Progetto di Compilatori e Interpreti:
FOOL compiler

Autore:
Paolo Santilli 842785

A.A. 2018/2019

ISTRUZIONI

1. Importare il progetto in Eclipse
   1.1. Click destro in *Package Explorer* > *Import* > *General* > *Existing Projects into Workspace;*
2. Tramite il file *main.fool* è possibile inserire il codice fool che si vuole compilare ed eseguire;
3. Lanciare *"Start.java"* per compilare ed eseguire *main.fool;*
4. L'output è visibile nella console dell'ide. In tale output è possibile vedere:
   - Eventuali errori riscontrati nell'analisi lessicale
   - Eventuali errori riscontrati nell'analisi sintattica
   - Eventuali errori riscontrati nell'analisi semantica (type checking) con il tipo del programma
   - l'utilizzo dello stack
   - l'eventuale risultato del programma nel caso in cui il tipo ottenuto durante la fase di type checking sia diverso da void.

N.B: nella directory "testFile" è possibile accedere ai test effettuati.

INTRODUZIONE

Il progetto consiste nella realizzazione di un compilatore per il linguaggio FOOL. In questo documento verrà descritta la struttura del progetto svolto per una più rapida consultazione. Gli autori hanno cominciato a lavorare partendo dal progetto esempio fornito dal professore. Hanno effettuato un refactoring del codice e successivamente modificato la grammatica FOOL.g4, aggiunto le classi relative alla programmazione ad oggetti, al sotto-tipaggio e alla gestione dello heap. Il codice del progetto è contenuto nella directory *"src"*, la quale è a sua volta divisa in diversi package:

- **ast**, contiene una classe per ogni nodo dell'AST. Ogni classe estende l'interfaccia *Node* ed implementa i relativi metodi per il controllo della semantica (*checkSemantics*()), per il type checking (*typeCheck*()) e per la generazione del codice (*codeGeneration*());
- **parser**, contiene la grammatica per FOOL (FOOL.g4) i relativi Parser e Lexer generati da ANTLR in base alla grammatica. Inoltre, contiene la classe *FoolVisitorImpl*, la quale estende *FoolBaseVisitor* e permette una più dettagliata visita dell'AST;
- **type**, contiene una classe per ogni tipo implementato dal linguaggio FOOL;
- **throwable,** contiene il package per le eccezioni;
- **symboltable,** contiene le classi ausiliarie per l'analisi semantica ossia la classe relativa alla symbol-table e la classe entry di cui la symbol-table è composta
- **codegen,** contiene la grammatica SVM.g4 per la generazione del codice macchina con i relativi parser e lexer, anch'essi generati automaticamente in base la grammatica da ANTLR, le classi ausiliare *FunctionCode* e *Label* per la gestione delle etichette e delle funzioni. Inoltre, il corrente package contiene le classi ausiliarie per la gestione della Virtua Machine e per la definizione dello Heap;
- **lib**, contiene la libreria "antlr-4.7-complete.jar" necessaria per ANTLR 4.

Il file nelle directory "main.fool" è utilizzato per inserire codice fool che successivamente verrà compilato ed eseguito dalla classa *Start.java*

Analisi Lessicale e Sintattica

L'analisi lessicale e sintattica viene eseguita automaticamente da ANTLR sulla base della grammatica fornita. Di seguito vengono indicate le principali modifiche effettuate alla grammatica (FOOL.g4) per includere le modifiche richieste dalla consegna.

Grammatica FOOL

Di seguito sono elencate le modifiche efffettuate:

- In *value* sono state aggiunte le regole per la chiamata di metodi (met) in maniera da differenziarla dalle funzioni normali e l'istanziazione di un oggetto (new).
- Sono stati aggiunti gli *statements* (if-stms e assegnamento) con aggiunta anche all'IN del LET.
- La dichiarazione di classi.
- È stata eliminata da *fun* la possibilità di definire al suo interno una nuova funzione ed è stata aggiunta anche la possibilità di restituire void (nel caso in cui il corspo sia composto da statement)
- In *varasm* è stata aggiunta la possibilità di assegnare null ad una variabile (il controllo che null possa essere assegnato solo ad una variabile di tipo oggetto viene eseguito nella analisi semantica).
- La chiamata di funzione.
- La chiamata di un metodo.
- Aggiunti gli operatori ($\leq,\ \geq,\ <,\ >,\ \&\&,\ ||,\ not, -$).

Analisi Semantica

Lo Scope Checking e il Type Checking vengono eseguiti su ogni nodo dell'AST. Ogni nodo estende l'interfaccia Node ed implementa i metodi checkSemantics() per lo Scope Checking e typeCheck() per il Type Checking. Viene lanciata una visita ricorsiva dell'AST e tramite FoolVisitorImpl si svolge il controllo semantico di ogni nodo visitato.

Scope Checking

Per eseguire Scope Checking su ogni nodo dell'AST è stata utilizzata la SymbolTable e la Entry, una classe ausiliaria per la SymbolTable. La Symbol Table è stata implementata tramite una lista di HashTable (campo symTable)

```
private List<HashMap<String, Entry>> symTable = new ArrayList<HashMap<String,Entry>>();
```

Con i seguenti campi:

- **private int offset,** offset della entry rispetto all'area di memoria in cui è stata definita.
- **private** Entry **lastEntryIstance,** ossia la entry dell'ultima classe visitata nella visita all'AST.

e metodi:

- **public void** newScope(), incrementa il nestingLevel (dimensione della symTable - 1) e aggiunge un livello di scope, aggiunge una nuova hasmap alla lista;
- **public void** exitScope(), rimuove lo scope di esterno ossia rimuove l'ultima hashmap aggiunta;
- **public** Entry checkID(String id),dato un ID cerca la Entry con quell'id nella Symbol Table;
- **public int** getNestingLevel();
- **public void** setOffset(**int** offset)
- **public void** decreaseOffset(), offset – 1;
- **public** SymbolTable insertDeclaration(String id, Type type, **int** offset), crea una entry con chiave "id" e la inserisce nello scope più esterno;
- **public** SymbolTable setDeclarationType(String id, Type newtype, **int** offset), aggiorna l'attributo type della entry con chiave id. E' utilizzato per aggiornare il supertipo delle classi.

Type Checking

Il compilatore esegue il type checking su ogni nodo dell'AST, con un approccio bottom-up. Ogni nodo contiene il metodo typeCheck() in cui sono applicate le relative regole di inferenza. Viene infine ritornato un tipo, rappresentante il tipo dell'espressione del programma dato in input. Per eseguire un'analisi più chiara, si è deciso di separare i tipi dai nodi dell'AST. Ogni tipo estende l'interfaccia Type (package type) ed implementa i seguenti metodi:

- ID getID(), restituisce il proprio tipo.
- **boolean** isSubType(Type t), utilizzato per gestire le regole di subtyping;
- String toPrint(), per stampare ad output il tipo finale del programma.

I tipi implementati sono stati:

- **INT,** IntType
- **BOOL**, BoolType
- **RETURN**, FunType, ossia il tipo di una funzione $(P_1 \times P_2 \times \ldots \times P_n) \rightarrow T$
- **CLASS**, ClassType
- **OBJECT**, ObjectType
- **VOID,** VoidType

Subtyping

Sono state implementate le regole di sotto-tipaggio come da consegna, in particolare:

- 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 supertipo del corrispondente tipo di parametro di f2. Tutto ciò stato implementato in FunType mediante il metodo isSubType.

- 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. Anche questo implementato tramite il metodo isSubType della classe ClassType.

Code Generation

In questo capitolo verranno trattate le parti più rilevanti della fase di generazione del codice.

Dispatch Table

Le dispach table di tutti gli oggetti vengono memorizzate in DispatchTable.java come DTentry.java. La struttura di una DTentry è la seguente:

DTentry		
Nome	**Tipo**	**Descrizione**
methodID	String	Identificatore del metodo
methodLabel	String	Etichetta delimitatrice per l'inizio del codice della funzione

Ogni classe può aggiungere la sua dispatch table usando il proprio nome come chiave addDispatchTable(idclasse,DTentries). Al termine della fase di generazione del codice, viene chiamato generateDispatchTableCode(): così facendo viene generato il codice relativo a tutte le dispatch table. Tale codice, viene concatenato al codice SVM precedentemente creato.

Virtual Machine

Il codice che è stato generato, viene eseguito da una Stack Virtual Machine (SVM). La memoria di questa macchina è uno stack e la computazione è espressa mediante modifiche allo stack tramite operazioni di push e pop. La classe ExecuteVM gestisce la macchina virtuale e associa ad ogni istruzione, che gli viene passata come input in un array, un'implementazione in termini di operazioni di push e pop sullo stack.

Gli oggetti, che sono istanze delle classi, non possono essere memorizzate sullo stack insieme al resto dei dati: perciò è stata inserita l'operazione di new, usata per allocare un oggetto nella parte più alta della memoria, in un'aerea chiamata heap.

Heap

Lo heap è definito dalle classi HeapMemory e HeapMemoryCell: in esse si trovano le istruzioni che gestiscono l'allocazione e la deallocazione degli elementi. Lo heap è implementato con una lista libera in modo da facilitare il garbage collection. Infatti, dopo una serie di allocazioni e deallocazioni di dimensione differenti, lo heap potrebbe presentare frammentazione interna e l'uso della lista libera permette alla macchina di operare senza tenere conto di questo problema.

Garbage Collector

Il garbage collector realizzato implementa la tecnica del mark and sweep:

1. Viene utilizzata una hashmap contenente l'indice delle celle di memoria e un valore booleano;
2. Il valore booleano di tutte le celle viene inizializzato a false;
3. Viene esaminato lo stack e per ogni oggetto trovato, il corrispondente valore booleano viene settato a true;
4. Se un oggetto viene trovato nel registro RV, il suo corrispondente valore booleano viene settato a true;
5. Viene deallocato ogni elemento che abbia il corrispondente valore booleano settato a false.

L'operazione di garbage collection viene eseguita prima di allocare un oggetto, se la differenza tra i registri sp e hp è minore o uguale al massimo tra 10% della memoria totale e in caso di memoria particolarmente piccola (10).

Grammatica SVM

Il file SVM.g4 definisce la grammatica riconosciuta dalla macchina virtuale. Sono state aggiunte le seguenti istruzioni:

• LOADC

• NEW

• HEAPOFFSET

• COPY

LOADC

L'istruzione LOADC `e stata aggiunta per implementare il dispatch dinamico. Dato un indice dell' array code (inizio della dispatch table di una classe, memorizzata come primo elemento dell'oggetto nello heap), mette sullo stack il contenuto di code[index] (indirizzo del metodo all'interno di code).

NEW

NEW alloca un'area di memoria nello heap e viene utilizzato per instanziare un oggetto. In cima allo stack deve essere presente l'indirizzo della dispatch table della classe istanziata, il numero di campi e i valori da assegnare ai campi. Nello heap viene inserito in prima posizione l'indirizzo della dispatch table e successivamente i valori dei campi. 14

HEAPOFFSET

Come detto precedentemente, il garbage collector pu`o causare la memorizzazione di oggetti in celle non contigue di memoria. Questa istruzione converte l'offset di un campo di un oggetto nell'offset reale tra l'indirizzo dell'oggetto nello heap e l'indirizzo del campo. L'offset cos`ı calcolato viene inserito in cima allo stack. Per poter funzionare, in cima allo stack deve essere presente l'indirizzo dell'oggetto del quale si richiede il valore del campo e l'offset logico del campo rispetto all'indirizzo del suo spazio nello heap.

COPY

L'istruzione COPY duplica il valore in cima allo stack.

Licenza