorum · Università di Bologna

Progetto di compilatori e interpreti: estensione del linguaggio SimpleStaticAnalysis

LM Informatica - Curriculum B: Informatica per il management

D'Ambrosio Luca
Matricola: 0000890761

Silvestri Marco
Matricola: 0000889874

Anno Accademico 2018/2019

Indice

Elenco delle figure

Introduzione

Il progetto sviluppato è il completamento dell'esame del corso di "Compilatori e Interpreti", che ha lo scopo di estendere un linguaggio denominato "SimpleStaticAnalysis".
Il progetto è stato realizzatto in diversi step, infatti, durante il corso, sono stati consegnati i seguenti assegnamenti:

1. Analisi Lessicale
2. Grammatica per estensione di SimpleStaticAnalysis
3. Analisi Semantica
4. Traduzione in bytecode e Interprete

Questo documento descrive la realizzazione del quarto assegnamento, i precedenti sono stati consegnati entro le relative scadenze durante lo svolgimento del corso tramite la piattaforma "IOL".
Nei prossimi capitoli verranno analizzati gli strumenti che sono stati utilizzati per la realizzazione del progetto con la sua relativa struttura, inoltre, verranno approfondite le 4 fasi che hanno caratterizzato la costruzione del nostro compilatore.

Capitolo 1

Strumenti utilizzati e struttura del progetto

1.1 Strumenti e software utilizzati

Il software utilizzato nello sviluppo del progetto è:

- **ANTLR**: un generatore di parser che fa uso del sistema di parsing LL. L'acronimo è il seguente: ANother Tool for Language Recognition. La versione utilizzata è la 4.6.

Figura 1.1: Logo di "Antlr"

- Il progetto è stato realizzato nell'ambiente di sviluppo integrato **Eclipse IDE for Java Developers**.

Figura 1.2: Logo di Eclipse

1.2 Struttura del progetto

Il progetto è organizzato in diversi package, che mirano a dividere le due parti del progetto: compilatore e interprete.

- **Compilatore**
  - models: contiene 3 classi che permettono di controllare il comportamento del compilatore.
  - models.exp
  - models.factor
  - models.parameter
  - models.statement
  - models.stentry
  - models.term
  - models.type

  - models.value

- **Interprete**
  - models.vm: contiene 2 classi che permettono il funzionamento dell'interprete.

- **parser**: contiene la grammatica e le classi relative a lexer e parser.

- **analyser**: contiene la classe Analyser che permette la di eseguire l'intero progetto.

- **util**: contiene delle classi utili alla gestione degli errori, al controllo del type check e dei metodi generali utili per l'implementazione.

Il progetto include anche 4 file di testo:

- **test.spl**: file di testo che permette di scrivere il codice da testare per il progetto.

- **test.spl.asm**: una volta generate, il file contiene le regole assembly utili per l'esecuzione del programma.

- **test.spl.syntaxError**: contiene gli errori sintattici individuati durante l'esecuzione del programma.

- **serbatoio**: contiene possibili esempi da testare per l'esecuzione del progetto.

Inoltre, il progetto è caratterizzato dalla presenza della libreria ANTLR.

Capitolo 2

Implementazione progetto

2.1 Compilatore

2.1.1 Analisi Lessicale

Scopo dell'analisi lessicale è quello di riconoscere nella stringa di ingresso alcuni gruppi di caratteri che corrispondono a specifiche categorie sintattiche.
In tal modo la stringa di ingresso è trasformata in una sequenza di simboli astratti, detti **token**, che sono poi passati all'analizzatore sintattico. Nel caso siano presenti token illegali, viene lanciata un'eccezione facendo terminare il processo. Gli errori generati vengono sia visualizzati sullo schermo che salvati nel file `test.spl.syntaxError`.

Il riconoscimento dei token avviene sulla base della grammatica descritta nel file "ComplexStaticAnalysis.g4" riportata di seguito.

```
grammar ComplexStaticAnalysis;

// PARSER RULES
block       : '{' statement* '}' ;

statement   : assignment ';'
            | deletion ';'
            | print ';'
            | functioncall ';'
            | ifthenelse
            | declaration
            | block ;

assignment  : ID '=' exp ;

deletion    : 'delete' ID ;

print       : 'print' exp ;

functioncall : ID '(' (exp (',' exp)* )? ')' ;

ifthenelse    : 'if' '(' condition=exp ')' 'then' thenBranch=block 'else' elseBranch=block ;

declaration : type ID '=' exp ';'                              #varDec
            | ID '(' ( parameter ( ',' parameter)* )? ')' block #funDec;

type        : 'int'
            | 'bool';

parameter   : (modeParameter='var')? type ID ;

exp         : (minus='-')? left=term (op=('+' | '-') right=exp)? ;

term        : left=factor (op=('*' | '/') right=term)? ;

factor      : left=value (op=ROP right=value)?             #intFactor
            | left=value (op=('&&' | '||') right=value)?   #boolFactor;

value       : INTEGER                          #intValue
            | ( 'true' | 'false' )             #boolValue
            | '(' exp ')'                      #expValue
            | ID                               #idValue;
```

```
// LEXER RULES
ROP     : '==' | '>' | '<' | '<=' | '>=' | '!=' ;

//NUMBERS
fragment DIGIT : '0'..'9';
INTEGER       : DIGIT+;

//IDs
fragment CHAR  : 'a'..'z' |'A'..'Z' ;
ID              : CHAR (CHAR | DIGIT)* ;

//ESCAPE SEQUENCES
WS              : (' '|'\t'|'\n'|'\r')-> skip ;
LINECOMENTS     : '//' (~('\n'|'\r'))* -> skip ;
BLOCKCOMENTS    : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'|BLOCKCOMENTS)* '*/' -> skip ;
ERR         : .  -> channel(HIDDEN) ;
```

Figura 2.1: Grammar ComplexStaticAnalysis

2.1.2 Analisi Sintattica

Costruita la lista di token, l'analizzatore sintattico (o **parser**) cerca di costruire un albero di derivazione per tale lista. Si tratterà di un albero di derivazione nella grammatica del linguaggio. Ogni foglia di tale albero deve corrispondere ad un token contenuto nella lista ottenuta dall'analisi lessicale. Questo albero rappresenta la struttura logica del programma, che sarà poi sfruttata nelle fasi successive della compilazione.
Quando viene riscontrata una malformazione dell'AST viene lanciata un'eccezione facendo terminare il processo, questi errori, come descritto nell'analisi precedente vengono stampati sia nella console Eclipse che nel salvati nel file `test.spl.syntaxError`.
Forniamo un esempio che rappresenta la costruzione dell'Abstract Syntax Tree.

Per il dato programma di input:

```
{
    int x = 1;
    f(int y){
        if (y == 0) then {
            print(x);
        } else {
            f(y-1) ;
        }
    }
    f(54) ;
}
```

Figura 2.2: test.spl

Viene generato il seguente albero di sintassi astratta (AST):

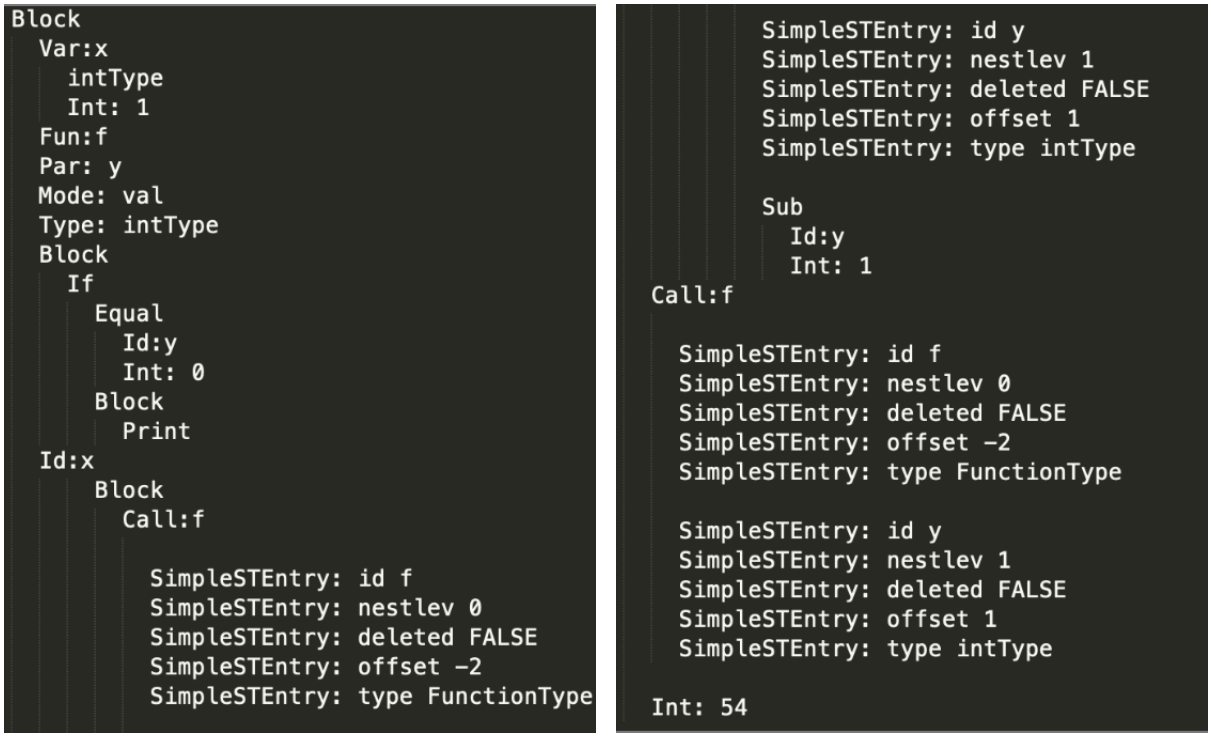

Figura 2.3: Abstract syntax tree

2.1.3 Analisi Semantica

L'analisi semantica consiste nell'analisi dell'albero di derivazione (che testimonia la correttezza sintattica della stringa di partenza) che viene sottoposto ai controlli relativi ai vincoli contestuali del linguaggio.
In quest'analisi vengono controllate le dichiarazioni, i tipi, il numero dei parametri delle funzioni ecc. Con il susseguirsi dei controlli, l'albero di derivazione si completa con la relativa informazione, e nuove strutture dati vengono generate.
Un analizzatore semantico agisce attraversando l'AST costruito e si compone di due fasi:

1. **Scope checking**: seguendo la teoria, nel progetto abbiamo utilizzato di 2 symbol table, rispettivamente:
   (a) **vtable**, che contiene le variabili con i relativi valori,
   (b) **ftable**, che lega le funzioni con i suoi dati.

Entrambe sono state costruite secondo il metodo delle liste di hashtable.
Ogni volta che si accede ad un nuovo blocco, quindi in nuovo scope, il nesting level viene aumentato di 1, di conseguenza ad ogni chiusura di questo corrisponde una diminuzione del nesting level.
Queste operazioni sono realizzate nel progetto tramite il metodo `checkSemantics()`.

2. **Type checking**: consiste nel controllare che i valori assegnati ad una variabile siano di un tipo di dato ammissibile per il tipo della variabile. Sono state applicate le regole di type checking viste a lezione.
   Queste operazioni sono realizzate nel progetto tramite il metodo `typeChecking()`.

Tutti gli errori semantici riscontrati sono stampati sulla console di Eclipse.

2.1.4 Generazione Bytecode

L'obiettivo di questa fase è generare le istruzioni assembly che permetteranno di memorizzare ogni variabile e funzione all'interno dell'indirizzo di memoria corretto. Questa fase è composta da e fasi:

1. **Generazione del codice oggetto**: è stato utilizzate il codgen visto a lezione (esempi compilatore FOOL) con l'aggiunta di altre regole specifiche.

2. **Visita del codice oggetto**: esaminiamo le istruzioni presenti nel file "test.spl.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 visita di un istruzione viene riempito l'array di interi (`code`) in cui l'indice di quest'ultimo rappresenta l'i-esima istruzione visitata che rappresenta il codice identificativo dell'istruzione definita nel parser.

3. **Esecuzione del Virtual Machine**: è una stack machine che utilizza il codice generato nelle fasi precedenti, il suo obiettivo è quello di scansionare l'array `code` e per ogni codice identificativo trovato, lo si associa all'istruzione corrispondente.

2.2 Interprete

L'interprete traduce il bytecode precedentemente generato eseguendo il codice ottenuto attraverso l'utilizzo di registri specifici.
Quest'esecuzione viene realizzata attraverso il metodo cpu della classe ExecuteVM.
La memoria è gestita dalla VM ed è costituita dallo **heap** e dallo **stack**, composta da 10000 indirizzi.