Fool Compiler

Alessandro Zini - 819437
Mattia Maldini - 800332
Carlo Stomeo - 819505

Introduzione

Questo progetto è un esercizio puramente accademico sull'implementazione di un compilatore/interprete per un linguaggio di programmazione semplificato, FOOL. Il lavoro è stato svolto partendo da una base giá funzionante e fornita dal professore, che è stata poi estesa con le caratteristiche richieste. Il progetto si basa sul tool per la generazione automatica di parser Antlr, nella sua versione 4.7.
Nella sua versione iniziale FOOL prevedeva le operazioni di base di un linguaggio di programmazione imperativo (definizione e utilizzo di simboli e funzioni). Nel corso del progetto è stato esteso parzialmente con il paradigma orientato a oggetti, dando la possibilitá di definire classi e istanziarle a runtime.
Si tratta appunto di un esercizio di stile senza alcuna finalitá pratica, in quanto il linguaggio si basa su programmi e funzioni composti da una singola istruzione effettiva e utilizza variabili dal valore non modificabile (di fatto soltanto costanti). Essendo queste limitazioni derivanti direttamente dalla grammatica BNF stabilita dalla consegna sono state intese come imposizioni e non modificate.

Nella sua versione definitiva, il compilatore così costruito accetta un linguaggio con le seguenti caratteristiche:

- Dichiarazione di variabili (booleane, intere o oggetti) non modificabili
- Dichiarazione e definizione di funzioni
- Dichiarazione e istanziazione di classi

Il linguaggio, per quanto semplice è comunque Turing completo. Mostreremo dopo aver definito la grammatica una motivazione formale.

Utilizzo

Il progetto è scritto in Java e, data la relativa semplicitá della struttura, si basa sull'utility GNU Make. È stato sviluppato e pensato come tool a riga di comando. Non è stato usato l'ambiente di sviluppo Eclipse, consigliato durante il corso, in quanto alcuni membri del gruppo non dispongono di dispositivi con prestazioni sufficienti da utilizzare un ambiente così pesante in maniera affidabile.
Le istruzioni sulla compilazione e l'utilizzo sono contenute nel file README.md della repository git.

Sintassi

La sintassi di FOOL è definita dalla seguente grammatica. Un programma FOOL puó essere inteso come tre diversi costrutti incrementali: una singola espressione, una serie di dichiarazioni a cui segue un'espressione, oppure uno di questi costrutti preceduto da una lista di definizioni di classi.

```
prog   : exp SEMIC                        #singleExp
       | let exp SEMIC                    #letInExp
       | (classdec)+ SEMIC (let)? exp SEMIC     #classExp
       ;
```

In ognuno di questi casi l'effettiva istruzione è una sola. Questo rende il linguaggio vicino al paradigma funzionale (ML) piuttosto che a quello imperativo (C).
Il non terminale exp puó infatti tradursi solo nel modo seguente. Le prime regole servono a forzare la precedenza dei vari operatori (nell'ordine di importanza: confronto, moltiplicazione/divisione, somma/sottrazione).

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

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

factor : left=value ((EQ|GREAT|LESS|GREATEQ|LESSEQ) right=value)?
     ;
```

Il non terminale value indica invece quello che puó essere presente in un'espressione:

- un intero
- un booleano
- un costrutto if-then-else
- un riferimento a variabile
- una chiamata di funzione
- una chiamata di metodo su un oggetto

```
value  :  INTEGER                                    #intVal
     | ( TRUE | FALSE )                                    #boolVal
     | LPAR exp RPAR                                          #baseExp
     | IF cond=exp THEN CLPAR thenBranch=exp CRPAR ELSE CLPAR elseBranch=exp CRPAR
#ifExp
     | ID                                                  #varExp
     | THIS                                                                  #thisExp
     | ID ( LPAR (exp (COMMA exp)* )? RPAR )          #funExp
     | PRINT ( LPAR exp RPAR )          #printExp
     | ID DOT ID ( LPAR (exp (COMMA exp)* )? RPAR )    #methodExp
     | NEW ID (LPAR exp (COMMA exp)* RPAR)?                   #newExp
     ;
```

Il resto della grammatica viene qui omesso. Copre la parte di definizione di variabili, classi e funzioni, e segue l'approccio classico.

Oltre a quelle per la definizione di classi, funzioni e variabili e al costrutto if-then-else l'unica altra keyword di FOOL è "print", che ovviamente stampa a video il valore intero passato come parametro.

Struttura

Il progetto si suddivide concettualmente in due parti:

1. Fjc (Fool Java Compiler): il compilatore che traduce il linguaggio FOOL nel codice assembler di una arbitraria macchina virtuale.
2. Fool: l'interprete del codice assembler, la macchina virtuale a pila.

Di fatto il passaggio da FOOL al codice assembler è assolutamente superfluo. Sarebbe possibile e molto più efficiente compilarlo direttamente in un bytecode (o nel codice assembler di una macchina reale) senza dover passare attraverso un ulteriore parser per il codice intermedio. Tuttavia in questo modo si è fatta esperienza su un campo di lavoro più

ampio, comprendente anche un interprete. Avere un codice assembler umanamente "leggibile" è stato inoltre un supporto non indifferente allo studio e all'implementazione. Compilatore e interprete sono presenti nei file `Fjc.java` e `Fool.java`. Le cartelle contengono, rispettivamente:

- `ast/` (abstract syntax tree) : contiene tutte le definizioni dei nodi dell'albero di sintassi astratta (sotto forma di classi java) che viene costruito facendo il parsing del programma.
- `lib/` : contiene i file `.jar` delle librerie utilizzate nel progetto.
- `parser/` : contiene le grammatiche di FOOL e del codice assembler, insieme a tutti i file automaticamente generati da ANTLR.
- `util/` : contiene alcune classi utilizzate a vario titolo nel progetto, come l'implementazione della symbol table (`Environment.java` ed `STentry.java`).
- `test/` : contiene una lista di esempi di programmi fool. Usando il comando `make test` vengono tutti compilati ed eseguiti, controllandone l'output.
-

Il compilatore usa il pattern Visitor offerto dal runtime di ANTLR (file `ast/FoolVisitorImpl.java`) per percorrere l'albero di parsing e costruire un albero di sintassi astratta formato dai nodi di vario tipo. Una volta costruito questo albero viene percorso al più altre 4 volte ricorsivamente. Necessariamente per fare il check semantico, di tipo e generazione di codice del programma, opzionalmente per essere stampato a video.

Analisi Semantica

La fase di analisi semantica ha il compito di attraversare l'albero di sintassi astratta, costruito dal parser, e per ogni *scope* all'interno del programma deve:

- **processare le dichiarazioni**: per ogni nuova variabile dichiarata deve essere aggiunta una *entry* corrispondente nella tabella dei simboli. Questa fase si occupa anche del rilevamento delle dichiarazioni multiple all'interno di uno stesso scope del programma.
- **processare le istruzioni**: ogni istruzione del programma viene analizzata per rilevare eventuali utilizzi di identificatori di variabili non dichiarati, e per aggiornare e collegare correttamente i nodi dell'albero di sintassi astratta alle relative *entry* della tabella dei simboli.

Il design della tabella dei simboli è stato realizzato in accordo con le peculiarità di FOOL. In primo luogo, il linguaggio utilizza scoping e sistema di tipaggio statici, e per questo motivo è necessario dichiarare ogni variabile prima del suo utilizzo. Inoltre, è stata negata la possibilità di utilizzare uno stesso identificatore per la dichiarazione di più variabili all'interno dello stesso scope; tuttavia, è possibile utilizzare uno stesso identificatore all'interno di scope annidati. Infine, è stato utilizzato lo stesso scope per i parametri di funzioni e per le variabili dichiarate all'inizio di tali funzioni.

Il design risultante per la tabella dei simboli è stato quello di una lista di tabelle hash. Per semplificare il più possibile la struttura e la forma della tabella, è stato scelto di mantenere ogni entry il più semplice possibile; per questo motivo, sono presenti solo 3 campi:

- il nodo tipo relativo al simbolo a cui l'entry fa riferimento;
- il suo nesting level;
- il suo offset all'interno del nesting level.

L'estensione *object-oriented* di FOOL ha richiesto l'introduzione di numerosi cambiamenti. In primo luogo, è stato necessario estendere il parser in modo che riconoscesse correttamente la definizione di una nuova classe, compresi i suoi campi e i suoi metodi, nonché l'istanziazione di un nuovo oggetto, la chiamata al suo costruttore e i riferimenti relativi alla chiamata di metodi su un oggetto.

Pertanto, al momento della creazione dell'albero di sintassi astratta, nel farne il parsing il nostro compilatore costruisce due nodi:

- un **ClassNode**, che contiene tutte le informazioni della classe, come i suoi campi e metodi;
- un **ClassTypeNode**, che rappresenta il tipo della classe e ricorda soltanto le informazioni ad esso relative.

Il tipo è ovviamente utilizzato per i controlli di tipo nella fase di type check. La classe invece usa queste informazioni per costruire, nella fase di generazione di codice, la propria Vtable e il proprio costruttore.

Nonostante nelle fasi iniziali del progetto le dichiarazioni di nuove classi erano state inserite a livello di annidamento 0 della tabella dei simboli, nei passaggi successivi è stato deciso di spostare tali dichiarazioni all'interno dell'*environment* vero e proprio, nel quale la tabella stessa vive. Infatti, è importante notare come la sintassi di FOOL permetta la definizione di classi solamente come prime istruzioni di un programma; inoltre, una dichiarazione di classe non è altro che una definizione di un "nuovo tipo", ed un suo eventuale inserimento a livello di annidamento 0 avrebbe reso necessario creare un ulteriore "meta-tipo" che indicasse una definizione di classe.

In aggiunta ai nodi ClassNode e ClassTypeNode sono stati aggiunti tutti i nodi relativi alle componenti di una classe, come FieldNode (per la gestione dei campi), MethodNode (per la gestione dei metodi), MethodCallNode (per la gestione delle invocazioni di metodo) e ConstructorNode (per la gestione delle invocazioni del costruttore durante l'istanziazione di nuovi oggetti). Il punto di ingresso del programma nel caso esso contenga una dichiarazione di classe è stato gestito da ProgClassNode.

Type Check

Per prima cosa definiamo le regole di derivazione dei tipi, successivamente descriveremo gli interventi che sono stati attuati per applicarle.

1. Costanti intere:

$$\frac{x \text{ é } un\ token\ di\ tipo\ numerico}{\Gamma \vdash x : int}\ \text{[int Val]}$$

2. Costanti booleane:

$$\frac{}{\Gamma \vdash true : bool}\ \text{[Bool 1]} \qquad \frac{}{\Gamma \vdash false : bool}\ \text{[Bool 2]}$$

3. Parentesi:

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash (x) : T}\ \text{[Par]}$$

4. If-then-else:

$$\frac{\Gamma \vdash cond : bool \qquad \Gamma \vdash a : T1 \qquad \Gamma \vdash b : T2 \qquad T1 <: T2}{\Gamma \vdash if\ cond\ then\ \{a\}\ else\ \{b\} : T2}\ \text{[Ite 1]}$$

$$\frac{\Gamma \vdash cond : bool \qquad \Gamma \vdash a : t1 \qquad \Gamma \vdash b : T2 \qquad T2 <: T1}{\Gamma \vdash if\ cond\ then\ \{a\}\ else\ \{b\} : T1}\ \text{[Ite 2]}$$

5. Variabili:

$$\frac{}{\Gamma[x \rightarrow t] \vdash x : t}\ \text{[Var]}$$

6. Applicazione di funzione:

$$\frac{\Gamma \vdash f : (T1, \ldots, Tn) \rightarrow T \qquad (\ \Gamma \vdash xi : Ti' \qquad Ti' <: Ti\ )\ i \in [1 \ldots n]}{\Gamma \vdash f(x1, \ldots, xn) : T}\ \text{[FunApp]}$$

7. Let-in:

$$\frac{\Gamma\ [x \rightarrow T] \vdash e : T'}{\Gamma \vdash let\ T\ x\ in\ e : T'}\ \text{[Let]}$$

$$\frac{(\ \Gamma \vdash ei : Ti' \qquad Ti' <: Ti\ )\ i \in [1 \ldots n] \qquad \Gamma\ [x1 \rightarrow T1] \ldots [xn \rightarrow Tn] \vdash b : T}{\Gamma \vdash let\ T1\ x1 = e1\ , \ldots , Tn\ xn = en\ \ in\ b : T}$$

[LetIn]

8. Print:

$$\frac{}{\Gamma \vdash print(x) : void} \quad \text{[Print]}$$

9. Classi:

$$\frac{\Gamma : new\ C :\ (t1, \ldots, tn) \rightarrow C \qquad (\Gamma \vdash ei : ti' \qquad ti' <: ti)\ i \in [1...n]}{\Gamma \vdash new\ C(e1, \ldots, en) : C} \quad \text{[Class]}$$

10. Applicazione di un metodo:

$$\frac{\Gamma \vdash m : (C, t1, \ldots, tn) \rightarrow T \qquad (\Gamma \vdash xi : ti' \qquad ti' <: ti)\ i \in [1...n] \qquad \Gamma \vdash o : C' \qquad \Gamma \vdash C' <: C}{\Gamma \vdash o.m(x1, \ldots, xn) : T}$$

[MethodApp]

11. Field:

$$\frac{\Gamma \vdash o: C \qquad C \vdash x:T}{\Gamma \vdash o.x : T} \quad \text{[Field]}$$

12. Assegnamento di variabile:

$$\frac{\Gamma[x \rightarrow T] \vdash e : T' \qquad T' <: T}{\Gamma[x \rightarrow T] \vdash x = e : T} \quad \text{[VarAsm]}$$

13. Confronto == :

$$\frac{\Gamma \vdash e1 : T \qquad \Gamma \vdash e2 : T}{\Gamma \vdash e1 == e2 : bool} \quad \text{[Eq]}$$

14. Confronto >= :

$$\frac{\Gamma \vdash e1 : T \qquad \Gamma \vdash e2 : T}{\Gamma \vdash e1 >= e2 : bool} \quad \text{[GreatEq]}$$

15. Confronto <= :

$$\frac{\Gamma \vdash e1 : T \qquad \Gamma \vdash e2 : T}{\Gamma \vdash e1 <= e2 : bool} \quad \text{[LessEq]}$$

16. Contronto > :

$$\frac{\Gamma \vdash e1 : T \qquad \Gamma \vdash e2 : T}{\Gamma \vdash e1 > e2 : bool} \quad \text{[Great]}$$

17. Confronto < :

$$\frac{\Gamma \vdash e1 : T \qquad \Gamma \vdash e2 : T}{\Gamma \vdash e1 < e2 \ : bool} \text{ [Less]}$$

18. Somma :

$$\frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 + e2 \ : int} \text{ [Sum]}$$

19. Sottrazione :

$$\frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 - e2 \ : int} \text{ [Sub]}$$

20. Moltiplicazione:

$$\frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 * e2 \ : int} \text{ [Mult]}$$

21. Divisione:

$$\frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 / e2 \ : int} \text{ [Div]}$$

Per implementare il type checking relativo alle classi inizialmente è stata estesa la funzione *isSubtype,* che come il nome stesso suggerisce, prende in input due tipi e risponde positivamente o negativamente a seconda che il primo sia sotto-tipo del secondo o meno.
Inizialmente questa funzione si limitava a funzionare solamente con i tipi Booleani ed Interi.
È stato aggiunta la nozione di sotto tipaggio tra funzioni (e metodi), *ArrowTypeNode.*
Come da definizione, il tipo di una funzione f viene considerato sottotipo di quello di una seconda funzione t solamente se il tipo di ritorno di f è sottotipo del tipo di ritorno di t, mentre tutti i parametri di f sono super tipi dei parametri di t.
Per quanto riguarda il sotto tipaggio tra classi invece si è per prima cosa controllato che il numero di metodi e campi della sotto-classe fosse almeno pari a quelli della superclasse. Successivamente campi e metodi vengono scanditi e si verifica che valga il sotto-tipaggio tra questi.
In questa funzione troveremo anche due nuovi tipi, ovvero il tipo *void* che viene considerato un sottotipo solamente di se stesso, ed il tipo *bottom* che non è sottotipo di alcun altro tipo, e che viene ritornato ogni qual volta venga riscontrato un errore di tipaggio.

La funzione di tipe check viene lanciata ricorsivamente sull'intera struttura del codice, analizzando il corretto tipaggio di ogni nodo, e ritornando il tipo corretto di ogni

elemento. Come già detto ogni volta che si riscontra un errore di tipaggio viene ritornato un tipo *bottom*. Mentre il tipo associato alla funzione print è *void*.
Per verificare il corretto tipaggio di una classe si verificano due cose, ovvero che tutti i suoi metodi siano correttamente tipati, e nell'eventualità in cui sia presente una super classe si verifica anche il sotto-tipaggio tra le due.
Il controllo dei metodi di una classe invece consiste in due fasi, ovvero nel controllo ricorsivo di tutte le sue dichiarazioni e nel controllo del sotto-tipaggio tra il valore di ritorno del metodo, ed il suo body.
Ogni volta in cui viene chiamato un metodo invece si verifica che il suo tipo sia un *ArrowType* e che tutti i suoi dati in input siano sottotipi dei parametri del metodo.
Infine l'invocazione di un costruttore viene considerata come un metodo della classe stessa, per questo motivo anche il type check viene svolto in modo analogo.

Generazione di codice

Il codice assembler generato prevede le istruzioni standard per una macchina a pila. Eccetto per l'istruzione di `push` (che appunto deve mettere un elemento sulla pila) tutte le altre prendono i loro parametri dalla pila stessa. Non sono stati implementati registri con risultati intermedi per questioni di efficienza in quanto si tratta di una macchina virtuale: la pila è contenuta in un array Java di interi, per cui non c'è una reale differenza tra l'accesso alla pila (su un "disco" che peró è solo nominale) e una variabile della classe che si occupa dell'esecuzione.
La parte più significativa della generazione di codice è certamente la gestione dei record di attivazione delle funzioni. Questa viene implementata con la tecnica classica a stack usando access link (catena statica) e control link (catena dinamica). In particolare:

- Ogni chiamata di funzione inserisce sulla pila il suo record di attivazione.
- Il record di attivazione non è altro che la zona di lavoro della funzione preceduta da un header di informazioni (parametri, dichiarazioni locali, catena statica e dinamica).
- Per risolvere l'utilizzo di una variabile all'interno della funzione si procede risalendo la catena statica fino ad arrivare al blocco in cui la variabile è stata dichiarata (e in cui si trova quindi il suo valore)
- Una volta terminata l'esecuzione della funzione si ripristina il record di attivazione precedente usando la catena dinamica.

Funzioni e variabili vengono inserite all'inizio di ogni record di attivazione, prima del puntatore di catena dinamica.
Il meccanismo complessivo descritto fino ad ora è esattamente quello visto a lezione. Le estensioni che sono state apportate sono quelle necessarie per la gestione di classi e oggetti.

Classi

Si consideri il seguente esempio di definizione di classe:

```
class Test(int x, int y) {
	int sum()
		x + y;
	void printx()
		print(x);
	void printy()
		print(y);
}
```

La classe usa le informazioni raccolte durante la fase di check semantico per generare la propria Vtable e il proprio costruttore.

La Vtable di una classe è la tabella con i puntatori ai suoi metodi. Per il compilatore metodi e funzioni sono la stessa cosa: in questo modo una volta disposti nella Vtable possono essere chiamati usando il loro offset assoluto.
Un metodo rispetto a una funzione deve peró avere accesso ai campi della sua classe; per ottenere questo risultato ogni metodo viene trasformato in una funzione generica il cui primo parametro è l'istanza dell'oggetto su cui viene chiamato, e la risoluzione dei campi avviene facendo riferimento a questo primo parametro. Il metodo `sum()` dell'esempio precedente diventa quindi equivalente a una funzione così costruita:

```
int sum(Test self)
	self.x + self.y;
```

Questo approccio è ispirato a linguaggi di programmazione moderni come Python (che impone di esprimere esplicitamente il parametro self nei metodi) o Go (che permette di chiamare funzioni su determinate strutture come se fossero classi, passando implicitamente l'istanza della struttura come primo parametro; di fatto solamente zucchero sintattico).

In generale l'incapsulamento dei campi di una classe è totale: è proibito far riferimento al campo di una classe al di fuori di essa.

Il costruttore di una classe viene costruito implicitamente sui suoi campi e viene considerato come il primo metodo della classe.
Tutto quello che fa un costruttore è allocare la struttura dei suoi campi nello heap come memoria dinamica. A questo scopo è stata aggiunta l'istruzione assembly `mall n` , che alloca un blocco di dimensione n e mette sulla pila l'indirizzo a cui comincia. Una variabile che contiene un oggetto non è altro che un intero avente come valore suddetto puntatore.

Quando si fa riferimento a un campo all'interno di un metodo per accedervi è necessario aggiungere un'ulteriore istruzione di load word con aggiunta dell'offset relativo per caricare il suo valore contenuto sullo heap.

```
for (int i = 0; i < nestinglevel - entry.getNestLevel() -1; i++)
	getAR += "lw\n";
return "push 1\n" + //Riferimento al primo parametro - self
	"lfp\n" +
	getAR +	//Risalita della catena statica
	"add\n" +
	"lw\n" +	//Carico sulla pila l'indirizzo di self nello heap
	"push " +
	entry.getOffset() + "\n" +  //Offset del campo
	"add\n" +
	"lw\n";
```

Questa è la funzione che genera il codice di risoluzione di riferimento a un campo. Il numero di load word necessari è pari alla differenza tra il nesting level del riferimento e quello della definizione meno uno per i motivi spiegati nella sezione di check semantico: l'entrata in una definizione di classe comporta un incremento del nesting level che poi non trova correlazione in un effettivo record di attivazione sullo stack di esecuzione.

Ereditarietá

Come giá detto è possibile definire classi che ereditano campi e metodi da altre. Questo a livello di generazione di codice si ripercuote nella forma di Vtable e struttura contenente i metodi.
La Vtable di una sottoclasse contiene tutti i metodi della superclasse (sotto forma di puntatori) nello stesso ordine. Se eventualmente ne definisce di nuovi vengono inseriti al di sotto di questi ultimi, e se invece ne sovrascrive si vanno a trovare nella stessa posizione.
In questo modo usando lo stesso offset nella Vtable di superclasse e sottoclasse si riesce ad accedere allo stesso metodo, che puó appartenere alla prima o alla seconda. Per esempio, date le seguenti classi

```
class Test1(int x) {
	void main() print(512);
}
class Test2 implements Test1(int y) {
	void main() print(42);
```

Una istanza di Test2 su cui viene chiamato il metodo main() stamperá 512 o 42 a seconda del tipo della variabile che la contiene, ma in entrambi i casi si cercerá il metodo a offset 1 a partire dall'inizio della Vtable (si ricordi che il metodo a offset 0 è il costruttore).

Anche i campi sono sovrascrivibili, e si usa la stessa tecnica: si mantengono invariati gli offset di quelli che vengono ereditati, sovrascrivendoli dove necessario.

Macchina Virtuale

La macchina virtuale che esegue il codice assembler è contenuta nel file `parser/ExecuteVM.java`. A differenza del compilatore, il codice assembler testuale viene riconosciuto da un parser ma non visitato esternamente con il pattern Visitor.
Le informazioni del programma assembler vengono lette e accumulate direttamente nel parser generato da ANTLR usando delle regole embedded nella grammatica (si veda `parser/SVM.g4`). Si mantiene un array `code` in cui vengono inseriti i codici delle istruzioni macchina man mano che vengono lette.
Dopodichè questo array viene passato alla macchina virtuale che lo esegue, istruzione per istruzione.

Garbage collection

È stato infine implementato un primitivo sistema di Garbage Collection basato su Reference Counting. Il meccanismo si trova nel file `parser/ExecuteVM.java`, ovvero la macchina virtuale che esegue il codice assembler.
Ogni volta che si incontra un'istruzione di allocazione di memoria (mall) si sta creando un oggetto nello heap. Si inserisce quindi in un dizionario (`heapReferences`) l'indirizzo puntato nell'heap indicizzato con l'indirizzo nello stack in cui viene salvato. Si salva poi in un altro dizionario (`garbageCollector`) un oggetto `heapBlock` che mantiene il numero di riferimenti presenti sullo stack a tale blocco di memoria, indicizzato con l'indirizzo nello heap a cui si trova.
Ogni qualvolta un indirizzo sullo stack presente nel primo dizionario viene acceduto e duplicato (ovvero inserito sulla pila) si incrementa il contatore di riferimenti a quel blocco. Quando uno di questi riferimenti sulla pila viene rimosso (operazione di pop) il contatore viene invece decrementato. Quando raggiunge lo 0 la macchina virtuale lo dealloca, avendo cura di decrementare allo stesso modo anche riferimenti ad altri oggetti presenti sullo heap.

Quando un blocco viene deallocato torna libero di essere utilizzato per allocazioni future. Non è preso alcun provvedimento contro il fenomeno della frammentazione esterna, per cui eventualmente l'heap è destinato a riempirsi comunque.

Anche se in teoria il Garbage Collector permette la riallocazione di memoria non più utilizzata, le limitazioni del linguaggio di programmazione fanno si che ció non avvenga mai: se infatti un oggetto viene deallocato è in seguito alla chiusura di un record di attivazione, ed essendo il numero di istruzioni in un programma fool limitato a una, in questo caso sta anche terminando, e la memoria liberata non verrá più utilizzata.

Nonostante ció chiamando l'interprete Fool con l'opzione -d (debug) è possibile osservare dei messaggi di debug su allocazione e deallocazione di oggetti nello heap.

Conclusioni

È stato correttamente implementato un compilatore e un interprete che, combinati, permettono di eseguire codice di un linguaggio di programmazione "giocattolo". Il lavoro ha permesso agli autori di capire meglio ed esercitare alcuni dei meccanismi di base della teoria dei linguaggi di programmazione.
Il linguaggio, per quanto ristretto e semplificato, permette di calcolare funzioni di base quali il fattoriale utilizzando la ricorsione.
Il linguaggio è in realtá Turing completo, essendo in grado di rappresentare le funzioni ricorsive generali:

1. **Funzioni costanti**: FOOL è banalmente in grado di esprimere le costanti numeriche.

2. **Funzione successore:** una possibile implementazione della funzione successore è la seguente:

```
int succ(int x)
	x + 1;
```

3. **Proiezione**: data una serie di parametri è banale restituirne l'i-esimo.

4. **Chiusura per composizione:** in FOOL è possibile comporre due funzioni f e g su input x con la semplice chiamata `f(g(x))`.

5. **Chiusura per ricorsione primitiva**: Siano h e g due funzioni ricorsive generali giá definite in un programma FOOL. La ricorsione primitiva su di esse è facilmente esprimibile come:

```
int rec(int counter, int input)
	if counter == 0 then {
		g(input)
	} else {
		h(counter - 1, input, rec(counter - 1, input))
	};
```

6. **Chiusura per minimizzazione**: Sia g una funzione ricorsiva generale giá definita in un programma FOOL. La minimizzazione limitata su di essa è facilmente esprimibile come:

```
int min(int input, int check)
	if g(input) == 0 {
		check
```

```
} else {
	min(input, check + 1)
};
```

L’applicazione di questa dimostrazione si puó trovare nell’ultimo test, `test_special_4_turing.fool`, che cerca il minimo numero tale che la sua sottrazione a 42 sia 0 (ovviamente, 42).