Progetto di Compilatori e Interpreti

CdLM Università di Bologna

Andrea Gurioli
Giovanni Pietrucci
Mario Sessa

Anno Accademico 2020/2021

Contents

Installazione

1.1 IntelliJ IDEA

1.1.1 Installazione ANTLR

Prima di poter installare il nostro compilatore sull'IDE di IntelliJ IDEA, abbiamo bisogno di un plugin: ANTLR 4.9.2 o versioni successive. Per farlo, bisogna seguire i seguenti passi:

1. Aprire IntelliJ IDEA IDE
2. Cliccare Plugins nella barra laterale
3. Nella barra di ricerca del Marketplace cercare "ANTLR"
4. Cliccare su Install alla voce "ANTLR v4"

In seguito ai passaggi sopra elencati abbiamo installato correttamente ANTLR. Per poterlo utilizzare, abbiamo bisogno di riavviare l'IDE.

1.1.2 Caricamento Compilatore

Per poter eseguire il compilatore, abbiamo bisogno di importare all'interno di IntelliJ IDEA il progetto completo. Per farlo bisogna:

1. Scaricare il progetto da GitHub all'indirizzo https://github.com/kode-git/project-compiler-antlr4
2. Aprire IntelliJ IDEA
3. Andare alla voce "Projects" nella barra laterale
4. Cliccare su "Open"
5. Scegliere il progetto
6. Cliccare su "Aprire" nell'interfaccia visualizzata

In seguito, possiamo visualizzare l'intero progetto sul nostro IDE. Per poter visualizzare il codice che si intende compilare, basta accedere al file "code.slp" nel Project Explorer dell'IDE. Per poter eseguire la compilazione del codice bisogna selezionare il file main/src/mainPackage/Test.java e, cliccando il tasto destro, selezionare la voce "Esegui 'Test.main()' ". Nella console di output di Java, si può vedere il risultato della compilazione.

1.2 Eclipse

1.2.1 Installazione ANTLR

Prima di poter installare il nostro compilatore sull'IDE di IntelliJ IDEA, abbiamo bisogno di un plugin: ANTLR 4.9.2 o versioni successive. Per farlo, bisogna seguire i seguenti passi:

1. Aprire Eclipse IDE
2. Cercare nel menu la voce Help, e clicccare su "Ecplipse Marketplace"
3. Cercare "Antlr" nella barra di ricerca del marketplace
4. Installare il plugin AntlrDT 4.2.1 o versioni successive
5. Installare il plugin ANTLR 4 IDE

Si presume che, durante l'installazione di Eclipse, si sia selezionata l'installazione per Java Developers.

1.2.2 Caricamento Compilatore

In seguito ai passaggi sopra elencati abbiamo installato correttamente ANTLR. Per poterlo utilizzare, abbiamo bisogno di riavviare l'IDE. Successivamente, selezionare "File" dalla barra principale, cliccare su "Apri Progetto", selezionare il progetto precedentemente scaricato, cliccare su "Apri".

Per visualizzare o modificare il codice da compilare, andare su "code.slp" visualizzato nel Project Explorer di Eclipse. Per poter eseguire il codice, andare su main/src/mainPackage/Test.java e, cliccando col tasto destro, selezionare le voci "Run As" e "Java Application".

Analisi Lessicale

2.1 Gestione Errori Lessicali

Le scelte implementative fatte nell'analisi lessicale si basano principalmente su come gestire gli errori, poichè l'implemetazione del lexer è stata fatta in maniera autogenerata grazie ai tool di generazione forniti da ANTLR v4.

Dal **lexer** possiamo utilizzare alcuni metodi di rimozione degli errori dal codice dato in input e, allo stesso tempo, richiamare il nostro **listener** implementato per gestirli nel caso si presentassero nel codice.

Abbiamo scelto di utilizzare un listener, implementato come segue:

```
1  public class ThrowingErrorListener extends BaseErrorListener {
2
3      public static final ThrowingErrorListener INSTANCE = new
       ThrowingErrorListener();
4
5      @Override
6      public void syntaxError(Recognizer<?, ?> recognizer, Object
       offendingSymbol, int line, int charPositionInLine, String msg,
       RecognitionException e)
7              throws ParseCancellationException {
8          throw new ParseCancellationException("line " + line + ":" +
       charPositionInLine + " " + msg);
9      }
10 }
```

Listing 2.1: Classe ThrowingErrorListener

ThrowingErrorListener si occuperà quindi di catturare l'evento di rilevamento dell'errore creato dal lexer e, successivamente, stampare la linea e il carattere errato. In più, stamperà nel medesimo messaggio anche una sequenza di caratteri terminali validi che renderebbero il codice lessicalmente corretto.

Analisi Sintattica

3.1 Gestione Visitor per la navigazione nell'AST

Come nell'analisi lessicale, anche l'analisi sintattica ha i vari elementi implementati che sono stati generati automaticamente dai tool di ANTLR.

Le uniche modifiche rispetto al codice generato, sono state all'interno del visitor in cui abbiamo indirizzato la visita dell'AST a tipi di nodi differenti dipendenti da alcune caratteristiche del contesto dato in input.

```
1     @Override
2     public Node visitType(SimpLanPlusParser.TypeContext ctx) {
3         if(ctx.getText().equals("int"))
4             return new IntTypeNode();
5         else if(ctx.getText().equals("bool"))
6             return new BoolTypeNode();
7         if(ctx.type()!= null){
8             GenericTypeNode typeNode = (GenericTypeNode) visit(ctx.type());
9             return new PointerTypeNode( typeNode);
10        }
11        return null;
12    }
```

Listing 3.1: Visita di un nodo Type differenziando il nodo da visitare a secondo se sia intero o booleano oppure un puntatore

La visita dei vari nodi, in seguito alle modifiche, ha fatto si che vi sia una visita corretta dell'AST ed eliminanndo disambiguità presenti all'interno del codice autogenerato.

Analisi Semantica

4.1 Symbol Table

La **symbol table** è stata implementata all'interno di una classe Environment ed è rappresentata da un ArrayList di HashMap che è strutturata come una collezione di coppie chiavi-valore in cui le **chiavi** sono delle stringhe raffiguranti gli identificatori di una entry e come **valore** ha un oggetto STentry che mantiene al suo interno le seguenti informazioni:

1. **Nesting Level** - Corrispondente all'indice dell'HashMap a cui appartiene nell'ArrayList che rappresenta la symbol table

2. **Type** - Rappresenta il tipo del nodo o dell'elemento a cui fa riferimento l'entry secondo l'AST del parser

3. **Offset** - Utilizzato per alcuni tipi di elementi per accedere all'indirizzo di memoria in cui vi è un possibile valore.

4. **pointerCounter** - Che rappresenta il livello del puntatore di un valore. Nel caso non sia un puntatore, tale valore è impostato per default a 0.

5. **Collezione degli stati degli effetti** - Implementato come un array di grandezza pari a pointerCounter + 1 in cui mantenere lo stato degli effetti dell'entry. Nel caso del puntatore, l'array avrà una grandezza maggiore di 1 poichè avrà più possibili riferimenti e cambiamenti di stato in relazione al livello del puntatore a cui si fa riferimento nel codice.

6. **Riferimento alla funzione** - Nel caso l'id corrisponda ad una funzione, tale variabile è un riferimento a DecFunNode corrispondente alla dichiarazione della funzione da innvocare nel caso di **controlli semantici su CallNode**.

7. **Hashmap di propagazione** - Utilizzato per la propagazione di effetti su puntatori in cui mantiene come chiave un id del puntatore a cui propagare la cancellazione della entry attuale e come valore un intero corrispondente al livello di referenziazione del puntatore a cui è stato assegnata l'entry corrente.

La classe **Environment**, al di là della **symbol table**, mantiene anche una variabile di instanza intera che rappresenta il **nestingLevel corrente** in relazione al nodo dell'AST che si sta visitando per il controllo semantico.

L'Environment è istanziato inizialmente nella classe Test.java e passato ad un metodo **checkSemantic** che andrà ad eseguire i controlli semantici sugli elementi dell'AST precedentemente creato.

La symbol table viene gestita in relazione al tipo di elemento che si sta visitando nell'AST. In seguito andremo a spiegare come sono state implementate le principali operazioni per la gestione della symbol table.

4.1.1 Operazione di entryScope

L'operazione di **EntryScope** è raffigurata dal metodo **addTable** di **Environment**

```
1  public void addTable(HashMap<String, STentry> hm){
2    this.nestingLevel++;
3    this.symTable.add(hm);
4  }
```

Listing 4.1: Metodo addTable nella classe Environment

Abbiamo semplicemente aumentato il **nestingLevel dell'Environment** e aggiunto un **nuovo HashMap** alla **Symbol Table**.

4.1.2 Operazione di exitScope

L'operazione di **ExitScope** è raffigurata dal metodo **removeTable** di **Environment**

```
1  public void removeTable(){
2    this.symTable.remove(this.nestingLevel);
3    this.nestingLevel--;
4  }
```

Listing 4.2: Metodo removeTable nella classe Environment

Abbiamo semplicemente diminuito il **nestingLevel dell'Environment** e **rimosso l'hashmap del Nesting Level corrente**.

4.1.3 Operazione di lookup

L'operazione di **lookup** è stata implementata nel metodo lookup di Environment.

```
1  public STentry lookup(int nestingLevelIntern,String id){
2    STentry tmp;
3    for(tmp = null; nestingLevelIntern >= 0 && tmp == null; tmp = (STentry)
     ((HashMap)this.symTable.get(nestingLevelIntern--)).get(id)) {
4      // lookup
5    }
6    if (tmp == null) {
7      return null;
8    } else {
9      return tmp;
10
11   }
12
13 }
```

Listing 4.3: Metodo lookup nella classe Environment

Tale metodo prende in input l'**id dell'entry** da dover leggere, se presente, e il **nesting level** in cui è stata dichiarata. Nel caso la **entry** non sia presente, il metodo torna un valore **null**, altrimenti ritorna l'oggetto **STentry** desiderato.

Da specificare, che la risalita della catena visita tutti i livelli o Hashmap della symbol table a partire dal nesting level specificato nei parametri del metodo fino al livello del blocco della radice, fermandosi al primo matching dell'id.

4.1.4 Aggiunta new Entry

Il metodo corrispondente a tale operazione è **addEntry** che aggiunge una **entry** all'interno dell'**Hashmap** all'indice specificato dal **nesting level** preso come parametro.

```
1  public SemanticError addEntry(int nestingLevelIntern, String id, STentry
    entry){
2    HashMap<String, STentry> hm = (HashMap)symTable.get(nestingLevelIntern);
3    if(hm.put(id, entry)!=null) return new SemanticError( "Environment Error
    : entry " + id + " already declared");
4    return null;
5  }
```

Listing 4.4: Metodo addEntry inserisce un'entry nell'Hashmap specificandone l'indice

Il metodo effettua anche un **controllo semantico** andando a verificare se una **entry**, con lo stesso **id** specificato tra i parametri del metodo, esiste già all'interno della **symbol table** nel livello di nesting dato in input.

4.2 Type Checking

Le scelte progettuali sul **Type Checking** si limitano sul come effettuare il controllo di tipo per assegnazioni, passaggio di parametri, valori di ritorno di funzioni, corrispondenza tra valore di ritorno di una funzione e valore ritornato da un invocazione.

4.2.1 Definizione dei tipi

Abbiamo utilizzato un metodo della classe **SimpLanLib** chiamato **isSubtype** che prende come parametri due nodi e vede se essi appartengono o meno alla stessa classe.

Inoltre, ogni classe implementava un metodo dell'interfaccia **Node**, **typeCheck** che ritornava il tipo del nodo. Il typeChecking è invocato secondo l'ordine di visita dell'AST generato dal codice dato in input. Il tipo del nodo è anch'esso una classe che implementa l'intefaccia Node e può essere:

1. **IntTypeNode** : Nodo che rappresenta il tipo intero
2. **BoolTypeNode**: Nodo che rappresenta il tipo booleano
3. **PointerTypeNode** : Nodo che rappresenta un puntatore ad tipo intero o ad un tipo booleano
4. **VoidNode** : Nodo che rappresenta il tipo vuoto

Per poter distinguere i due casi del **PointerTypeNode**, è stato implementata una versione parametrica in cui si ha un solo tipo T che implementa un interfaccia **GenericTypeNode**; quest'ultima è un'interfaccia di markup implementata dalle classi **IntTypeNode** e **BoolTypeNode** ma non in **VoidNode**.

```
1  public class PointerTypeNode <T extends GenericTypeNode> implements Node,
    GenericTypeNode, Cloneable {
2
3    private T val
4    public PointerTypeNode (T n) {
5        val=n;
6    }
```

```
7       public PointerTypeNode () {val=null; }
8   }
```

Listing 4.5: Firma della classe PointerTypeNode

Per ricoprire i casi di puntatori a puntatori, abbiamo implementato l'interfaccia **GenericTypeNode** anche per la classe **PointerTypeNode**.

I nodi che fanno riferimento ad una dichiarazione di funzione o ad una variabile, ritornano rispettivamente il valore di ritorno della funzione e il tipo della variabile.

4.2.2 Controllo Tipi Su Variabili

Nei costrutti operazionali binari, il controllo di tipo si limita a controllare se il tipo delle due variabili è supportato dall'operazione (solo tipo interi per operazioni aritmetiche, solo booleane per **or** e **and** logico ed entrambe per gli altri casi).

```
1       @Override
2       public Node typeCheck() {
3           if (!(SimpLanlib.isSubtype(left.typeCheck(), new IntTypeNode()) &&
4                   SimpLanlib.isSubtype(right.typeCheck(), new IntTypeNode())))
        {
5               System.out.println("Sum Error: bad operand types for binary
    operator '+'");
6               System.exit(0);
7           }
8           return new IntTypeNode();
9       }
```

Listing 4.6: typeCheck di BinExpSumNode che accetta solamente due nodi aventi il typeCheck uguale a IntTypeNode

Nelle classi degli **statement** che rappresentano operazioni proprie e prendono nel costrutto qualche voce della grammatica che può corrispondere ad una variabile come **print** e **delete**, si ritorna un tipo **VoidNode** dato che il tipo del **typeCheck** non viene utilizzato in nessun altro nodo padre dell'AST generato.

Al contrario, nodi che rappresentano **statement** come **RetNode** e **CallNode** ritornano nel typeCheck un tipo che può essere utilizzato da un nodo padre come, ad esempio, l'assegnazione o la somma su invocazioni di funzioni.

4.2.3 Controllo Tipi Su Funzioni

I controlli per le funzioni si dividono tra:

1. **Dichiarazione di funzione** - Gestendo il controllo di tipi dentro la classe **DecFunNode**
2. **Invocazione di fuzione** - Gestendo il controllo di tipi dentro la classe **CallNode**

Controllo di tipi nelle dichiarazioni di funzione

Il controllo di tipi all'interno della dichiarazione di funzione, gestisce solamente il controllo di tipo tra il tipo della funzione e il tipo del valore di ritorno dal suo corpo.

```
1           @Override
2       public Node typeCheck() {
3
4           // case: void with return in function
5           if(this.type instanceof VoidNode && this.block.checkRet()) {
```

```
            // return error - case of return in void func
            System.out.println("Function Declaration Error: function " + id
    + " is void and can't have return statement");
            System.exit(0);
        }

        // here if type is not void, need to check if return statement is
    present
        else if(!this.block.checkRet() && !(this.type instanceof VoidNode)){
            // no return statement in type != void
            System.out.println("Function Declaration Error: function " + id
    + " don't have return statement");
            System.exit(0);
        }


        // here type != void and there is return as block.typeCheck()
        if (!(SimpLanlib.isSubtype(block.typeCheck(), type))) {
            System.out.println("Function Declaration Error: wrong return
    type for function " + id);
            System.exit(0);
        }

        return block.typeCheck();
     }
```

Listing 4.7: TypeCheck nella classe DecFunNode

Nello specifico, da come si può notare nello snippet di codice, abbiamo inserito una funzione **checkRet** che verifica se il corpo della funzione abbia o meno i **return** per effettuare il controllo con il tipo **void**. Secondo tale scelta progettuale, **non è possibile avere dei return all'interno del blocco della funzione se il tipo di quest'ultimo è void**.

Controllo di tipi nelle invocazioni di funzione

Nella classe **CallNode** andiamo a controllare tramite un nodo di riferimento a funzioni chiamato **ArrowTypeNode**, se l'entry corrispondente all'id invocato corrisponde effettivamente ad una funzione. Successivamente, tramite **ArrowTypeNode** che fa riferimento alla dichiarazione di funzione invocata, andiamo a controllare:

1. Se il numero degli argomenti della dichiarazione di funzione corrisponde effettivamente al numero dei parametri dell'invocazione della funzione

2. Se ogni parametro dell'invocazione è dello stesso tipo dell'argomento corrispondente nella dichiarazione.

Di seguito mostriamo lo snippet di codice corrispondente al **typeCheck** dell'invocazione a funzione definito all'interno della classe **CallNode**:

```
    public Node typeCheck () {  //

      ArrowTypeNode t=null;
      if (entry.getType() instanceof ArrowTypeNode) t=(ArrowTypeNode) entry.
    getType();
      else {
          System.out.println("Call Error: invocation of a non-function "+id)
    ;
          System.exit(0);
```

```
8       }
9       ArrayList<Node> p = t.getArgList();
10
11      // Checking of number of arguments equals to the number of parameters
    in DecFun
12      if ( !(p.size() == exp.size()) ) {
13          System.out.println("Call Error: wrong number of parameters in the
    invocation of "+id);
14          System.exit(0);
15      }
16      for (int i=0; i<exp.size(); i++) {
17          if (!(SimpLanlib.isSubtype((exp.get(i)).typeCheck(), ((ArgNode)p.
    get(i)).getType()))) {
18              System.out.println("Call Error: wrong type for " + (i + 1) + "
    -th parameter in the invocation of " + id);
19              System.exit(0);
20          }
21      }
22      return t.getRet();
23  }
```

Listing 4.8: TypeCheck nella classe CallNode

Alla fine si da in output il return value della funzione per possibili controlli sulla gestione dell'uso del tipo del valore di ritorno.

4.3 Controllo per errori semantici

Per quanto riguarda gli errori semantici che non riguardano il controllo sui tipi, ogni classe che rappresenta un nodo dell'AST ha implementato un metodo **checkSemantics** proprio dell'interfaccia **Node**. Il metodo prende **Environment** e, nel caso di nodi di dichiarazione come DecFunNode e DecVarNode, prende anche un offset come parametri e restituisce un **array di SemanticError** in cui, se ha errori, verranno stampati successivamente alla visita semantica dell'AST in maniera cumulativa (come nel linguaggio C).

4.3.1 Controllo Nell'Uso Di Variabili e di Funzioni

Quando si va ad utilizzare una **entry** contenuta all'interno della **symbol table**, andiamo sempre ad effettuare un controllo tramite **lookup** nell'**Environment** passato come parametro nel **checkSemantics** per verificare se l'entry esista o meno. Nel caso non esista, si andrà ad utilizzare una variabile o una funzione precedentemente non dichiarata e quindi si avrà la generazione di un SemanticError inserito all'interno dell'array degli errori semantici dato in output.

Una delle decisioni progettuali sul **checkSemantics** riguardo il controllo della presenza o meno di una variabile (nello specifico, della entry corrispondente) nella **symbol table** è quella della divisione del controllo semantico tra variabili intere o booleane e variabili di puntatori. Tale problema è proprio dei nodi **LhsNode** in cui hanno un **LhsVar** come variabile di instanza che rappresenta l'id della variabile (nel caso in cui non fosse un puntatore) oppure un **LhsNode** nel caso sia un puntatore.

Per poter prelevare l'id del puntatore del **LhsNode** corrente, andiamo a sfruttare un metodo **getId()** che prende l'id dalla catena dei riferimenti del puntatore come fatto nel seguente snippet di codice:

```
1       public String getId(){
```

```
        T value = lhVar;
        if(value instanceof String){
            return (String) value;
        } else {
            while ((value instanceof LhsNode)) {
                value = (T) ((LhsNode<?>) value).getLhVar();
            }
            return (String) value;
        }
    }
```

Listing 4.9: Metodo di LhsNode per prelevare l'id corrispondente ad un puntatore tramite discesa dei riferimenti

La discesa dei riferimenti va a prendere l'id dalla foglia dei riferimenti del puntatore corrispondente ad LhsNode attuale e lo ritorna al nodo corrente cosi da poter effettuare un lookup sull'id del puntatore a qualsiasi livello di referenziazione.

Per quanto riguarda invece il checkSemantics di un CallNode, si controlla se l'id della funzione è presente nella symbol table e, nel caso sia al suo interno, controlliamo che il reference della sua entry non sia null. Nel caso lo sia, l'entry non fa riferimento ad una funzione ma ad una variabile, portando cosi alla rilevazione di un errore semantico.

4.3.2 Controllo Dichiarazioni Multiple

Per quanto riguarda invece il checkSemantic per le dichiarazioni di funzioni (DecFunNode) e di variabili (DecVarNode) si avrà l'invocazione del metodo addEntry di Environment implementato come illustrato nel paragro 4.1.4.

Nei checkSemantics di DecFunNode e DecVarNode, si ha anche una versione in cui, in aggiunta ad Environment, si ha come parametro anche l'offset corrente che viene aggiornato in seguito alla dichiarazione ed utilizzato successivamente per la fase di code generation.

4.4 Gestione Effetti

La gestione degli effetti è stata divisa, per modularità, dal metodo checkSemantic. Il metodo corrispondente a tale fase è checkEffects definita nell'interfaccia Node ed implementata in ogni classe usata nell'AST. Tale metodo viene richiamato alla fine della procedura del checkSemantics solamente se, nella prima parte del controllo semantico, non vi sono stati errori.

Gli effetti corrispondente ad una entry della symbol table, come descritto precedentemente, sono mantenuti in un array di effetti nella classe STentry che è pari a grandezza 1, nel caso in cui sia una variabile intera, booleana o una funzione e pari al pointerCounter + 1 nel caso dei puntatori.

I valori dell'array possono essere:

- **0** - Corrispondente all'effetto bottom, ossia quando si ha solamente una variabile dichiarata ma non inizializzata.
- **1** - Corrispondente all'effetto rw, ossia quando ad una variabile gli viene assegnato (write) o letto (read) il suo valore.
- **2** - Corrispondente all'effetto delete, per scelte progettuali. Il delete effect è proprio solamente alle entry che fanno riferimeto ad un puntatore.

I problemi risolti in questa fase si interfacciano ai casi di gestione degli effetti per:

1. Variabili
2. Propagazione cancellazione dei puntatori
3. Invocazione di funzione

4.4.1 Gestione effetti per variabili

Nei metodi di checkEffects implementati per tutte le classi statements che fanno direttamente riferimento ad una variabile (come ad esempio print, return, delete), si controlla se le effect values della entry siano maggiori di 0, nel caso non lo fosse, siamo in un caso in cui si ha l'entry di riferimento contenuta nella symbol table ma, allo stesso tempo, la variabile non ha avuto inizializzazione, per cui non è corrispondente a nessun valore in memoria e, di conseguenza, viene lanciato un errore sugli effetti per variabile non inizializzata.

Secondo le scelte progettuali fatte, tale gestione viene integrata all'interno del checkEffects di DerExp che rappresenta un Nodo wrapper in cui è contenuto un LhsNode.

```
1
2     public ArrayList<SemanticError> checkEffects(Environment env) {
3         ArrayList<SemanticError> res = new ArrayList<>();
4         STentry myEntry=null;
5         if(effectDecFun == 0) {
6             if (derExp instanceof LhsNode) {
7                 // derExp :: LhsNode
8                 effectsST = ((LhsNode<?>) derExp).getEffectsST();
9             } else {
10                // derExp :: String
11                myEntry = env.lookup(env.getNestingLevel(), derExp + "");
12                effectsST = myEntry.getEffectState(0);
13            }
14            if (effectsST == 0) {
15                res.add(new SemanticError("DerExp Error: variable " + derExp
    .toPrint("") + " not initialized"));
16                return res;
17
18            } else if (effectsST == 2) {
19                res.add(new SemanticError("DerExp Error: variable " + derExp
    .toPrint("") + " previously deleted"));
20                return res;
21
22            }
23        } else {
24            // do nothing
25        }
26        return res;
27    }
```

Listing 4.10: checkEffects della classe DerExpNode

4.4.2 Gestione della cancellazione dei puntatori

Una scelta progettuale importante riguardo la gestione degli effetti sui puntatori, è stata quella della propagazione della cancellazione.

Quando ci troviamo in un caso simile a quello seguente:

```
1         ^int p = new;
2         ^int x = new;
```

```
        x^ = 10;
        p = x; // p^ = 10;
        print p^; // 10
        delete x; // delete x :-> delete p
        print p^; // variable p^  previously deleted
```

In questo caso, dato che p punta ad x, anche p passerà allo stato di delete (ossia il suo stato degli effetti al livello di assegnazione passerà a 2). Per gestire tali effetti, si è implementato il seguente algoritmo:

Durante l'**assegnazione** tra due puntatori:

1. Data l'assegnazione left '=' right
2. Prendo le entry di left e di right dall'ambiente
3. Verifico che siano puntatori
4. Tolgo dagli hashmap di propagazione delle variabili dal nesting level dell'assegnazione fino al nesting level della dichiarazione di left il valore left, se presente
5. Aggiungo left come chiave di una nuova entry dell'hashmap di propagazione di right e imposto come valore la differenza tra il pointerCounter della entry di left e il pointerCounter dell'oggetto left dell'assegnazione (di tipo LhsNode)

Di seguito mostriamo uno snippet di codice racchiuso dentro il checkEffects di AssignmentNode che implementa tale parte dell'algoritmo:

```
if (lhs instanceof LhsNode && exp instanceof DerExpNode){

    LhsNode left = (LhsNode) lhs;
    LhsNode right = (LhsNode) ((DerExpNode) exp).getDerExp();

    if(left.getCounterST() > 0 && right.getCounterST() > 0) {

        STentry rightEntry = right.getEntry();
        STentry tmp = env.lookup(env.getNestingLevel(), left.getId());
        if (tmp == null) {
            res.add(new SemanticError("Assignment Error: cannot find symbol
    " + left.getId()));
            return res;
            }
        // reset the propagation for id = left in any other variables
        SemanticError err = env.resetPropagation(left.getId(), nestingLevel)
    ;
        rightEntry.addPropagation(left.getId(), left.getCounterST() - left.
    getCounter());
        if(err != null){
        res.add(err);
        }
}
```

Listing 4.11: Snippet di checkEffects in AssignmentNode che implementa la propagazione della cancellazione per i puntatori

Durante il checkEffects per il DeleteNode, andiamo ad impostare i puntatori dell'hashmap di propagazione dell'elemento eliminato a delete andando a seguire i passi successivi:

1. Per ogni entry dell'hashmap di propagazione dell'entry eliminata:

(a) Effettuo un lookup dell'ambiente tramite la chiave dell'entry per prelevarmi l'elemento x in cui propagare la cancellazione

(b) Imposto tutte le celle dell'array di stato di x dall'indice che parte dal nestingLevel salvato nel valore dell'entry fino alla cella 0 a 2.

Tale algoritmo è implementato da un metodo di DeletionNode chiamato propagateDelete che prende in input la entry da eliminare e l'ambiente passato nel checkSemantics e invoca tale procedura all'interno del checkEffects.

```
1   public void propagateDelete(Environment env, STentry entry){
2       for(String id : entry.getPropagation().keySet()){
3           // Getting the id of the entry in the symbol table where these
    variables are assigned
4           // to the current deleted one, we need to propagate the delete
    state for them in the
5           // pointer level between the assigned level to pointed value ::
    level - 0
6           STentry prop = env.lookup(env.getNestingLevel(), id);
7           int level = entry.getPropagation().get(id);
8
9           for(int i = level; i >= 0; i--){
10              prop.setEffectState(i, 2); // propagation of delete state
11          }
12
13      }
14  }
```

Listing 4.12: deletePropagation di DeletionNode

4.4.3 Invocazione di funzioni

La gestione degli effetti durante un invocazione di funzione, deve far si che il cambiamento degli effetti per un puntatore passato come parametro, si propaghi anche in seguito all'invocazione di funzione stessa. Per farlo, andiamo a passare all'interno del checkEffects di CallNode il riferimento alla funzione DecFunNode dell'entry presa tramite lookup sull'id definito in CallNode.

Successivamente, andiamo a passare al riferimento della funzione gli array degli effetti dei parametri puntatori cosi da poter andare ad aggiornare tali effetti e mantenerli anche in seguito all'invocazione.

Punto fisso e funzioni ricorsive

Dato che le funzioni possono essere ricorsive, bisogna andare a considerare il cambio di stato anche delle invocazioni interne. Per farlo, andiamo a distinguere tramite una variabile booleana chiamata 'myInnerState' nel checkEffects del CallNode che evita di richiamare la procedura del punto fisso per le invocazioni interne, cosi da limitarsi a definire l'analisi degli effetti solamente in una sola invocazione della procedura per determinare il minimo punto fisso.

La procedura del punto fisso corrisponde al metodo **fixedPointProc** della classe FixedPoint. Ciò che va a fare è definire vari cloni dello stato dell'ambiente del CallNode:

1. **Ambiente precedente** - Rappresentante l'ambiente all'invocazione precedente del controllo degli effetti sulla funzione, con relativi cambiamenti di stato

2. **Ambiente corrente** - Rappresenta l'ambiente all'invocazione del passo corrente del controllo degli effetti sulla funzione, che rappresenta il passo corrente del punto fisso.

3. **Ambiente finale** - In cui si salvano i cambiamenti di stato finali dati dal minimo punto fisso raggiungibile quando l'ambiente precedente e quello corrente non hanno differenze.

Nell'ambiente finale viene inserito il massimo tra il valore dell'ambiente precedente e il valore dell'ambiente corrente per ogni entry dell'Environment. Per l'aggiornamento dell'ambiente si utilizza una collezione di Ambienti che mantengono lo stato dell'ambiente ad ogni interazione del punto fisso.

Code Generation

5.1 Grammatica

5.1.1 Istruzioni per i registri

All'interno dell'ExecuteVM, sono stati aggiunti alcunni registri:

1. **r1** - Registro temporaneo per la gestione di valori logici, nonchè il valore di ritorno in seguito all'invocazione di una procedura di codeGeneration()
2. **r2** - Registro temporaneo utilizzato esclusivamente per mantenere valori all'interno di una sottoprocedura
3. **fp** - Registro per il valore del frame pointer
4. **sp** - Registro per il valore dello stack pointer
5. **hp** - Registro per il valore dell'heap pointer
6. **ra** - Registro per il valore intero che rappresenta l'indirizzo di ritorno di una procedura invocata
7. **rv** - Registro per il valore di ritorno di una procedura
8. **al** - Registro per il valore dell'access Link

Ogni registro è un valore intero utilizzato per gestire un array di interi rappresentante la memoria; ogni voce della lista precedente ha due istruzioni ad-hoc per effettuare push e pop dallo stack e istruzioni per copiare un valore di un registro ad un altro. Inoltre sono stati creati dei istruzioni per caricare da e verso la memoria ad un determinato offset per alcuni dei registri sopra elencati.

```
1     STORER1   : 'sr1' ; // store top of stack into r1
2     LOADR1    : 'lr1' ; // load r1 into the top of stack
```

Listing 5.1: Push e Pop dal e verso il registro r1

5.1.2 Istruzioni per operazioni aritmetiche o di logica booleana

Sono stati inseriti dei istruzioni per poter effettuare le operazioni aritmentiche o logiche tra i registri r1 e r2, il risultato di tutte le varie operazioni va a sovrascrivere il valore di r1.

```
ADD  : 'add' ;  // r1 <- r1 + r2
SUB  : 'sub' ;  // r1 <- r1 - r2
MULT : 'mult' ; // r1 <- r1 * r2
DIV  : 'div' ;  // r1 <- r1 / r2
AND : 'and'; // r1 = 1 if r1 && r2 is true else r1 = 0
OR : 'or'; // r1 = 0 if r1 || r2 is false else r1 = 1
NOT : 'not'; // if r1 = 1 then r1 = 0 else r1 = 1
```

Listing 5.2: Operazioni aritmetiche e di logica booleana nella grammatica per la SVM

5.1.3 Istruzioni per la gestione dell'heap

Per gestire l'heap, sono stati inseriti due istruzioni per e leggere o scrivere valori dalla memoria usando il registro r1 ed indirizzi pari al valore dell'heap pointer spostato di un offset dato come parametro. Infine, si ha una istruzione **'new'** che non fa altro che aggiornare il valore di r1 al valore dell'heap e decrementare di uno il suo valore

```
SWHP    : 'swhp' ;   // loading the value pointed by hp into r1
LWHP    : 'lwhp' ;  // storing the value of r1 into the value pointed by hp
NEW     : 'new';   //  initialization of a new pointer
```

Listing 5.3: istruzioni per la gestione dell'Heap

5.1.4 Terminazione di un programma

Un programma può terminare in maniera anomala quando il valore di hp e minore di sp + 1 oppure si fa riferimento ad un valore non compreso tra **0 e MEMSIZE - 1**.

In alternativa, un programma termina correttamente se esegue un comando **'halt'**.

5.2 Code Generation

Il code generation è rappresentato dal metodo codeGeneration dell'interfaccia Node che ha come tipo di ritorno una stringa e non prende nessun parametro.

5.2.1 Code Generation per l'Assegnazione

Per l'implementazione del code generation nel nodo AssignmentNode, abbiamo diviso il caso in cui si ha un assegnazione per un puntatore e il caso in cui si ha un assegnazione per una variabile intera e booleana. Questo perchè andiamo a dividere il caso in cui il valore da aggiornare è all'interno dello stack utilizzando il registro fp e il caso in cui il valore da aggiornare si trovi nell'heap utilizzando il registro hp.

In entrambi i casi si ha la risalita della catena statica poichè:

- Nel caso di un **puntatore**, bisogna trovare il valore dello stack nell'AR corrispondete allo scope in cui è stato dichiarato il puntatore, all'interno dello stack si avrà il riferimento all'indirizzo dell'heap in cui punta il puntatore
- Nel caso di una **variabile intera o booleana**, bisogna trovare il valore dello stack nell'AR corrispondente alla cella di memoria in cui è mantenuto il valore della variabile.

Inoltre, si va a fare la distinzione tra assegnazione di un livello di referenziazione interno del puntatore e del livello iniziale raggiugibile direttamente tramite stack andando ad eseguire una variante del codice differente tra i due casi.

5.2.2 Code Generation per Operazioni Aritmetiche e di Logica Booleana

Il codice intermedio generato per le operazioni aritmetiche o di logica booleana sono stati implementati seguendo il costrutto teorico sulle **istruzioni ad-hoc** implementate nella **grammatica** SVM.

5.2.3 Code Generation per le espressioni condizionali

Le espressioni condizionali eseguono dei controlli usando alcune istruzioni implementate nella grammatica della SVM:

```
1 BRANCHEQ : 'beq' ;  // jump to label if r1 == r2
2 BRANCHLESSEQ:'bleq' ; // jump to label if r1 <= r2
3 BRANCHLESS : 'bless' ; // jump to label if r1 < r2
```

Listing 5.4: Istruzioni utilizzate per i costrutti condizionali

In base alla logica del costrutto condizionale che si implementa, si imposta il valore di r1 a 0, nel caso la condizione risulti falsa, 1 altrimenti.

5.2.4 Code Generation per gli Identificatori

La generazione del codice per gli identificatori, come per le assegnazioni, divide i casi in cui l'ID fa riferimento ad un **puntatore** e il caso in cui l'ID fa riferimento ad una **variabile o una funzione**. In entrambi i casi si ha una risalita statica per individuare l'AR corrispondente allo scope in cui è dichiarato e utilizzare il valore in memoria direttamente (nel caso sia una semplice variabile) oppure usarlo come riferimento di partenza la risalita nell'heap.

5.2.5 Code Generation per le Dichiarazioni

Per quanto riguarda la dichiarazione di una funzione che, rispetto all'ambito teorico, si va ad inserire il codice della procedura del corpo e il riferimento al label della funzione al di sotto dell'istruzione halt cosi da poter dividere il codice da eseguire dall'esecuzione delle varie procedure invocate.

D'altra parte, per le dichiarazione di variabili, non andiamo a fare altro che inserire tramite l'istruzione **swfp**s (ossia andiamo a salvare nella memoria all'indirizzo fp + offset il valore il valore di r1). Il valore di r1 è dato dal codeGeneration dell'espressione di inizializzazione all'interno di una nuova cella all'offset definito all'interno della symbol table. Nel caso non esistesse, ritorniamo una procedura vuota. Questo perchè andremo a scrivere nell'area di memoria della variabile solamente durante l' inizializzazione; che verrà effettuata all'interno del codeGeneration dell'AssignmentNode secondo la procedura logica descritta nel paragrafo 5.2.1

5.2.6 Code Generation per le chiamate a funzioni

La procedura di chiamata a funzione è affine alla struttura teorica, l'unica differenza fatta durante l'implementazione è che il valore del registro ra, che rappresenta l'indirizzo di ritorno, è settato staticamente ad **ip + 4**, ossia il numero di istruzioni e parametri tale che, una volta richiamato ra, si punti all'istruzione successiva al salto sulla procedura della funzione.

Interprete

6.1 Implementazione di ExecuteVM

La classe **ExecuteVM** implementa la memoria come un array di interi e i registri specificati precedentemente come delle variabili di istanze intere che rappresentano un indice della memoria. Inoltre, abbiamo diviso il codice dai valori nella memoria mantenendoli in due array separati e, mentre nell'array di memoria si utilizzano i valori di hp, sp ed fp per i riferimenti, nell'array del codice si utilizza un intero ip che aggiorna il suo valore man mano durante l'esecuzione delle istruzioni.

La classe ha alcuni metodi:

1. **Push** - Effettua un push all'interno dello stack di un valore intero passato per parametro
2. **Pop** - Effettua un pop del valore in testa dello stack e lo restituisce come valore di ritorno
3. **PushHp** - Effettua un push all'interno dell'heap di un valore intero passato per parametro
4. **PopHp** - Effettua un pop del valore in testa dell'heap e lo restituisce come valore di ritorno
5. **cpu** - Esegue il codice generato dal codeGeneration

- Lo stack è stato implementato secondo un **ordine crescente**, quindi il valore in top si troverà ad un indice maggiore rispetto ai valori già presenti nella pila.
- L'heap è stato implementato secondo un **ordine decrescente**, andando ad inserire un nuovo riferimento ad un valore minore rispetto a quelli già presenti.

```
1  public class ExecuteVM {
2
3      public static final int CODESIZE = 10000;
4      public static final int MEMSIZE = 10000;
5
6      public int[] code;                          // instructions memory
7      public int[] memory = new int[MEMSIZE]; // activation records memory
8      private int ip = 0;                          // pointer to the next code
      instruction
9      public int sp = 0;                          // pointer of the next free
      record of the stack
10     private int hp = MEMSIZE - 1; // heap-> next :: heap - 1
11     private int fp = 2;  // frame ->next :: frame + 1
12     private int ra=0;      // return address
13     private int rv;      // return value
14     private int r1;       // register r1
15     private int r2;       // register r2
16     private int offset; // temporal offset for sw and lw
17     private int number; // for li on r1 and r2
```

```
18      private int al = 2;      // access link
19
20      public ExecuteVM(int[] code) {
21          this.code = code;
22      }
23
24      public void cpu(){
25          // ...
26      }
27
28      private int pop() {
29          return memory[sp--];
30      }
31
32      private void push(int v) {
33          memory[++sp] = v;
34      }
35
36      private int popHp(){ return memory[hp++];}
37
38      private void pushHp(int v) { memory[--hp] = v;}
39
40      }
```

Listing 6.1: Dichiarazione della classe ExecuteVM ed elenco delle sue variabili di istanza e dei suoi metodi