Corso di Complementi di Linguaggi di Programmazione

Report di progetto

SimpLanPlus

Gruppo:

Mae Sosto (0001039236) mae.sosto@studio.unibo.it
Francesco Vece (0001100490) francesco.vece3@studio.unibo.it

2022/2023 – 2° Semestre

Indice

1 Introduzione a SimpLanPlus

In questo progetto viene sviluppato un compilatore in grado di eseguire SimpLanPlus, un linguaggio imperativo che estende SimpLan [1], le sue caratteristiche sono elencate come segue:

- I tipi ammessi sono: interi, booleani o void.
- Le dichiarazioni di variabili sono della forma *type ID* e non ammettono inizializzazione in fase di dichiarazione.
- Il corpo di una funzione può contenere dichiarazioni e statement, e facoltativamente, possono restituire il valore ottenuto da un'espressione. La dichiarazioni di variabli con lo stesso id è possibile Nel corpo di una funzione è possibile accedere a variabili globali e chiamare altre funzioni, se precedentemente dichiarate. Inoltre, è possibile dichiarare una funzione all'interno del corpo di una funzione (se questa non hanno lo stesso id) ed è possibile avere funzioni ricorsive, ma non mutuamente ricorsive.

Il codice utilizzato per lo sviluppo del progetto è disponibile nella repository GitHub nel link a piè di pagina [2].

1.1 Grammatica di SimpLanPlus

In questa sezione vengono evidenziate le differenze e i cambiamenti effettuati tra la grammatica fornita inizialmente e quella effettivamente utilizzata in questo progetto con lo scopo di raggiungere gli obiettivi e le specifiche preposte. Segue grammatica SimpLanPlus.

Al fine di rispettare i requisiti del linguaggio, la grammatica è stata modificata introducendo gli elementi *stms: (stm)+;* e *stme: (stm)* exp;* in modo tale da differenziare i contenuti dei blocchi then e else in entrambi i tipi di if. Più nel dettaglio, l'if presente nell'elemento *stm* diventa *'if' '(' exp ')' '' left=stms '' ('else' '' right=stms '')?* e l'if presente nell'elemento *exp* diventa *'if' '(' cond=exp ')' '' left=stme '' 'else' '' right=stme ''*.

Nelle produzioni dell'elemento *exp* sono state aggiunte delle "etichette" che permettono di semplificare i controlli sugli elementi presenti nella produzione. Tali etichette sono *leftExp*, *op* e *rightExp*. Le stesse notazioni sono usate negli elementi riguardanti gli if condizionali.

Inoltre, per avere maggiore controllo su possibili errori sintattici, è necessario aggiungere in coda al programma in input la notazione *EOF* (end of file).

Per finire, sono state aggiunte delle notazioni sotto forma di *#notazione* per ogni produzione di ogni elemento che contiene più produzioni per semplificare la gestione di tali produzioni e creare le funzioni necessarie nel visitor.

```
1  grammar SimpLanPlus ;
2  prog    : exp EOF                          #expProg
3          | (dec)+ (stm)* (exp)?  EOF        #letProg
4          ;
5  dec     : type ID ';'                  #varDec
6          | type ID '(' ( param ( ',' param)* )? ')' '{' body '}' #funDec
```

[1] https://virtuale.unibo.it/pluginfile.php/1550508/mod_resource/content/1/1.SimpLan_AND_ANTLR.pdf
[2] https://github.com/StereotAIps/SimpLanPlus.git

```
7           ;
8   param   : type ID
9           ;
10  body    : (dec)* (stm)* (exp)?
11              ;
12  type    : 'int'
13          | 'bool'
14          | 'void'
15          ;
16  stm     : ID '=' exp ';' #asgStm
17          | ID '(' (exp (',' exp)* )? ')' ';'#callStm
18          | 'if' '(' exp ')' '{' left=stms '}' ('else' '{' right=stms '}')? #ifStm
19              ;
20  stms    : (stm)+
21          ;
22  stme    : (stm)* exp
23          ;
24  exp     :  INTEGER #intExp
25          | ('true' | 'false') #boolExp
26          | ID #idExp
27          | '!' exp #notExp
28          | leftExp=exp (op='*' | op='/') rightExp=exp #numExp
29          | leftExp=exp (op='+' | op='-') rightExp=exp #numExp
30          | leftExp=exp  '==' rightExp=exp #eqExp
31          | leftExp=exp (op='>' | op='<' | op='>=' | op='<=' ) rightExp=exp #compExp
32          | leftExp=exp (op='&&' | op='||') rightExp=exp #opExp
33          | 'if' '(' cond=exp ')' '{' left=stme '}' 'else' '{' right=stme '}' #ifExp
34          | '(' exp ')' #baseExp
35          | ID '(' (exp (',' exp)* )? ')' #callExp
36          ;
```

1.2 Struttura compilatore

Lo schema utilizzato per la stuttura del compilatore è stata costruita su modello della struttura utilizzata per SimpLan, dove:

- **ast**: contiene le implementazioni dei nodi dell'albero sintattico e dei visitor
- **evaluator**: contiene le implementazioni per eseguire il codice assembly generato
- **mainPackage**: contiene il main del compilatore, il file del codice in input e del codice assembly generato
- **parser**: contiene le classi generate da ANTLR sulla base della grammatica SimpLanPlus.
- **svm**: contiene le classi generate da ANTLR sulla base della grammatica SVM.
- **symboltable**: contiene le classi che servono a gestire la symbol table.

- **utils**: contiene le classi utilizzate come supporto alla gestione delle strutture utilizzate.

1.3 Software e tools

Il codice di questo elaborato è stato implementato usando come IDE IntelliJ IDEA [3] sviluppato da JetBrains, un ambiente di sviluppo integrato scritto in Java per lo sviluppo di software scritto in Java. Per quanto riguarda il versioning, è stato utilizzato GitHub, Inc. [4] sviluppato da Microsoft Corporation, un servizio di hosting per lo sviluppo di software e il controllo del versioning tramite Git. Per la generazione del parser, è stato utilizzato il software ANTLR [5] (ANother Tool for Language Recognition), un generatore di parser che utilizza un algoritmo per il parsing LL.

La versione di Java utilizzata come SDK è la *11.0.18* scaricabile al link [6] a fondo pagina. L'unica libreria importata è la seguente *antlr-4.12.0-complete.jar*, anche questa scaricabile gratuitamente tramite il link [7] a fondo pagina.

[3] https://www.jetbrains.com/idea/
[4] https://github.com
[5] https://www.antlr.org
[6] https://www.oracle.com/java/technologies/javase/11-0-18-relnotes.html
[7] https://www.antlr.org/download.html

2 Esercizio 1

Una volta effettuate le modifiche alla grammatica SimpLanPlus fornita come modello, è stato utilizzato ANTLR per generare il codice utilizzato dal parser e dal lexer.

Nel main è stato definito il file *prova.simplan* da utilizzare per passare il programma in input e sono stati definiti gli oggetti che gestiscono il parser e il lexer. La libreria di ANTLR ha permesso di utilizzate la classe *ANTLRInputStream* per ottenere il contenuto del file di input, estrarre i token e effettuare il parsing.

In questa fase sono stati definiti tutti gli elementi (appartenenti all'interfaccia *Node*) che fanno parte della grammatica per andare a costruire l'AST (albero di sintassi astratta). Sempre per questo scopo, è stato effettuato un override dei metodi predefiniti contenuti nella classe *SimpLanPlusBaseVisitor*. Questi nuovi metodi si trovano in *SimpLanPlusBaseVisitorImpl* contenuto nel pacchetto *ast* e vengono utilizzati per costruire l'AST nel momento in cui nel main viene chiamata la funzione *visit ()*.

È stata inoltre implementata la funzione *toPrint()* in tutti i nodi, questa è in grado di stampare sul terminale di IntelliJ l'AST del programma per avere una rappresentazione visiva più chiara.

Per quanto riguarda la gestione degli errori, è in questa fase che è stata implementata la classe *ParserErrorHandler* contenuta nel pacchetto parser per gestire gli errori lessicali e sintattici. Alla verifica di uno o più errori, l'handler verrà triggerato in modo automatico, le funzioni presenti in *ParserErrorHandler* permetteranno di salvare all'interno di un ArrayList di stringhe gli errori lessicali e sintattici, tali errori verrano scritti nel file *errori.log* presente nel pacchetto mainPackage. La classe *ParserErrorHandler* estende la classe base di ANTLR *BaseErrorListener*, eseguendo l'override del metodo *syntaxError*.

2.1 Esempi

Vengono di seguito mostrati tre diversi esempi di codice errati che contengono corrispondentemente un errore lessicale, sintattico e semantico. Tali errori vengono scritti nel file *errori.log*, che si aggiorna ogni volta che un nuovo programma viene eseguito.

- Il primo tipo di errore trattato è quello lessicale. Questo avviene quando il lexer non è in grado di riconoscere nella stringa di ingresso gruppi di simboli che corrispondono a specifiche categorie sintattiche. Consideriamo il seguente codice e errore riportato dal compilatore SimpLanPlus :

```
1 int a;
2 int 1b;
```

```
1 [!] An error occurred at line 2, character 4 :no viable alternative at
2 input 'int1'
3 [!] An error occurred at line 2, character 5 :mismatched input 'b'
4 expecting {<EOF>, '*', '/', '+', '-', '==', '>', '<', '>=', '<=',
5 '&&', '||'}
```

In questo caso il carattere *1* non può essere usato come nome di indentificatore, poichè la grammatica definisce che gli identificatori debbano essere della forma *ID : CHAR (CHAR*

| DIGIT) ;*, difatti, l'errore indica l'impossibilità di riconoscere 'int1' in una grammatica (o meglil, sequenza di token) nota.

- Il secondo tipo di errore trattato è quello sintattico. Questo si verifica quando data una sequenza s di token (in cui il programma sorgente è stato tradotto dall'analizzatore lessicale), la sequenza non appartiene al linguaggio sorgente del compilatore specificato dalla grammatica G. Consideriamo il seguente codice e errore riportato dal compilatore SimpLanPlus :

```
1 int a;
2 a = 2;
3 int b;
```

```
1 [!] An error occurred at line 3, character 0 :extraneous input 'int'
2 expecting {<EOF>, '(', 'if', 'true', 'false', '!', INTEGER, ID}
```

  In questo caso la dichiarazione di b non può essere effettuata in seguito all'inizializzazione di a. Come specificato nella grammatica *prog : (dec)+ (stm)* (exp)? EOF*, in seguito a una o più dichiarazioni e uno statemente è possibile avere solo altri statement o un'espressione, ma non è possibile avere altre dichiarazioni, difattil'errore indica "extraneous input 'int' expecting <EOF>", la gramamtica si aspetta che ci sia un end of file e non una dichiarazione.

- Il terzo tipo di errore trattato è quello semantico. Questi si verificano quando ci sono degli errori logici che derivano da un'errata logica di stesura del programma da parte del programmatore. Consideriamo il seguente codice e errore riportato dal compilatore SimpLanPlus :

```
1 int a;
2 int b;
3 a = b;
```

```
1 [!] A semantic error occurred: Id b used but not initialized
```

  In questo caso b viene dichiarato ma mai inizializzato, quindi nel momento in cui si trova nella parte destra di un assegnamento viene segnalato un errore logico.

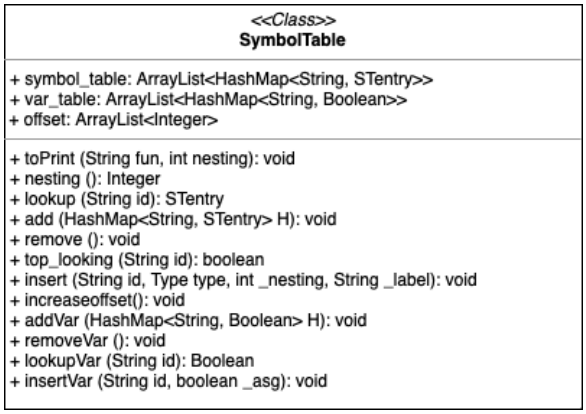
3 Esercizio 2

Lo scopo del secondo esercizio serve a gestire:

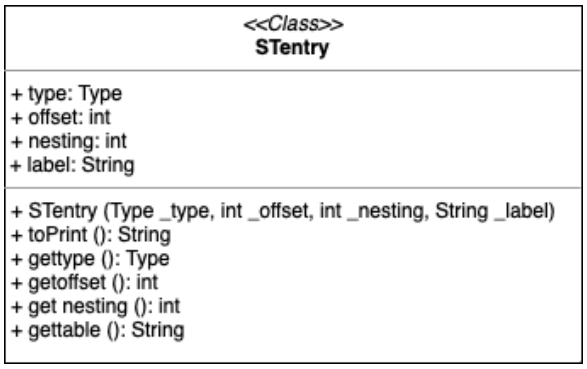
1. identificatori/funzioni non dichiarati
2. identificatori/funzioni dichiarate più volte nello stesso ambiente

3.1 Symbol Table

Per soddisfare i punti elencati precedentemente viene utilizzata una symbol table (ST) con struttura invariata rispetto a quella di SimpLanPlus fornito come modello. Gli attributi e i metodi della classe *SymbolTable* sono mostrati in Figura 1a. Tale struttura implementa una lista di HashMap per la gestione degli ambienti della symbol table. Ogni HashMap è costruita fornendo come chiave una stringa contenente l'identificativo *id* della variabile o della funzione e da un oggetto STentry che contiene le informazioni dell'elemento con identificativo *id*.



(a) Struttura della classe SymbolTable



(b) Struttura della classe STentry

La struttura di STentry contiene i metodi e i campi mostrati in Figura 1b, dove:

- **type** è il tipo della varibile o della funzione, dove nel primo caso i tipi ammessi sono int, bool e void, mentre nel secondo caso ho un un tipo *ArrowType* costruito nel modo seguente. Dove *inputtype* è la lista che contiene i tipi dei parametri in input e *outputtype* è il tipo dell'output della funzione, che può essere anch'esso int, bool e void.

```
1       public class ArrowType extends Type {
2               private ArrayList<Type> inputtype;
3               private Type outputtype;
4       ...
5       }
```

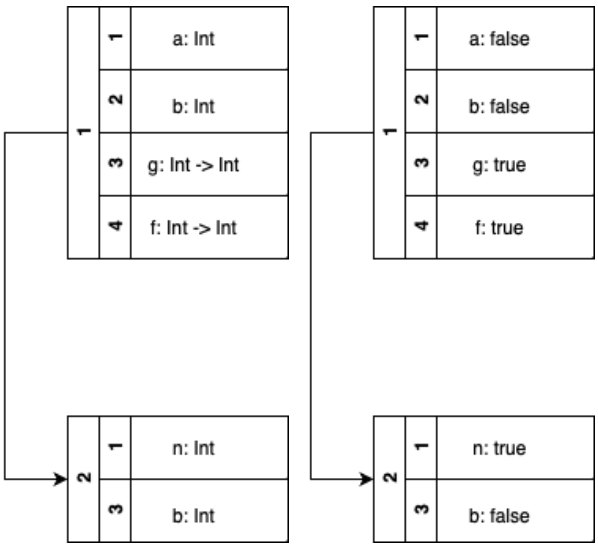
- **offset** è un numero intero che indica l'offset di quella variabile o funzione in quell'ambiente.
- **nesting** è un numero intero che indica il livello di nesting nella quale la variabile o funzione è stata dichiarata.

- **label** è una stringa che, nel caso delle variabili è vuota (""), nel caso delle funzioni contiene l'etichetta assegnata a quella funzione da utilizzare in fase di code generation.

Per quanto riguarda gli oggetti *var_table* e *offset* servono a gestire rispettivamente le variabili dichiarate ma non assegnate e l'assegnamento degli offset. Più informazioni riguardanti la Var Table (VT) sono presenti nella Sezione 4.1.

Di seguito viene mostrato un esempio del funzionamento della ST.

```
1  int a ;
2  int b ;
3  int g(int p){
4      int y ;
5      y = 2;
6      p+y
7  }
8  int f(int n){
9      int b ;
10     b = 3;
11     g(b+n)
12 }
13 a = 1;
14 b = 2 ;
15 f(a) ;
```



(a) Symbol table **(b)** Var table

Figura 2: ST e VT alla riga 10

La ST e VT riga 10 sono mostrate in Figura 2. Da questa immagine è possibile notare che alle dichiarazioni di funzioni, i loro identificatori vengono salvati nell'ambiente in cui queste vengono dichiarate (in questo caso abbiamo g e f nell'ambiente 0), quando si entra nel corpo di una funzione viene creato un nuovo ambiente contenente i parametri formali (in questo caso

la variabile intera n) e gli identificatori degli elementi dichiarati nel corpo della funzione (in questo caso x).

È inoltre utile notare come gli offset siano assegnati agli elementi della ST, il meccanismo è quello di assegnare un valore intero incrementale a partire da 0 in ogni ambiente. Nell'ambiente creato al corpo della funzione verranno prima assegnati i parametri formali, lasciato un offset per il valore di ritorno e solo in seguito vengono assegnati gli elementi dichiarati nel corpo della funzione.

3.2 Dichiarazioni multiple

Per quanto riguarda la gestione degli elementi non dichiarati o dichiarazioni multiple nello stesso ambiente vengono fatti dei controlli durante le dichiarazioni di varibili, funzioni e parametri formali di una stessa funzione. Nel primo caso si controlla che l'dentificativo utilizzato per la nuova varibile non sia già presente nell'ultimo ambiente della ST (in questo caso viene riportato l'errore), se così non fosse l'identificativo viene aggiunto all'ultimo ambiente della ST. Il controllo è implementato nel nodo *DecvarNode* come segue.

```
1  if (ST.top_lookup(id) == true)
2          errors.add(new SemanticError("Var id " + id + " already declared"));
3      else{
4          ST.insert(id, (Type) type, nesting,"") ;
5          ST.insertVar(id);
6      }
```

Nel caso del controllo sulla dichiarazione di funzione e dei suoi paramenti, il controllo eseguito nel nodo *DecfunNode* è il seguente.

```
1  if (ST.lookup(id) != null)
2      errors.add(new SemanticError("Identifier " + id + " already declared"));
3  else {
4      HashMap<String, STentry> HM = new HashMap<String, STentry>();
5      flabel = SimpLanlib.freshFunLabel();
6      ST.insert(id, type, nesting, flabel);
7      ST.add(HM);
8      for (ParNode arg : parlist) {
9          if (ST.top_lookup(arg.getId()))
10             errors.add(new SemanticError("Parameter id " + arg.getId() +
11             " already declared"));
12         else {
13             ST.insert(arg.getId(), arg.getType(), nesting + 1, "");
14         }
15     }
16     ...
```

Come nel caso della dichiarazione di variabile, anche qua si controlla che l'identificativo della funzione non sia presente nell'ultimo ambiente della ST (in questo caso viene riportato l'errore), se così non fosse l'identificativo viene aggiunto all'ultimo ambiente della ST. Solo in seguito

viene creato un nuovo ambiente e vengono effettuati i controlli sui paramentri, se questi non sono presenti nell'ultimo (nuovo) ambiente allora vengono aggiunti, altrimenti viene restituito errore.

3.3 Identificativi non dichiarati

Per la gestione degli identificativi non dichiarati sono stati implementati dei controlli sui nodi che richiamano variabili o funzioni, ovvero nel momento in cui viene fatto un assegnamento, espressione che include identificativi o chiamate una fuzione.

Per quanto riguarda il primo caso, il controllo è il seguente e si trova nel nodo *AsgNode*.

```
1 STentry tmp = ST.lookup(id) ;
2     if (tmp != null) {
3         entry = tmp ; //Se è stato dichiarato allora ok prendo l'entry
4     } else { //Se non è stato dichiarato allora errore
5         errors.add(new SemanticError("Id " + id + " not declared")) ;
6     }
```

Prima si recupera l'elemento con identificativo *id* dalla ST, in seguito si controlla se questo elemento sia *null* o meno, nel primo caso si sta provando ad assegnare un'espressione a un elemento non dichiarato, in caso contrario restituisco l'elemento. Per quanto riguarda le espressioni, bisogna eseguire il controllo nel nodo stesso della variabile, ovvero *IdExpNode*. Stessa cosa riguarda la chiamata di funzione, il controllo viene fatto nel nodo *CallNode*. In entrambi i casi, il codice è pressochè simile a quello mostrato precedentemente quindi per semplicità non verrà qui riportato.

4 Esercizio 3

Lo scopo dell'esercizio 3 è quello di soddisfare i seguenti punti.

- correttezza dei tipi (in particolare numero e tipo dei parametri attuali se conformi al numero e tipo dei parametri formali).
- gestione dell'uso di variabili non inizializzate.

Una volta implementate le funzionalità della Symbol table, si è proseguito nel progetto lavorando sull'analisi semantica, ovvero verificare la correttezza dei tipi (type checking) e l'uso di variabili dichiarate ma non inizializzate.

4.1 Varibili utilizzate ma non inizializzate e Var Table

Prima di proseguire con le modalità di gesitone delle variabili dichiarate ma non inizializzate, è fondamentale fornire più informazioni riguardanti l'utilizzo della Var table utilizzata a questo scopo. Come è stato già introdotta nella Sezione 3.1, la Var table (VT) è un elemento della classe *SymbolTable* della forma $ArrayList < HashMap < String, Boolean >>$.

La necessità dell'utilizzo di tale struttura è nato dal momento in cui si è sentita la necessità di voler prendere nota di quali identificatori fossero stati inizializzati e quali no. Come primo approccio c'è stato un tentativo di integrale una variabile *assigned* di tipo bool alla struttura *STentry*, senza però putroppo aver riscontrato buorni risultati. In seguito viene mostrato un esempio di codice che mostra una delle criticità riscontrate.

```
1 int x ;
2 int y ;
3 if (e) {
4     x = 2 ;
5 } else {
6     y = x ;
7 }
```

Il primo approccio tentato avrebbe previsto l'assegnamento della variabile $assigned = true$ alla riga 4 del codice, per poi valutatre come lecita la riga di codice a riga 6, nonostante l'assegnamento fosse stato fatto nel ramo then, uno scope che rimane chiuso tra le due parentesi graffe a riga 3 e 5. In questo codice, la variabile x dovrebbe risultare come utilizzata ma non inizializzata.

```
1 int x ;
2 int y ;
3 if (e) {
4     y = 2 ;
5 } else {
6     x = 1 ;
7 }
8 y = x ;
```

Secondo il primo approccio tentato, la variabile x sarebbe stata assegnata a riga 6, anche qua, in uno scope che diverso rispetto alla quale questa è utilizzata a riga 8. In questo codice, la variabile x dovrebbe risultare come utilizzata ma non inizializzata.

Per ricorrere a questo problema, si è deciso di utilizzate una struttura separata da quella della della gestione della ST e quindi avere la possibilità di manipolare gli ambienti di scope delle variabili in modo coerente con la struttura e i nodi del codice. Ciò nonostante la struttura della VT è pressochè simile a quello della ST, difatti entrambe vengono modificate negli stessi punti del codice effettuando le stesse operazioni fatta eccezione dei nodi in cui bisogna effettuare controlli sull'utilizzo delle variabili. Un esempio di come si dovrebbe comportare la VT è presente in Figura 2b. La VT quindi crea/elimina nuovi "scope" nello stesso momento in cui sulla ST vengono creati/eliminati nuovi ambienti. Le uniche eccezioni e controlli aggiuntivin vengono applicati nei nodi del condizionale if, sia *IfExpNode* che *IfStmNode*, il nodo dell'assegnamento *AsgNode* e il nodo foglia degli id *IdExpNode*. I codici risultati sono i seguenti.

```
1  public ArrayList<SemanticError> checkSemantics(SymbolTable ST, int _nesting) {
2          nesting = _nesting;
3          ArrayList<SemanticError> errors = new ArrayList<SemanticError>();
4          errors.addAll(exp.checkSemantics(ST, nesting));
5          HashMap<String, Boolean> V1 = new HashMap<String,Boolean>() ;
6          ST.addVar(V1);
7          for (Node d : thenbranch) {
8              errors.addAll(d.checkSemantics(ST, nesting)) ;
9          }
10         ArrayList<String> V1List = TakeDeclaredVariables(V1);
11         ST.removeVar();
12         HashMap<String, Boolean> V2 = new HashMap<String,Boolean>() ;
13         ST.addVar(V2);
14         for (Node d : elsebranch) {
15             errors.addAll(d.checkSemantics(ST, nesting)) ;
16         }
17         ArrayList<String> V2List = TakeDeclaredVariables(V2);
18         ST.removeVar();
19         ArrayList<String> FinalList= CompareEnvironmentVariables(V1List, V2List);
20         for (String id: FinalList)
21             ST.insertVar(id, true);
22         return errors;
23     }
```

Il codice soprastante è quello utilizzato nelle classi *IfExpNode* che *IfStmNode*. In questo caso, a riga 7 e 14 vengono dichiarati due nuove strutture $HashMap < String, Boolean >$ utilizzate per tenere traccia delle variabili inizializzate corrispettivamente nel ramo then ed else. A riga 12 e 19, viene utilizzata la funzione $TakeDeclaredVariables(HashMap < String, Boolean > V)$ per mettere in una lista $ArrayList < String >$ il nome delle variabili inizializzate in quello scope. Solo a riga 21 la funzione $CompareEnvironmentVariables(ArrayList < String > L1, ArrayList < String > L2)$ compara le due liste e a riga 23 gli elementi presenti in entrambe le liste vengono aggiunte allo scope attuale. Tramite questa tecnica è possibile risolvere i

problemi mostrati precedentemente dal momento in cui veongo creati e distrutti scope apposta per i due rami dell'if senza andare a modificare direttamente lo socpe attuale.

Ma come è possibile sapere se un identificativo è stato assegnato o meno nel momento in cui viene eseguita un'espressione o statement? Nel nodo *IdExpNode* è stato aggiunto il seguente controllo.

```
1 //Controllo se è assegnato nella tabella degli assegnamenti
2 Boolean varInfo = ST.lookupVar(id) ;
3 if(varInfo!= null)
4     if(varInfo == FALSE && (st_type.getnesting() == nesting)){
5     //Se la variabile non è assegnata
6     errors.add(new SemanticError("Id " + id + " used but not initialized"));
7 }
```

Tramite questo controllo è possibile ottenere tramite la funzione *lookupVar(String id)* l'oggetto nella VT che fa riferimento all'id *id*. A riga 3 viene verficato che l'elemento esista, in questo caso è stato dichiarato e aggiunto alla tablla correttamente. Dopo di che viene controllato a riga 4 se l'elemento è presente in questo scope e ha come valore FALSE allora non è mai stato inizializzato e deve essere restituito un errore.

È inoltre importante precisare che nel momento in cui l'id delle funzioni viene aggiunto alla VT, questo avrà sempre valore TRUE per fare in modo che nel corpo di una funzione, questa possa richiamare sè stessa pur appartenendo allo stesso scope. Stessa cosa avviene per i parametri formali delle funzioni, anch'essi vengono assegnati come TRUE dal momento in cui vengono inizializzati durante la chiamata a funzione. In fine, nel momento in cui viene fatto un assegnamento, l'identificativo di sinista viene aggiunto alla VT con valore TRUE.

4.2 Regole di inferenza

Per quanto riguarda il type cheking sono state specificate le seguenti regole di inferenza dove:

- L'ambiente Γ è una funzione definita come $\Gamma : ID \rightarrow type, O, nesting$ e fa riferimento alla ST.
- Lo scope Θ è una funzione definita come $\Theta : ID \rightarrow assigned$ e fa riferimento alla VT.

4.2.1 Progammi

Programma solo con espressione

$$\frac{\emptyset \cdot [], \emptyset \cdot [], O, N \vdash exp : \Gamma, \Theta, T}{\vdash exp : T} \text{ [ProgExp]}$$

Regola iniziale del programma che contiene solo un espressione. L'offset e il livello di nesting iniziale viene posto a 1.

Programma con dichiarazioni, statement ed espressione

$$\frac{\begin{array}{c}\emptyset \cdot [], \emptyset \cdot [], O, N \vdash dec : \Gamma, \Theta \\ \Gamma, \Theta \vdash stm : \Gamma, \Theta^{'} \\ \Gamma, \Theta^{'} \vdash exp : \Gamma, \Theta^{''}, T\end{array}}{\vdash dec\ stm\ exp} \text{ [Prog1]}$$

Regola iniziale del programma, consente di avere dichiarazioni, statements e un'epressione. L'offset e il livello di nesting iniziale viene posto a 1.

Programma con dichiarazioni ed espressione

$$\frac{\begin{array}{c}\emptyset \cdot [], \emptyset \cdot [], O, N \vdash dec : \Gamma, \Theta \\ \Gamma, \Theta \vdash exp : \Gamma, \Theta^{'}, T\end{array}}{\vdash dec\ exp} \text{ [Prog2]}$$

Regola iniziale del programma, consente di avere dichiarazioni e un'epressione. L'offset e il livello di nesting iniziale viene posto a 1.

4.2.2 Dichiarazioni

Dichiarazione di variabile

$$\frac{id \notin dom(top(\Gamma))}{\Gamma, \Theta, O \vdash T\ id; : \Gamma[id \to T], \Theta[id \to false], O + 1} \text{ [DecVar]}$$

Nella dichiarazione di variabile bisogna controllare che l'identificativo non faccia parte dell'ultimo ambiente di Γ, in caso contrario l'id viene aggiunto all'ambiente Γ e scope Θ attuale. Nello scope Θ, l'identificatore della variabile viene assegnata a false perché non è ancora stata inizializzata. L'offset O viene incrementato di 1 per fare riferimento alla variabile nello stack.

Sequenza di dichiarazioni

$$\frac{\Gamma, \Theta, O \vdash d : \Gamma^{'}, \Theta^{'}, O^{'} \qquad \Gamma^{'}, \Theta^{'}, O^{'} \vdash D : \Gamma^{''}, \Theta^{''}, O^{''}}{\Gamma, \Theta, O \vdash d\ D : \Gamma^{''}, \Theta^{''}} \text{ [DecSeq]}$$

Nella sequenza di dichiarazioni viene esplorata una dichiarazione alla volta. L'ambiente, lo scope e l'offset vengono aggionati dopo la prima dichiarazione e quindi vengono passati alle dichiarazioni che seguono la prima.

Dichiarazione di funzione

$$\frac{\begin{array}{c}\Gamma \cdot [f \mapsto (argsTypes(args)) \to T], \Theta \cdot [f \mapsto true], O + 1, N + 1 \vdash args : \Gamma^{'}, \Theta^{'}, O^{'}, N^{'} \\ \Gamma^{'}, \Theta^{'}, O^{'} + 1, N^{'} \vdash body : T1 \\ f \notin dom(top(\Gamma)) \qquad\qquad T = T1\end{array}}{\Gamma, \Theta, O, N \vdash T\ f(args)\{body\} : \Gamma[f \mapsto (argsTypes(args)) \to T], \Theta[f \mapsto true]} \text{ [DecFun]}$$

Nella dichiarazione di funzione viene creato un nuovo ambiente in Γ e un nuovo scope in Θ. L'oggetto $args$ è costruito nel modo seguente $T_1x_1, ..., T_nx_n$. In Γ viene aggiunto l'id della funzione che ha come tipo un oggetto costrutito come $(T_1, ..., T_n) \to T$, ovvero "i tipi dei parametri

in input" → "il tipo della funzione". Per ottenere il tipo dei parametri $T_1, ..., T_n$ viene usata la funzione *argsTypes(args)*. In Θ viene aggiunto l'identificativo della funzione con valore true per fare in modo che questo possa essere richiamato nell corpo della funzione stessa. Nella regola che va in arg, l'offset O viene incrementato di 1 perché si stanno valutando elementi interni a un nuovo ambiente e scope, inoltre, anche il livello di nesting N aumenta di 1 per fare riferimento all'identificativo della funzione nello stack. Nella regola che va nel body l'offset O' viene incrementato nuovamente per fare spazio all'indirizzo di ritorno.

Parametro (un argomento)

$$\frac{id \notin dom(top(\Gamma))}{\Gamma, \Theta, O \vdash T\ id : \Gamma[id \to T], \Theta[id \to true], O + 1}\ \text{[Arg]}$$

Se l'identificativo del parametro non appartiene all'ultimo ambiente Γ allora lo si aggiunge all'ultimo ambiente Γ, gli viene assegnato nello scope Θ il valore true e l'offset viene aumentato di 1.

Parametri (argomenti di funzione)

$$\frac{\begin{array}{c}\Gamma, \Theta, O \vdash a : \Gamma', \Theta', O' \\ \Gamma', \Theta', O' \vdash A : \Gamma'', \Theta'', O''\end{array}}{\Gamma, \Theta, O \vdash a\ A : \Gamma'', \Theta'', O''}\ \text{[Args]}$$

Dichiarazioni, statement ed espressione (Body)

$$\frac{\begin{array}{c}\Gamma, \Theta, O \vdash dec : \Gamma', \Theta, O' \\ \Gamma', \Theta, O' \vdash stm : \Gamma', \Theta', O' \\ \Gamma', \Theta', O' \vdash exp : \Gamma', \Theta'', O', T\end{array}}{\Gamma, \Theta \vdash dec\ stm\ exp : T}\ \text{[Body]}$$

4.2.3 Statements

Assegnamento

$$\frac{\begin{array}{c}\Gamma, \Theta, N \vdash Id : T1 \quad \Gamma, \Theta \vdash exp : \Gamma, \Theta', T2 \\ T1 = T2\end{array}}{\Gamma, \Theta, N \vdash Id\ =\ exp : \Gamma, \Theta'[Id \mapsto true], void}\ \text{[Asg]}$$

Viene controllato che i tipi dell'identificativo e dell'espressione siano gli stessi. Nello scope Θ l'identificativo viene assegnato a true in seguito all'assegnamento.

Condizionale if (stm)

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash exp : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash stme1 : \Gamma, \Theta'', T2 \quad \Gamma, \Theta' \vdash stme2 : \Gamma, \Theta''', T3 \\ T1 = bool\end{array}}{\Gamma, \Theta \vdash if\ (exp)\ \{stme1\}\ else\ \{stme2\} : \Gamma, CompScope(\Theta'', \Theta'''), void}\ \text{[IfStm1]}$$

Nel condizionale if bisogna innanzitutto controllare che l'espressione della condizione sia di tipo booleano. I due rami alterano lo scope Θ in modo diverso, ragione per cui viene utilizzata una

funzione $CompScope(\Theta'', \Theta''')$ per paragonare i due nuovi scope e passare lo scope di ritorno contenente gli identificativi valutati come assegnati in modo corretto. Tale funzione compara i due scope tenendo traccia di quali identificativi siano assegnati come true, se questi sono presenti in entrambi gli scope allora rimangono, altrimenti le modifiche vengono riportate a Θ'.

Condizionale if senza ramo else (stm)

$$\frac{\Gamma, \Theta \vdash exp : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash stme1 : \Gamma, \Theta'', T2 \quad T1 = bool}{\Gamma, \Theta \vdash if\ (exp)\ \{stme1\} : \Gamma, \Theta', void}\ [\text{IfStm2}]$$

Statement

$$\frac{\Gamma, \Theta \vdash s : \Gamma, \Theta', T}{\Gamma, \Theta \vdash s\ S : \Gamma, \Theta'', void}\ [\text{Stm}]$$

Sequenza di statements

$$\frac{\Gamma, \Theta \vdash s : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash S : \Gamma, \Theta'', T2}{\Gamma, \Theta \vdash s\ S : \Gamma, \Theta'', void}\ [\text{SeqStm}]$$

Chiamata di funzione

$$\frac{\Gamma, \Theta \vdash f : (T_1, ..., T_n) \to T \quad (\Gamma e_i : T_i')_{i=1}^n \quad (T_i = T_i')_{i=1}^n}{\Gamma, \Theta \vdash f(e_1, ..., e_n); : \Gamma, \Theta, T}\ [\text{CallExp}]$$

Si controlla che i parametri attuali e formali in input alla funzione abbiamo un tipo compatibile.

4.2.4 Espressioni

Espressione generica

$$\frac{\Gamma, \Theta \vdash exp : \Gamma, \Theta', T}{\Gamma, \Theta \vdash exp : \Gamma, \Theta'', T}\ [\text{Exp}]$$

Assioma intero

$$\frac{}{\Gamma \vdash num : \Gamma, int}\ [\text{IntExp}]$$

Assioma booleano

$$\frac{}{\Gamma \vdash true : \Gamma, bool}\ [\text{BoolExpTrue}]$$

$$\frac{}{\Gamma \vdash false : \Gamma, bool}\ [\text{BoolExpFalse}]$$

Variabile

$$\frac{id \in dom(\Gamma) : T \quad (\Theta(id) = true \vee \neg(\Gamma(id).nesting = N))}{\Gamma, \Theta, N \vdash id : T}\ [\text{IdExp}]$$

In questo caso bisogna controllare che l'identificativo della variabile sia presente in Γ e che il valore dell'identificatico in Θ sia true (ovvero la variabile è assegnata) oppure che non sia stata dichiarata nell'ambiente attuale (e quindi in quel caso è una variabile globale e deve essere accettata).

Espressione in parentesi

$$\frac{\Gamma, \Theta \vdash exp : \Gamma, \Theta', T}{\Gamma, \Theta \vdash (exp) : \Gamma, \Theta', T} \text{ [BaseExp]}$$

Espressioni booleane

$$\frac{\Gamma, \Theta \vdash exp : \Gamma, \Theta', T \quad T == bool}{\Gamma, \Theta \vdash ! \; exp : \Gamma, \Theta', bool} \text{ [NotExp]}$$

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash e1 : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash e2 : \Gamma, \Theta'', T2 \\ T1 = bool = T2 \qquad \sigma : bool \times bool \to bool\end{array}}{\Gamma, \Theta \vdash e1 \; \sigma \; e2 : \Gamma, \Theta'', bool} \text{ [OpExp]}$$

Dove il simbolo σ viene utilizzato per fare riferimento ai simboli:&& e ||.

Espressioni intere

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash e1 : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash e2 : \Gamma, \Theta'', T2 \\ T1 = int = T2 \qquad \sigma : int \times int \to int\end{array}}{\Gamma, \Theta \vdash e1 \; \sigma \; e2 : \Gamma, \Theta'', int} \text{ [NumExp]}$$

Dove il simbolo σ viene utilizzato per fare riferimento ai simboli:$+, -, *$ e $/$.

Espressioni di confronto

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash e1 : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash e2 : \Gamma, \Theta'', T2 \\ T1 = T2 \qquad ==: T1 \times T2 \to bool\end{array}}{\Gamma, \Theta \vdash e1 \; == \; e2 : \Gamma, \Theta'', bool} \text{ [EqExp]}$$

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash e1 : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash e2 : \Gamma, \Theta'', T2 \\ T1 = int = T2 \qquad \sigma : int \times int \to bool\end{array}}{\Gamma, \Theta \vdash e1 \; \sigma \; e2 : \Gamma, \Theta'', bool} \text{ [NumExp]}$$

Dove il simbolo σ viene utilizzato per fare riferimento ai simboli:$<, \Leftarrow, >$ e $>=$.

Condizionale if (exp)

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash cond : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash stme1 : \Gamma, \Theta'', T2 \quad \Gamma, \Theta'' \vdash stme2 : \Gamma, \Theta''', T3 \\ T1 = bool \qquad\qquad\qquad\qquad\qquad\qquad T2 = T = T3\end{array}}{\Gamma, \Theta \vdash if \; (cond) \; \{stme1\} \; else \; \{stme2\} : \Gamma, CompScope(\Theta'', \Theta'''), T} \text{ [IfExp]}$$

Anche in questo caso, come nel caso del condizionale if (stm) senza espressione, bisogna controllare che la condizione sia di tipo booleano ed anche qua viene utilizzata la funzione $CompScope(\Theta'', \Theta''')$ per comparare gli scope dei due branch.

Statement ed espressione

$$\frac{\begin{array}{c}\Gamma, \Theta \vdash stm : \Gamma, \Theta', T1 \quad \Gamma, \Theta' \vdash exp : \Gamma, \Theta'', T2 \\ T1 = void\end{array}}{\Gamma, \Theta \vdash stm \; exp : \Gamma, \Theta'', T2} \text{ [StmExp]}$$

Chiamata di funzione

$$\frac{\Gamma, \Theta \vdash f : (T_1, ..., T_n) \to T \quad (\Gamma e_i : T_i')_{i=1}^n \quad (T_i = T_i')_{i=1}^n}{\Gamma, \Theta \vdash f(e_1, ..., e_n) : \Gamma, \Theta, T} \text{ [CallExp]}$$

4.3 Esempi

In questa sezione testiamo i seguenti 4 codici forniti nelle specifiche del progetto.

1. **Codice 1**

```
1  int a;
2  int b;
3  int c ;
4  c = 2 ;
5  if (c > 1) {
6      b = c ;
7  } else {
8      a = b ;
9  }
```

I messaggi in output dal compilatore sono i seguenti.

```
1  Parsing in progress....
2  Parse completed without errors!
3  Checking semantic errors...
4  You had: 1 errors:
5  [!] A semantic error occurred: Id b used but not initialized
```

Come si legge dallì'ultimo messaggio di errore, l'identificatore b è utilizzato ma mai usato, infatti è possibile vedere che nel ramo then b è effettivamente inizializzato ma tale modifica è presente e valida solo in quel ramo, quindi una volta che il compilatore controlla gli elementi utilizzati nell'espressione del ramo else, trova errore a riga 8 nel momento in cui si prova a utilizzare b, dichiarato a riga 2 ma mai inizializzato in questo scope.

2. **Codice 2**

```
1  int a ;
2  int b ;
3  int c ;
4  void f(int n){
5      int x ;
6      int y ;
7      if (n > 0) {
8          x = n ;
9      } else {
10         y = n+x ;
11     }
12  }
13  c = 1 ;
14  f(0) ;
```

I messaggi in output dal compilatore sono i seguenti.

```
Parsing in progress....
Parse completed without errors!
Checking semantic errors...
You had: 1 errors:
[!] A semantic error occurred: Id x used but not initialized
```

Anche in questo caso, come nel caso precedente, il compilatore segnala un uso illecito della variabile x, questa è inizializzata nel corpo della funzione f a riga 5 e inizializzata nel ramo then, quindi nel momento in cui il controllo raggiunge il ramo else l'id x non risulta inizializzato in questo scope.

3. **Codice 3**

```
void h(int n){
    int x ;
    int y ;
    if (n==0){
        x = n+1 ;
    } else {
    h(n-1) ;
    x = n ;
    y = x ;
    }
}
h(5) ;
```

I messaggi in output dal compilatore sono i seguenti.

```
Parsing in progress....
Parse completed without errors!
Checking semantic errors...
Visualizing AST...
Checking type errors...
Type checking ok! Type of the program is: [T]Void
Code generated! Assembling and running generated code.
...
```

Eseguedo questo codice non vengono trovati errori da parte del compilatore.

4. **Codice 4**

```
int a;
void h(int n){
    int x ;
    int y ;
    if (n==0){
        x = n+1 ;
    } else {
```

```
8           h(n-1) ;
9           y = x ;
10      }
11  }
12  h(5) ;
```

I messaggi in output dal compilatore sono i seguenti.

```
1  Parsing in progress....
2  Parse completed without errors!
3  Checking semantic errors...
4  You had: 1 errors:
5  [!] A semantic error occurred: Id x used but not initialized
```

Anche in questo codice, come in quelli precedenti, il compilatore segnala che la variabile x sia stata utilizzata ma non inizializzata, questa è inizializzata nel corpo della funzione h a riga 2 e inizializzata nel ramo then, quindi nel momento in cui il controllo raggiunge il ramo else l'id x non risulta inizializzato in questo scope.

5 Esercizio 4

Lo scopo di questo esercizio è quello di estendere l'interprete di SimpLan per implementare l'interprete di SimpLanPlus.

5.1 Interprete SVM

Per perseguire gli obiettivi di questa fase, è stato importato al progetto il file *SVM.g4*, contentente la grammatica dell'interprete di base, nel package *svm*. In seguito è stato utilizzato lo stesso metodo già utlizzato per la grammatica del compilatore *SimpLanPlus.g4*, ovvero è stato utilizzato ANTLR per generare il codice del visitor della SVM. Successivamente, sono stati creati i metodi per la generazione di codice in ogni nodo dell'albero creato precedentemente per costruire l'AST.

Nel main sono state aggiunte le componenti che scrivono in un file di appoggio *prova.simplan.asm* (contenuto nel package *mainPackage*) il codice generato dal programma. Venogno poi utilizzati gli stessi step usati precedentemente per leggere il nuovo file creato, estrarre i token ed effettuare il parser. In fine, viene eseguita la VM per eseguire il codice generato.

5.2 Esercizi

In questa sezione testiamo i seguenti 4 codici forniti nelle specifiche del progetto.

1. **Codice 1**

```
int x ;
void f(int n){
        if (n == 0) { n = 0 ; }        // n e` gia` uguale a 0; equivale a fare s
        else { x = x * n ; f(n-1) ; }
}
x = 1 ;
f(10)
```

Il codice viene compilato ed eseguito correttamente ed il risultato ottenuto è 0 dal momento in cui la funzione chiamata è di tipo void, ovvero non restituisce nessun valore. Il valore di ritorno è salvato nel registro a0 che viene inzialmente settato a 0.

```
Parsing in progress....
Parse completed without errors!
Checking semantic errors...
Visualizing AST...
Checking type errors...
Type checking ok! Type of the program is: [T]Void
Code generated! Assembling and running generated code.
Starting Virtual Machine...
*** STACK ***
Code executed with success!
The result is: 0
```

2. **Codice 2**

```
int u ;
int f(int n){
        int y ;
        y = 1 ;
        if (n == 0) { y }
        else { y = f(n-1) ; y*n }
}
u = 6 ;
f(u)
```

Il codice viene compilato ed eseguito correttamente ed il risultato ottenuto è 720.

```
Parsing in progress....
Parse completed without errors!
Checking semantic errors...
Visualizing AST...
Checking type errors...
Type checking ok! Type of the program is: [T]Void
Code generated! Assembling and running generated code.
Starting Virtual Machine...
*** STACK ***
Code executed with success!
The result is: 720
```

3. **Codice 3**

```
int u ;
void f(int m, int n){
        if (m>n) { u = m+n ;}
        else { int x ; x = 1 ; f(m+1,n+1) ; }
}
f(5,4) ;
```

Il compilatore si interrompe al controllo degli errori presenti nel parser nel momento in cui il codice non è coerente con la gammatica. Infatti, nel corpo del ramo else è presente una dichiarazione di variabile. Le dichiarazioni non sono consentite nei branch del costrutto if.

```
Parsing in progress....
[!] An error occurred at line 4, character 8 :extraneous input 'int'
expecting {'if', ID}
[!] An error occurred at line 4, character 14 :no viable alternative
at input 'x;'
parser.ParserErrorHandler@15d0c81b
```