Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Background

1.1 Concetti base del corso

- Parse Tree (Anche chiamato Sintax Tree in questo corso): struttura ad albero che rappresenta la struttura sintattica completa di una stringa di input, inclusi tutti i passaggi nella derivazione secondo le regole di una grammatica formale. Ogni nodo dell'albero corrisponde a un simbolo grammaticale (terminale o non terminale) e le foglie dell'albero corrispondono ai terminali (token) della stringa di input. **In soldoni è utile per generare il codice**

- Abstract Syntax Tree (AST): struttura sintattica astratta di una stringa di input, enfatizza la struttura logica e il significato del codice piuttosto che i dettagli sintattici. È una versione semplificata del Parse Tree che rimuove i dettagli implementativi e di formattazione non rilevanti per l'analisi o l'esecuzione del codice. I nodi dell'AST rappresentano costrutti sintattici essenziali come istruzioni, espressioni, dichiarazioni di variabili, ecc., che sono più significativi per l'interpretazione e l'esecuzione del codice. **In soldoni è utile per rappresentare visualmente la struttura del linguaggio e permettere al programmatore di fare delle analisi**

- Struttura blocchi Let-In: costruzione sintattica che si trova in alcuni linguaggi di programmazione funzionali. Questa costruzione consente di definire e utilizzare variabili locali all'interno di uno specifico blocco di codice. Un concetto fondamentale dei blocchi Let-In è il "nesting level" ovvero il livello di annidamento (dentro a quanti blocchi let-in l'attuale blocco è annidato), se il blocco non è annidato il nesting level è 0.

  La struttura tipica di un blocco "let-in" è la seguente:

  let

```
    <dichiarazioni di variabili locali>
```

in

```
    <espressione principale>
```

Esempio nel linguaggio FOOL:

```
ProgLetIn
  Var: x
    IntType
    Minus
      Int: 3
      Int: 1
  Var: b
    BoolType
    Bool: true
  Print
    If
      Not
        Equal
          Id: x at nestinglevel 0
            STentry: nestlev 0
            STentry: type
              IntType
            STentry: offset -2
          Int: 4
      Int: 1
      Int: 0
```

Capitolo 2

Analisi

2.1 Requisiti

Si vuole realizzare un compilatore per il Functional and Object Oriented Language (FOOL), estendendo il progetto che abbiamo realizzato in laboratorio, producendo un linguaggio che possa essere eseguito dalla Stack Virtual Machine (SVM) sviluppata dal professor Mario Bravetti in laboratorio.

Il progetto dovrà introdurre nel linguaggio:

- gli operatori "<=", ">=", "||", "&&", "/", "-" e "!", con stesso significato che hanno in C/Java .
- estensione con object-orientation con ereditarietà e ottimizzazioni

2.2 Risorse

2.2.1 Stack Vitrual Machine

Il professore rende disponibile la stack vitual machine, da non modificare.

La SVM decreta il linguaggio macchina, i cui comandi sono:

- PUSH n=INTEGER: aggiunge un valore intero allo stack. Il valore da inserire nello stack è specificato dall'intero che segue il comando PUSH.
- PUSH l=LABEL: simile a PUSH, ma invece di inserire un valore intero nello stack, inserisce l'indirizzo di un'etichetta (label).
- POP: rimuove il valore in cima allo stack.
- ADD: somma i due valori in cima allo stack e inserisce il risultato nello stack.

- SUB: sottrae il valore in cima allo stack dal valore successivo nello stack e inserisce il risultato nello stack.
- MULT: moltiplica i due valori in cima allo stack e inserisce il risultato nello stack.
- DIV: divide il valore successivo nello stack per il valore in cima allo stack e inserisce il risultato nello stack.
- STOREW: memorizza il valore in cima allo stack in una locazione di memoria specifica.
- LOADW: carica il valore da una locazione di memoria specifica e lo inserisce nello stack.
- l=LABEL COL: definisce un'etichetta (label) associata alla posizione corrente nel codice.
- BRANCH l=LABEL: esegue un salto incondizionato all'istruzione con l'etichetta specificata.
- BRANCHEQ l=LABEL: esegue un salto condizionato all'istruzione con l'etichetta specificata se i due valori in cima allo stack sono uguali.
- BRANCHLESSEQ l=LABEL: esegue un salto condizionato all'istruzione con l'etichetta specificata se il secondo valore nello stack è minore o uguale al primo.
- JS: esegue un salto alla subroutine (funzione).
- PRINT: stampa il valore in cima allo stack.
- HALT: interrompe l'esecuzione del programma.

2.2.2 Linguaggio FOOL

Il professore ha messo a disposizione le classi che creano il traduttore del linguaggio FOOL (sulle qauli è richiesto di apportare modifiche) che decretano i comandi del linguaggio che si mettono a disposizione dell'utente e li traduce in linguaggio macchina che verrà eseguito dalla SVM.

Le classi seguono il pattern Visitor e sono generate tramite la libreria ANTLR.

Il linguaggio viene letto e tradotto eseguendo le classi nel seguente ordine:

- FOOL.g4: qui vengono dichiarate le regole del lexer, dove vengono dichiarati tutti i simboli (token) possibili del linguaggio e vengono dichiarate le regole del parser, che prende in input i Token e ne genera una genrarchia chiamata AST (abstract syntax tree). La libreria ANTLR produce le classi "FOOLLexer" e "FOOLParser" nel package "gen".

- ASTGenerationSTVisitor: genera un AST del linguaggio a partire dal "ParseTree" (che viene generato dalla classe "FOOLParser").

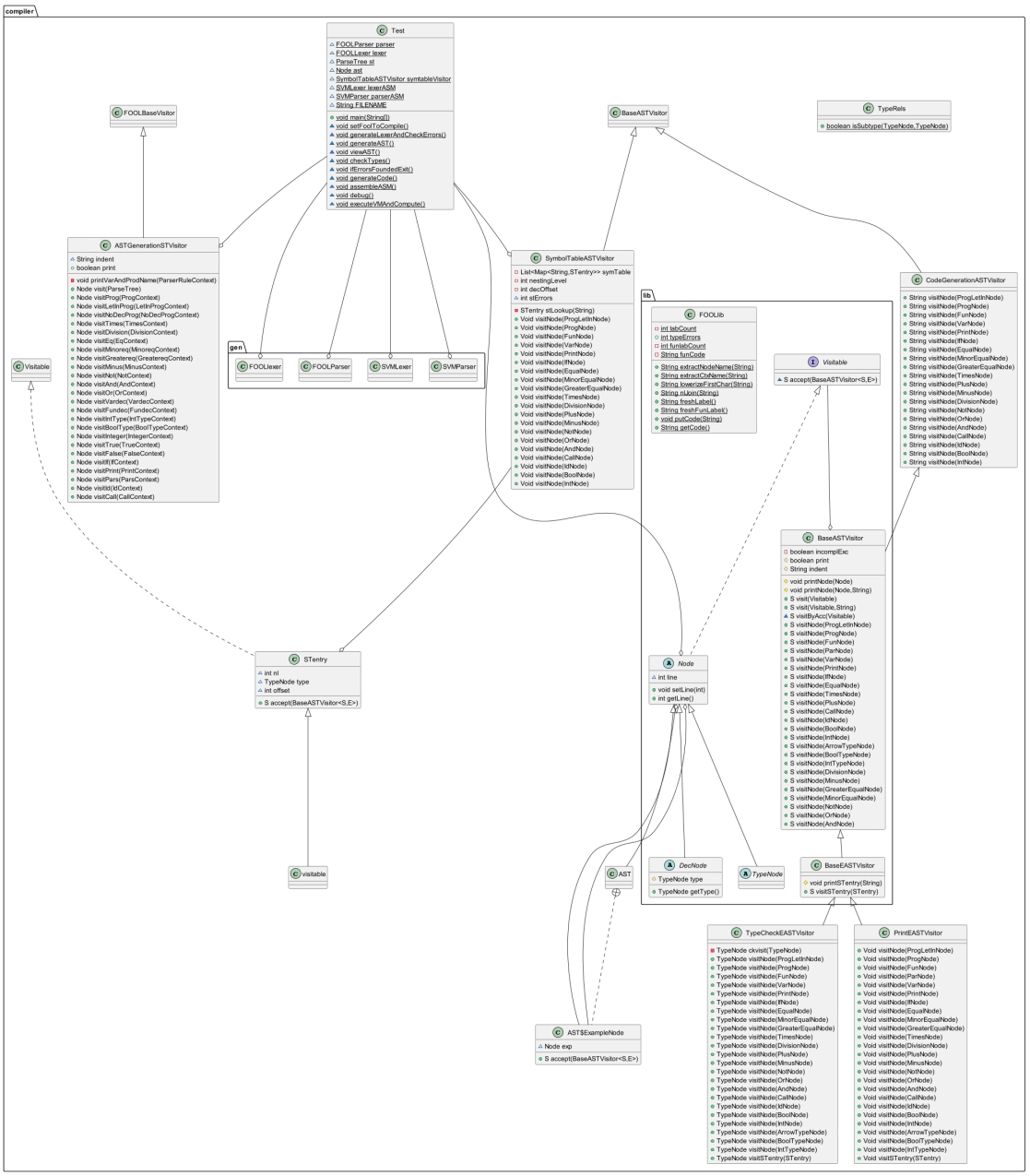
- SymbolTableASTVisitor: trasforma l'AST in una struttura dati del tipo "let-in".

- PrintEASTVisitor: stampa la struttura "let-in" del codice.

- TypeCheckEASTVisitor: visita il nodo che gli viene passato tramite il metodo "Visit(Node node)" e ne restituisce il tipo.

- CodeGenerationASTVisitor: traduce i nodi dell'AST in codice da eseguire linguaggio macchina (nel nostro caso in SVM).

Figura 2.1: UML del progetto FOOL originale



Capitolo 3

Risoluzione

3.1 Validazione

3.1.1 Aggiunta degli operatori

L'obbiettivo di questa sezione è quello di aggiungere gli operatori "<=", ">=", "||", "&&", "/", "-" e "!" al linguaggio, con stesso significato che hanno in C/Java.

L'implementazione a livello generale è la medesima per ogni operatore, quindi nella relazione mi limiterò a descrivere l'implementazione di uno dei nodi presa arbitrariamente. In questo caso ho scelto il simbolo ">=".

- FOOL.g4: il primo file da modificare, per rigenerare il package "gen" e aggiornarlo per ricreare il lexer e il parser che prenda in considerazione i nuovi simboli. In questo caso ci basta aggiungere l'elemento nella grammatica dell'exp nell'ordine corretto:

```
exp     : exp TIMES exp #times
        | exp DIVISION  exp #division
        | exp PLUS  exp #plus
        | exp MINUS  exp #minus
        | exp EQ  exp   #eq
        | exp MINOREQ exp #minoreq
        | exp GREATEREQ exp #greatereq //aggiunto
```

  exp naturalmente rappresenta un nodo.

- AST: Inserire la classe nell'AST che rappresenta il nodo preso in considerazione:

Listato 3.1: Classe GreaterEqualNode

```
1  public static class GreaterEqualNode extends Node
       {
2    final Node left;
3    final Node right;
4    GreaterEqualNode(Node l, Node r) {left = l;
       right = r;}
5
6    @Override
7    public <S,E extends Exception> S accept(
       BaseASTVisitor<S,E> visitor) throws E {return
        visitor.visitNode(this);}
8  }
```

- BaseASTVisitor: Inserire il metodo che permette la visita nel BaseASTVisitor per il nodo preso in considerazione, per generare l'st:

Listato 3.2: Metodo visitNode(Node node) nel caso di GreaterEqualNode

```
1 public S visitNode(GreaterEqualNode n) throws E {
2   throw new UnimplException();
3 }
```

- ASTGenerationSTVisitor: genera un AST a partire dall'st del nodo preso in considerazione:

Listato 3.3: Metodo visitGreatereq

```
1 @Override
2 public Node visitGreatereq(GreatereqContext c) {
3   Node n = new GreaterEqualNode(visit(c.exp(0)),
      visit(c.exp(1)));
4   n.setLine(c.GREATEREQ().getSymbol().getLine());
5   return n;
6 }
```

- SymbolTableASTVisitor: visita il nodo e li trasforma in codice di tipo "let-in". Se il codice non è una foglia, la funzione che lo visita descrive in che altri nodi proseguire a lui collegati, se è terminale prende in input un valore intero o booleano.

Listato 3.4: Metodo visitNode(GreaterEqualNode node), non terminale

```
@Override
public Void visitNode(GreaterEqualNode n) {
  if (print) printNode(n);
  visit(n.left);
  visit(n.right);
  return null;
}
```

Listato 3.5: Metodo visitNode(IntNode node), terminale

```
@Override
public Void visitNode(IntNode n) {
  if (print) printNode(n, n.val.toString());
  return null;
}
```

- PrintEASTVisitor: stampa la struttura "let-in" del codice FOOL, passando dal singolo nodo i metodi funzionano in modo molto simile a quelli della classe SymbolTableASTVisitor.
- TypeCheckEASTVisitor: visita il nodo che gli viene passato tramite il metodo "Visit(Node node)" e ne restituisce il tipo.

Listato 3.6: Metodo visitNode(GreaterEqualNode node), non terminale

```
@Override
public TypeNode visitNode(GreaterEqualNode n) throws
    TypeException {
  if (print) printNode(n);
  TypeNode l = visit(n.left);
  TypeNode r = visit(n.right);
  if ( !(isSubtype(l, r) || isSubtype(r, l)) )
    throw new TypeException("Incompatible types in
       equal",n.getLine());
  return new BoolTypeNode();
}
```

- CodeGenerationASTVisitor: traduce i nodi dell'AST in codice da eseguire linguaggio macchina (nel nostro caso in SVM).

Listato 3.7: Metodo visitNode(GreaterEqualNode node), non terminale

```
@Override
public String visitNode(GreaterEqualNode n) {
  if (print) printNode(n);
  final String l1 = freshLabel();
  final String l2 = freshLabel();
  return nlJoin(
      visit(n.left),
      visit(n.right),
      "push " + 1,
      "sub",
      "bleq " + l1,
      "push " + 1,
      "b " + l2,
      l1 + ":",
      "push " + 0,
      l2 + ":"
  );
}
```

Dal punto di vista del Design delle classi il codice rimane invariato

3.1.2