Relazione del progetto del corso di Complementi di linguaggi di programmazione

Alessio Di Pasquale [0001097535]

Andrea Schinoppi [0001097628]

Indice

0 Introduzione **2**
0.1 Tipi di Dati . . . 2
0.2 Dichiarazioni . . . 2
0.3 Istruzioni . . . 2
0.4 Espressioni . . . 3
0.5 Scopo del progetto . . . 3
0.6 Ambiente di lavoro . . . 4
0.7 Contenuto della relazione . . . 5

1 Esercizio 1 - analisi lessicale **6**
1.1 Esempi di codice . . . 6
1.2 Creazione dell'AST . . . 7
1.2.1 Implementazione dei nodi . . . 7

2 Esercizio 2 - tabella dei simboli **10**
2.1 Implementazione della symbol table . . . 10
2.2 Variabili non dichiarate o dichiarate più volte . . . 11
2.3 Esempi di codice . . . 11

3 Esercizio 3 - analisi semantica **12**
3.1 Esempi di codice . . . 15

4 Esercizio 4 - interprete **17**
4.1 Esempi di codice . . . 18

Introduzione

SimpLanPlus (SLP) è un semplice linguaggio imperativo che estende SimpLan.
Di seguito si provvede ad una breve descrizione:

0.1 Tipi di Dati

SLP supporta i seguenti tipi di dati:

- **Integer**: rappresenta i numeri interi senza decimali.
- **Boolean**: rappresenta i valori di verità (*true* o *false*).
- **Void**: viene utilizzato per indicare l'assenza di un tipo di dato, ad esempio per le funzioni che non restituiscono alcun valore.

0.2 Dichiarazioni

SLP consente la dichiarazione di variabili e funzioni, che possono essere ricorsive ma non mutuamente ricorsive.

- **Dichiarazione di Variabili**: Le variabili possono essere dichiarate specificando il tipo di dato seguito da un identificatore univoco (ad esempio `int x;`).
  Non è permesso eseguire dichiarazione e inizializzazione nella stessa istruzione (`int x = 5;` è sbagliato).
- **Dichiarazione di Funzioni**: Le funzioni possono essere dichiarate specificando il tipo di dato restituito, un identificatore univoco, eventuali parametri e il corpo della funzione
  (es: `int fact(int n) {if (n == 0) {1} else {fact(n - 1) * n}}`).

0.3 Istruzioni

SLP prevede anche istruzioni per manipolare i dati e controllare il flusso di esecuzione del programma.

- **Assegnamento**: Consente di assegnare un valore a una variabile già dichiarata utilizzando l'operatore "=".
  Questo fa sì che la variabile diventi inizializzata permettendo le relative operazioni.

- **Chiamata di funzione**: Per richiamare una funzione precedentemente definita, è possibile utilizzare il suo identificatore seguito da eventuali parametri attuali racchiusi tra parentesi.
- **Istruzioni di controllo**: L'istruzione "if" è utile per eseguire una parte del codice solo se una determinata condizione è vera. È possibile specificare un blocco di istruzioni da eseguire nel caso in cui la condizione sia vera ed eventualmente un altro blocco di istruzioni da eseguire nel caso in cui la condizione sia falsa.

0.4 Espressioni

SLP fornisce alcune espressioni per eseguire operazioni aritmetiche e logiche.
Queste espressioni sono permesse soltanto su variabili già inizializzate di tipo int e bool.

- **Espressioni Aritmetiche**: Le espressioni aritmetiche supportate includono addizione, sottrazione, moltiplicazione e divisione di numeri interi e ritornano sempre un numero intero.
  Le espressioni aritmetiche accettano soltanto valori interi e restituiscono interi.
- **Espressioni di Confronto**: Sono supportati operatori di confronto come maggiore di, minore di, maggiore o uguale a, minore o uguale a e uguale a.
  Le espressioni di confronto accettano soltanto valori interi e restituiscono booleani ad eccezione dell'espressione "uguale a" (==) che accetta sia interi che booleani.
- **Espressioni logiche**: SLP supporta operatori logici come "&&" (AND), "||" (OR) e "!" (NOT) per eseguire operazioni logiche tra valori booleani. Le espressioni logiche accettano soltanto valori booleani e restituiscono booleani.
- **Espressioni Condizionali**: È possibile utilizzare l'istruzione "if" all'interno delle espressioni per valutare una condizione e restituire un valore diverso a seconda del risultato.

0.5 Scopo del progetto

Scopo del progetto è lo sviluppo di un compilatore per SimpLanPlus attraverso lo svolgimento di quattro esercizi:

- **Esercizio 1**: L'analizzatore lessicale deve ritornare la lista degli errori lessicali in un file di output.
- **Esercizio 2**: Sviluppare la tabella dei simboli del programma.
  Il codice sviluppato deve controllare:
  - Identificatori/funzioni non dichiarati;
  - Identificatori/funzioni dichiarate più volte nello stesso ambiente.
- **Esercizio 3**: Sviluppare un'analisi semantica che verifichi:
  - La correttezza dei tipi (in particolare numero e tipo dei parametri attuali se conformi al numero e tipo dei parametri formali);
  - Uso di variabili non inizializzate (solo per questo punto senza accesso di funzioni a variabili globali).
- **Esercizio 4**: Estendendo l'interprete di SimpLan, implementare l'interprete di SimpLanPlus.

0.6 Ambiente di lavoro

Per svolgere questo progetto abbiamo utilizzato IntelliJ IDEA community edition con il plugin per ANTLR, la funzione "code with me" dell'IDE per poter lavorare nello stesso ambiente anche a distanza e Git per il controllo di versione.

Il progetto è suddiviso in cartelle, di seguito elencate:

- **docs**: Cartella contenente questa relazione e le istruzioni per il progetto.
- **gen**: Cartella contenente i file generati da ANTLR, sia per la grammatica SimplanPlus.g4, sia per la grammatica SVM.g4 (package "SVMpkg").
- **lib**: Cartella contenente il file .jar di antlr versione 4.12.0.
- **out**: Cartella degli output di compilazione e del programma.
- **src**: Cartella dei file sorgente, contenente i file principali del progetto e i seguenti package:
  - **ast**: contiene le implementazioni dei nodi dell'albero sintattico;
  - **interpreter**: contiene l'implementazione dell'interprete di SimpLanPlus;
  - **others**: file utili allo svolgimento del progetto,
  - **semanticanalysis**: contiene l'implementazione della symbol table.

0.7 Contenuto della relazione

Ogni capitolo successivo alla presente introduzione rappresenta uno dei quattro esercizi assegnati.

Esercizio 1 - analisi lessicale

Per svolgere questo esercizio è stata sovrascritto l'error handler utilizzato dal lexer di ANTLR tramite l'utilizzo della classe `Handler` e le funzioni `removeErrorListeners` e `addErrorListener`.

In particolare la classe `Handler` estende il `BaseErrorListener` di ANTLR tramite Override del metodo `syntaxError` in modo da inserire in un'apposita struttura dati gli errori sintattici rilevati.

Gli errori sintattici vengono rilevati durante la creazione dell'albero di sintassi astratta (AST), spiegata nel dettaglio nella sezione 1.2 dedicata.

Se al termine dell'analisi, la lista degli errori sintattici non è vuota, viene creato il file `out/errors.txt` dentro al quale vengono indicati tutti gli errori e l'esecuzione viene fermata.

1.1 Esempi di codice

Di seguito vengono riportati 3 esempi di codice testati e relativi output:

Codice:

```
int x;
int x_1;
x = 5;
x_1 = 6;
```

Output:

```
Errore sintattico:
Riga 2, Carattere: 6
```

Codice:

```
int x;
int y;
int z;
z= x \ y;
```

Output:

```
Errore sintattico:
Riga 4, Carattere: 4
```

Codice:

```
int x;
x = 5:
```

Output:

```
Errore sintattico:
Riga 2, Carattere: 5
(missing ';' at '<EOF>')
```

Nei primi due casi è stato inserito un token non previsto dalla grammatica (rispettivamente "_" e "\"), nel terzo caso invece il ";" è stato sostituito da ":".

1.2 Creazione dell'AST

Per prima cosa la grammatica `SimplanPlus.g4` è stata modificata mediante inserimento di identificatori per permettere al parser di ANTLR di generare contesti differenti per ognuna delle produzioni possibili.
Inoltre sono state inseriti dei nuovi elementi della grammatica per accedere ai rami "then" ed "else" delle espressioni condizionali.

Abbiamo successivamente inserito delle etichette all'interno della grammatica per accedere in modo più semplice agli elementi delle produzioni, ad esempio in

- `e1=exp (op='+' | op='-') e2=exp #plusMinusExp`
- `'if' '(' condition=exp ')' '' thenB=thenStmBranch '' ('else' '' elseB=elseStmBranch '')? #ifStm`

è possibile notare l'uso degli identificatori `#plusMinusExp` e `#ifStm` e di etichette (come ad esempio `e1` e `elseB`), oltre ai nuovi elementi della grammatica `thenStmBranch` ed `elseStmBranch`.

In seguito alle modifiche abbiamo implementato la classe `Visitor` che estende la classe `SimpLanPlusBaseVisitor` generata da ANTLR.
Questa classe è utilizzata per la creazione vera e propria dell'AST tramite la chiamata della funzione `visit` della quale è stata implementata una versione per ogni possibile elemento della grammatica e relativo contesto generato.
Questa funzione viene chiamata con input `parser.prog()` che, come indicato nella grammatica, può essere etichettato come `#singleExp` o `#multipleExp`. In base a ciò, all'interno del `Visitor` verranno chiamate rispettivamente `visitSingleExp` o `visitMultipleExp`, che chiameranno a loro volta le `visit` degli elementi al loro interno.

Ogni `visit` crea e ritorna una struttura dati `Node` diversa per ogni elemento visitato.
L'insieme dei nodi ritornati e le relative implementazioni andranno a comporre l'AST.

1.2.1 Implementazione dei nodi

Ogni nodo è stato implementato a partire dall'interfaccia `Node` che contiene i seguenti metodi:

- `ArrayList<SemanticError> checkSemantics`:

  - Controllo degli errori semantici del nodo corrente.
    In caso di errori, essi vengono aggiunti ad un ArrayList che verrà infine ritornato.

- `TypeNode typeCheck`:
  - Type checking del nodo corrente mediante regole di inferenza
    Ritorna un nodo contenente le informazioni sul tipo.

- `String toPrint`:
  - Utility per stampare a schermo le informazioni del nodo.

- `String codeGeneration`:
  - Genera il codice intermedio per il nodo corrente.

In particolare i tipi di nodo implementati a partire da `Node` sono stati:

- **Nodo per il programma**: ProgramNode:
  - Contiene le Arraylist di `Node` delle dichiarazioni e degli statement e il `Node` della eventuale espressione finale.
    Nel caso in cui il programma fosse di tipo `#singleExp` sarebbe presente solo il `Node` dell'espressione.

- **Nodi per gli statement**:
  - AsgNode:
    * Contiene come stringa l'ID a cui si sta assegnando e il `Node` dell'espressione da assegnare.
  - FunCallNode:
    * Contiene come stringa l'ID della funzione chiamata e l'Arraylist di `Node` dei parametri attuali.

- **Nodi per dichiarazioni**
  - DecFunNode:
    * Contiene il `TypeNode` per accedere alle informazioni sul tipo, l'ID come stringa, gli Arraylist di `Node` per parametri, dichiarazioni e statement all'interno del corpo ed eventuale espressione di ritorno.
  - DecNode:
    * Contiene il `TypeNode` per accedere alle informazioni sul tipo e l'ID come stringa.

- **Nodo per parametri di funzioni**: ArgNode:
  - Contiene il `TypeNode` per accedere alle informazioni sul tipo e l'ID come stringa.
- **Lookup**: IdNode;
  - Contiene l'ID come stringa.
- **Nodi per condizioni**
  - IfExpNode:
    - Contiene il `Node` della guardia, gli Arraylist di `Node` per gli statement nei rami "then" ed "else" e i due `Node` per le eventuali espressioni di ritorno presenti nei rami.
  - IfStmNode:
    - Contiene il `Node` della guardia e gli Arraylist di `Node` dei rami.
- **Nodi per i tipi**
  - TypeNode:
    - Contiene la stringa che identifica il tipo e un Arraylist di `TypeNode` per accedere ai tipi dei parametri nel caso si trattasse di una funzione.
  - BoolNode:
    - Contiene il valore booleano del nodo.
  - IntNode:
    - Contiene il valore intero del nodo.
- **Nodi operazioni aritmetiche e booleane**
  - LogicalExpNode;
  - MulDivNode;
  - PlusMinusNode;
  - CfrExpNode:
    - Contengono i `Node` per il primo e il secondo elemento dell'operazione ed una stringa per identificare l'operatore.
  - NotExpNode;
    - Contiene il `Node` per l'espressione all'interno del Not.

Esercizio 2 - tabella dei simboli

Una volta terminata con successo l'analisi sintattica viene creato un oggetto di classe `Environment` a cui è affidata la gestione degli offset per la generazione di codice e la symbol table.

Questo Environment, inizialmente vuoto, viene passato alla funzione `checkSemantics` che in base al tipo di nodo dell'AST incontrato popolerà la symbol table e modificherà gli offset.

2.1 Implementazione della symbol table

La symbol table è stata implementata per mezzo di uno Stack di Hashtable, che a loro volta sono una coppia {ID, `SymbolTableEntry`}. Una SymbolTableEntry è a sua volta composta da una label, un tipo, un offset e uno status, utile a verificare se la entry è dichiarata o inizializzata.

La scelta di utilizzare uno Stack per gestire la symbol table è dovuta sia alla semplicità di utilizzo delle funzioni per inserire ed eliminare elementi, sia dal fatto che l'entrata e l'uscita dai blocchi è rappresentata in modo ottimale dal crescere della pila.
Inoltre grazie a questa struttura non è necessario tenere traccia del nesting level in quanto questo è rappresentato dalla profondità della pila.

Nuovi valori vengono aggiunti alla symbol table in caso di dichiarazioni o parametri attuali di funzioni, mentre il nesting viene modificato all'ingresso e all'uscita da una funzione.

Nel caso di rami "then" ed "else" di un costrutto if, è necessaria una *deep copy* di tutto l'environment al fine di controllare entrambi i rami a partire dallo stesso ambiente.
Se in un ramo viene inizializzata una variabile e ciò non avviene anche nell'altro viene sollevata un'eccezione dato che non è possibile stabilire staticamente a livello di type checking quale sarà l'ambiente risultante.

Se si volesse generare il codice per programmi che non rispettano il vincolo di cui sopra è necessario utilizzare la flag `-skiptypecheck` (o `-s`) in fase di esecuzione per ignorare il type checking del programma.

2.2 Variabili non dichiarate o dichiarate più volte

La verifica delle variabili non dichiarate o dichiarate più volte è affidato alle funzioni `typeCheck` dei nodi dell'AST, dove si controlla se per un ID dato è già presente una entry nella symbol table allo stesso nesting level.

2.3 Esempi di codice

Codice:

```
int x;
y = 5;
```

Output:

```
Variabile y non dichiarata
nell'ambiente corrente.
```

Codice:

```
int x;
int y;
int x;
y = 5;
```

Output:

```
Variabile x già dichiarata
nel blocco corrente.
```

Codice:

```
int x;
int y;
x = y + z;
```

Output:

```
Variabile z
non dichiarata.
```

Codice:

```
int x;
int y;
y = 1;
if (y > 0) {x = 1;}
else {y = 0;}
```

Output:

```
Incompatibilità
di assegnamenti
nel ramo then
e nel ramo else
per la variabile x
```

Esercizio 3 - analisi semantica

Nel caso dell'analisi semantica, la correttezza dei tipi e l'uso di variabili non inizializzate sono verificati dalle funzioni `typeCheck` implementate nei vari nodi.

Nelle regole di inferenza sottostanti, ci siamo concentrati nel far rispettare a ogni gruppo di elementi della stessa categoria (ad esempio i vari `exp`) lo stesso tipo di `unification`. Ad esempio nel caso delle exp tutte tornano un tipo seguito da uno stato.

Si assume inoltre di non dover rappresentare tutti i possibili scenari dovuti alla chiusura di kleene tramite Alberi di derivazione.

Di seguito le regole di derivazione implementate:

- Program

$$\_ \quad \frac{\emptyset \cdot [\,], 0 \vdash dec : \Gamma, n \; \Gamma, n \vdash stm : \Gamma', n \; \Gamma', n \vdash exp : T, init}{\emptyset, 0 \vdash dec \; stm \; exp : T}$$

$$\_ \quad \frac{\emptyset \cdot [\,], 0 \vdash exp : T, init}{\emptyset, 0 \vdash exp : T}$$

- Asg

$$\_ \quad \frac{\Gamma, n \vdash e : T, init \; \Gamma, n \vdash ID : T', S \; T = T'}{\Gamma, n \vdash ID = e : \Gamma[ID \to T, \; init], \; n}$$

  Dove lo stato $S \in \{dec, init\}$

- StmSeq

$$\_ \quad \frac{\Gamma, n \vdash stm_1 : \Gamma', n \; \Gamma', n \vdash stm_2 : \Gamma'', n}{\Gamma, n \vdash stm_1 \; stm_2 : \Gamma''; n}$$

- Bool

$$\_ \quad \frac{}{\Gamma, n \vdash true : bool, init}$$

$$\_ \quad \frac{}{\Gamma, n \vdash false : bool, init}$$

- CfrExp

$$\_ \quad \frac{\Gamma, n \vdash e_2 : T, init \; \Gamma, n \vdash e_1 : T', init \; T = T' \;\; ==: T \times T \to bool}{\Gamma, n \vdash e_1 == e_2 : bool, init}$$

$$\_ \quad \frac{\Gamma, n \vdash e_2 : T, init \; \Gamma, n \vdash e_1 : T', init \; T = int = T' \;\; >=: int \times int \to bool}{\Gamma, n \vdash e_1 >= e_2 : bool, init}$$

$$\_ \quad \frac{\Gamma, n \vdash e_2 : T, init \; \Gamma, n \vdash e_1 : T', init \; T = int = T' \;\; <=: int \times int \to bool}{\Gamma, n \vdash e_1 <= e_2 : bool, init}$$

$$\_ \quad \frac{\Gamma, n \vdash e_2 : T, init \; \Gamma, n \vdash e_1 : T', init \; T = int = T' \;\; >: int \times int \to bool}{\Gamma, n \vdash e_1 > e_2 : bool, init}$$

$$\_ \quad \frac{\Gamma, n \vdash e_2 : T, init \; \Gamma, n \vdash e_1 : T', init \; T = int = T' \;\; <: int \times int \to bool}{\Gamma, n \vdash e_1 < e_2 : bool, init}$$

- DecFun

$$\frac{\Gamma \cdot [f \to (T_1 ... T_n) \to T, x_1 \to T_1, ..., \overline{x_n \to T_n}], n \vdash body : T' \; T = T' \; f \notin dom(top(\Gamma))}{\Gamma, n \vdash Tf(T_1 x_1 ... T_n x_n) = body; \; \Gamma[f \to (T_1 ... T_n) \to T], n}$$

- FunBody

$$\_ \quad \frac{\Gamma, n \vdash dec : \Gamma', n' \; \Gamma', n' \vdash stm : \Gamma'', n' \; \Gamma'', n' \vdash exp : T, init}{\Gamma, n \vdash dec \; stm \; exp : T}$$

$$\_ \quad \frac{\Gamma, n \vdash dec : \Gamma', n' \; \Gamma', n' \vdash stm : \Gamma'', n'}{\Gamma, n \vdash dec \; stm : void}$$

- Dec

$$\_ \quad \frac{ID \notin dom(top(\Gamma))}{\Gamma, n \vdash T \; ID : \Gamma[ID \to T, \; dec], n + 1}$$

$$\_ \quad \frac{\Gamma, n \vdash d : \Gamma', n' \; \Gamma', n' \vdash D : \Gamma'', n''}{\Gamma, n \vdash d \; D : \; \Gamma'', n''}$$

- FunCallExp

$$\_ \quad \frac{\Gamma, n \vdash f : T_1 \times ... \times T_n \to T \; (\Gamma, n \vdash e_i : T'_i)_{i \in 1...n} \; (T_i = T'_i)_{i \in 1...n}}{\Gamma, n \vdash f(e_1 ... e_n) : T, init}$$

- FunCallStm

$$\_ \quad \frac{\Gamma, n \vdash f : T_1 \times ... \times T_n \to T \; (\Gamma, n \vdash e_i : T'_i)_{i \in 1...n} \; (T_i = T'_i)_{i \in 1...n}}{\Gamma, n \vdash f(e_1 ... e_n) : \Gamma, n}$$

- Id

$$\_ \quad \frac{\Gamma(ID) : T, S}{\Gamma, n \vdash ID : T, S}$$

Dove lo stato $S \in \{dec, init\}$

- IfExp

$$\frac{\Gamma, n \vdash e_1 : bool, init \; \Gamma, n \vdash IfExp_1 : \Gamma', T \; \Gamma, n \vdash IfExp_2 : \Gamma'', T'' \; T' = T'' \Gamma''' = choose(\Gamma', \Gamma'')}{\Gamma, n \vdash \text{if } e_1\{IfExp_1\} \text{ else } \{IfExp_2\}\text{: } T, init}$$

- Then/elseExpBranch

$$\_ \quad \frac{\Gamma, n \vdash stm : \Gamma', n \qquad \Gamma', n \vdash e : T, init}{\Gamma, n \vdash stm \; e\text{: } \Gamma', T}$$

- IfStm

$$\_ \quad \frac{\Gamma, n \vdash e_1 : bool, init \; \Gamma, n \vdash Stm_1 : \Gamma', n \; \Gamma, n \vdash Stm_2 : \Gamma'', n \; \Gamma''' = choose(\Gamma', \Gamma'')}{\Gamma, n \vdash \text{if } e_1\{Stm_1\} \text{ else } \{Stm_2\}\text{: } \Gamma''', n}$$

- Int

$$\_ \quad \frac{}{\Gamma, n \vdash NUM : int, init}$$

- LogicalExp

$$\_ \quad \frac{\Gamma, n \vdash e_1 : bool, init \; \Gamma, n \vdash e_2 : bool, init \; \&\& : bool \times bool \to bool}{\Gamma, n \vdash e_1 \&\& e_2 : bool, init}$$

$$\_ \quad \frac{\Gamma, n \vdash e_1 : bool, init \; \Gamma, n \vdash e_2 : bool, init \; || : bool \times bool \to bool}{\Gamma, n \vdash e_1 || e_2 : bool, init}$$

- MulDiv

$$\_ \quad \frac{\Gamma, n \vdash e_1 : T, init \; \Gamma, n \vdash e_2 : T', init \; T = int = T' * : int \times int \to int}{\Gamma, n \vdash e_1 * e_2 : int, init}$$

$$\_ \quad \frac{\Gamma, n \vdash e_1 : T, init \; \Gamma, n \vdash e_2 : T', init \; T = int = T' \; / : int \times int \to int}{\Gamma, n \vdash e_1 / e_2 : int, init}$$

- NotExp

$$\_ \quad \frac{\Gamma, n \vdash e : bool, init \; ! : bool \to bool}{\Gamma, n \vdash !e : bool, init}$$

- Bracket

$$\_ \quad \frac{\Gamma, n \vdash e : T, init}{\Gamma, n \vdash \; (e) : T, init}$$

- PlusMinus

$$\frac{\Gamma, n \vdash e_1 : T, init \; \Gamma, n \vdash e_2 : T', \overline{init} \; T = int = T' + : int \times int \to int}{\Gamma, n \vdash e_1 + e_2 : int, init}$$

$$\frac{\Gamma, n \vdash e_1 : T, init \; \Gamma, n \vdash e_2 : T', \overline{init} \; T = int = T' - : int \times int \to int}{\Gamma, n \vdash e_1 - e_2 : int, init}$$

3.1 Esempi di codice

Codice:

```
int a;
int b;
int c ;
c = 2 ;
     if (c > 1) { b = c ; }
     else { a = b ; }
   }
```

Output:

```
Errore di TypeChecking:
Uso della variabile b
dichiarata ma
non inizializzata.
```

Codice:

```
int a;
int b;
int c ;
void f(int n){
  int x ; int y ;
    if (n > 0)
    { x = n ;}
    else { y = n+x ;}}
c = 1 ;
f(0) ;
```

Output:

```
Errore di TypeChecking:
Uso della variabile x
dichiarata ma
non inizializzata.
```

Codice:

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

Output:

```
Type Checking
completato
con successo.
```

Codice:

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

Output:

```
Errore di TypeChecking:
Uso della variabile x
dichiarata ma
non inizializzata.
```

Esercizio 4 - interprete

Scopo di questo esercizio è estendere l'interprete di SimpLan fornito per creare l'interprete di SimpLanPlus.

Una volta eseguite le precedenti analisi viene generato il codice intermedio mediante le funzioni `codeGen` dei nodi dell'AST: in base al nodo consultato saranno scritte le istruzioni nel file `code.asm` che poi sarà letto dall'interprete per eseguirlo.

La grammatica di riferimento è nel file `SVM.g4` e il bytecode è composto dalle seguenti istruzioni:

- **load REG NUMBER (REG)**
- **store REG NUMBER (REG)**
- **storei REG NUMBER**
- **move REG REG**
- **add REG REG**
- **addi REG NUMBER**
- **sub REG REG**
- **subi REG NUMBER**
- **mul REG REG**
- **muli REG NUMBER**
- **div REG REG**
- **divi REG NUMBER**
- **push (n=NUMBER | l=LABEL)**
- **pushr REG**
- **pop**
- **popr REG**
- **b LABEL**
- **beq REG REG LABEL**
- **bleq REG REG LABEL**
- **jsub LABEL**

- **rsub REG**
- **l=LABEL**
- **halt**

A queste regole ne sono state aggiunte altre 3 per consentire tutte le operazioni di confronto consentite dalla grammatica. In particolare:

- **bgt REG REG LABEL**
- **blt REG REG LABEL**
- **bgte REG REG LABEL**

Queste istruzioni sono utilizzate per effettuare salti condizionali con operatori di: $>, <, \geq$. Il comportamento di queste operazioni è stato implementato all'interno di `ExecuteVM`.

I registri utilizzati per il funzionamento dell'interprete sono SP, FP, AL, RA, IP, A0, T1 e T2, rispettivamente i registri Stack pointer, Frame pointer, Access link, Return address, Instruction pointer, il registro contenente il risultato dell'ultima istruzione eseguita e due registri temporanei.

Per facilitare il debug sono stati sostituiti gli opcode nel memory inspector con i rispettivi nomi delle operazioni tramite una HashTable.

4.1 Esempi di codice

Codice:

```
int x ;
void f(int n){
if (n == 0) { n = 0 ; }
else {x = x * n ; f(n-1);}
}
x = 1 ;
f(10)
```

Output:

```
Ignoro Type Checking.
Starting Virtual Machine...
Result: 0
```

Codice:

```
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)
```

Output:

```
Ignoro Type Checking.
Starting Virtual Machine...
Result: 720
```

Codice:

```
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) ; u
```

Output:

```
Ignoro Type Checking.
Starting Virtual Machine...
Result: 9
```

Codice:

```
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(4,5) ; u
```

Output:

```
Ignoro Type Checking.
Starting Virtual Machine...
Result: (LOOP)
```