Università di Bologna

Corso di Compilatori e Interpreti

Dip. di Informatica

Progetto SimpLanPlus

Benetton Alessandro [0001038887]

Crespan Lorenzo [0001038888]

Li Zhiguang [0001029938]

1 settembre 2022

Indice

1 Descrizione progetto

1.1 Definizione di SimpLanPlus

SimpLanPlus è un linguaggio imperativo che presenta le seguenti caratteristiche:

- Parametri delle funzioni possono essere passati per valore o per riferimento:
  - In caso di **passaggio per valore**, alla funzione viene passata una copia degli argomenti e le potenziali modifiche non hanno ripercussioni sul valore originale delle variabili passate alla funzione;
  - In caso di **passaggio per riferimento** (utilizzando il prefisso "var" sul parametro), alla funzione viene passato un puntatore all'indirizzo della variabile. Ogni modifica avrà ripercussioni sul valore originale della variabile passata alla funzione;
- Le funzioni non sono annidate, ergo una funzione non può essere definita nel corpo di un'altra;
- Ammette la ricorsione, ma non la mutua ricorsione (detta anche ricorsione indiretta). Quest'ultima si verifica quando nel codice di una funzione viene richiamata un'altra funzione che a sua volta richiama la prima (vedi 1.1).

```
1  public class MyRecursiveMethods {
2
3      // Funzione ping richiamata pong
4      public static int ping(int n) {
5          if (n < 1) return 1;
6          else return pong(n - 1);
7      }
8
9      // Funzione pong richiamata ping
10     public static int pong(int n) {
11         if (n < 0) return 0;
12         else return ping(n/2);
13     }
14 }
```

Listing 1: Esempio di mutua ricorsione

1.2 Obiettivi progetto

Nel progetto completo è richiesto lo sviluppo di un compilatore per SimpLanPlus e la produzione di un report, nel quale devono essere illustrate le scelte progettuali.

2 Esercizio 1

2.1 Obiettivo

Nel seguente esercizio viene richiesto dall'analizzatore lessicale di riportare gli errori in un file di output.
Di seguito sono riportati tre codici di esempio e i relativi output di errore.

2.2 Esempi errori segnalati

Nei seguenti codici è stata verificata la capacità dell'analizzatore lessicale di identificare token non presenti all'interno della grammatica. Questa tipologia di errore è causata principalmente da errori di battitura o distrazione.

Nel codice sono presenti anche errori di natura semantica che non devono essere riportati dall'analizzatore lessicale.

2.2.1 Esempio 1

```
1     {
2         7
3         bool isEven (int value){
4             if( value == 0){
5                 retun true;
6             } else {
7                 if (value === 1){
8                     return false;
9                 } else {
10                    return isEvens(value - 2);
11                }
12            }
13        }
14
15        bool num = 10a;
16        print (isEven(num));
17    }
```

Funzione ricorsiva per la valutazione di valore pari o dispari.

Errori riportati all'interno del documento "Test_1.simplan_[Data].log" generato dall'analizzatore lessicale:

```
1     Errors:
2       [ERROR] Error at pos 2:4. extraneous input '7' expecting {'void',
      'if', 'return', 'print', '{', '}', 'int', 'bool', ID}
3       [ERROR] Error at pos 5:18. no viable alternative at input '
      retuntrue'
4       [ERROR] Error at pos 7:24. extraneous input '=' expecting {'(',
      '!', '-', BOOL, ID, NUMBER}
5       [ERROR] Error at pos 15:17. extraneous input 'a' expecting ';'
```

Errori riportati da SimpLanPlus sul codice riportato precedentemente

Il primo errore, dovuto al carattere "7" inserito in riga 2, simula una pressione accidentale di un tasto da parte dell'utente mentre si muove nel codice.

Il secondo e il quarto errore, rispettivamente in riga 5 e 10, simulano un errore di battitura in fase di stesura del codice.

Il terzo errore, in riga 7, simula un utente abituato a programmare in un differente linguaggio che utilizza operatori differenti.

2.2.2 Esempio 2

```
1     {
2         int fibonacciSequence (int value){
3             if (value !> 1)[
4                 return value:
5             ]
6             return fib(value-1) + fib(value-2);
7
8
9         int num = 10;
10        print (fibonaciSequence(num));
11    }
```

Funzione ricorsiva per la valutazione dell'ennesimo numero della sequenza di Fibonacci.

Errore riportato all'interno del documento "Test_2.simplan_[Data].log" in output:

```
1     Errors:
2       [ERROR] Error at pos 3:18. mismatched input '!' expecting {')',
      '*', '/', '+', '-', '<', '>', '<=', '>=', '==', NEQ, '&&', '||'}
3       [ERROR] Error at pos 3:23. token recognition error at: '['
4       [ERROR] Error at pos 4:24. token recognition error at: ':'
5       [ERROR] Error at pos 5:8. token recognition error at: ']'
6       [ERROR] Error at pos 6:8. missing ';' at 'return'
7       [ERROR] Error at pos 9:4. extraneous input 'int' expecting {'if',
     'return', 'print', '{', '}', ID}
8       [ERROR] Error at pos 9:16. token recognition error at: ';'
9       [ERROR] Error at pos 10:4. missing ';' at 'print'
```

Errori riportati da SimpLanPlus sul codice riportato precedentemente

Gli errori in riga 3, 4, e 5 sono già stati analizzati nella sezione precedente.

Gli errori alla riga 6 e 10 sono dovuti ad errori nelle righe precedenti.

L'errore alla riga 9 simula un utente che copia una riga di codice da una sorgente esterna che utilizza una codifica differente. Il carattere ";" presente alla fine della riga 9, anche se uguale dal punto di vista grafico, ha una codifica numerica differente rispetto a quello standard.

2.2.3 Esempio 3

```
1     {
2         int addNumbers(int n) {
3             if (9 !== 0) return n + addNumbers(n - 1);
4             else return n;
5         }
6
7         int num1 = 10;
8         int result == addNumbers(num1);
9         printf(result);
10
11        int num2 = 10;
12        result = addNumbers(num2);
13        printf(result);
14    }
```

Funzione ricorsiva per la stampa della somma dei primi n numeri.

Errore riportato all'interno del documento "Test_3.simplan_[Data].log" in output:

```
1     Errors:
2       [ERROR] Error at pos 3:14. extraneous input '=' expecting {'(',
      '!', '-', BOOL, ID, NUMBER}
3       [ERROR] Error at pos 8:13. no viable alternative at input '
      intresult=='
4       [ERROR] Error at pos 11:2. extraneous input 'int' expecting {'if
      ', 'return', 'print', '{', '}', ID}
5
```

Errori codice test

L'errore alla riga 3 è già stato visto alla sezione precedente.

L'errore alla riga 8 è dovuto ad un carattere presente nel lexer, ma che non fa parte della sintassi di dichiarazione e assegnamento.

L'errore alla riga 11 è dovuto ad una dichiarazione di variabile in seguito ad uno statement. Questo non è consentito poichè, per definizione del linguaggio, devono essere definite prima tutte le dichiarazioni e poi tutti gli statement.

Da notare che in questo codice viene usato "printf" al posto della funzione "print" definita dal linguaggio, questo simula un utente abituato ad un linguaggio con sintassi differente. Questo errore non viene rilevato poichè l'analizzatore lessicale considera "printf" come una normale chiamata di funzione ed, in questa fase del progetto, non vi è alcun controllo sull'esistenza delle funzioni chiamate.

3 Esercizio 2

3.1 Obiettivo

Nel seguente esercizio viene richiesto di implementare nel linguaggio una tabella dei simboli sfruttando una lista di hash-table o un hash-table di liste.

Al termine dell'implementazione, SimpLanPlus deve essere in grado di verficare e riportare errori semantici come:

- Presenza di variabili/funzioni non dichiarate;
- Presenza di variabili/funzioni dichiarate più volte nello stesso ambiente.

3.2 Tabella dei simboli

La Symbol Table è stata implementata come una hash-table di liste (più di preciso Stack).

Considerando che le due soluzioni sono equivalenti nel risultato ottenuto e che entrambe hanno dei vantaggi e degli svantaggi, la scelta è stata fatta valutando il numero di aggiunte e rimozioni stimato nel corso del programma.

Dato che il numero di aggiunte e ricerche nella symbol table è generalmente superiore al numero di rimozioni, e dato che nel caso di HashMap di ArrayList la complessità di aggiunte e ricerche è O(1), abbiamo ritenuto ottimale implementare questa tipologia di tabella.

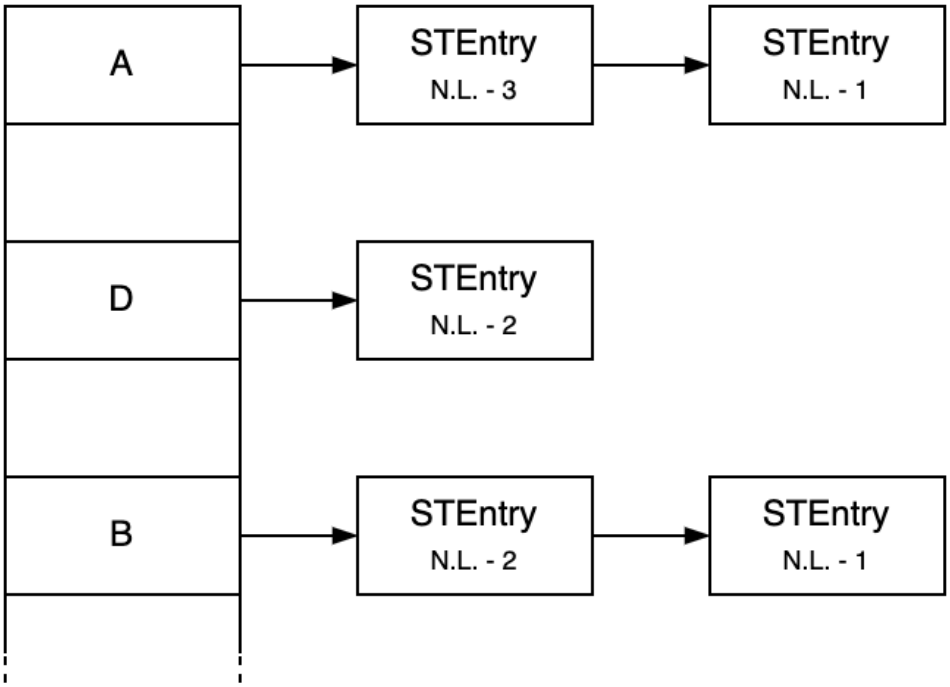

Figura 1: Rappresentazione grafica della struttura per la tabella dei simboli.

3.2.1 STEntry

- **nl**
  Rappresenta il livello di annidamento di riferimento della entry.
- **offset**
  Rappresenta il numero di byte che distanziano il frame pointer dalle celle in cui è allocata la variabile nello stack.
  Nel caso in cui la entry sia una funzione, questo valore è *-1*.
- **ID**
  Stringa identificativa della entry.
- **type**
  Tipo della entry.
  Nel caso in cui la entry sia una funzione, questo parametro sarà di tipo *FunctionSignature-Type* e conterrà i seguenti dati:
  - label
    Equivale a ID.
  - returnType
    Tipo di ritorno della funzione.
  - arguments
    Lista di parametri formali (*ArgNode*) della funzione.
- **effect**
  Parametro che permette di registrare lo stato di utilizzo di una determinata variabile, utile all' *Analisi Semantica*.
  Nel caso in cui la entry sia una funzione, questo valore è *None*.
- **isReference**
  Parametro che indica se la variabile di un determinato livello di annidamento è utilizzata per riferimento o per copia. Questo parametro è necessario in fase di *Code Generation* al fine di gestire le variabili passate per riferimento.

3.2.2 Funzionalità della tabella dei simboli

La classe "SymbolTableWrapper" contiene l'effettiva SymbolTable rappresentata come "HashMap<String, ArrayDeque<STentry»".

All'interno della classe sono presenti tre metodi principali utili alla gestione della SymbolTable:

- "addToSymbolTable(STentry entry)"
  Metodo che permette di aggiungere un STentry alla SymbolTable, verificando che non sia stato aggiunto in precedenza allo stesso livello.

- “findFirstInSymbolTable(String ID)”
  Metodo che permette di ricercare all’interno della SymbolTable la STentry in cima allo stack avente ID uguale a quello specificato.
- “removeLevelFromSymbolTable(int lev)”
  Metodo che permette di rimuovere tutte le STentry ad un determinato livello di nesting (questa funzione rappresenta la più costosa computazionalmente, con un costo pari a O(n), dove n indica il numero di entry nella Hash Table).

I restanti metodi sono utilizzati per lo svolgimento di operazioni sulla symbol table richieste dal compilatore.

3.3 Esempi SymbolTable

Nei seguenti codici è stata verificata la capacità di SimpLanPlus di generare tabelle dei simboli coerenti con quanto teorizzato.

3.3.1 Esempio 1

```
    // Funzione ricorsiva per la valutazione del valore pari/dispari
    {
        bool isEven (int value){
            if( value == 0){
                return true;
            } else {
                if (value == 1){
                    return false;
                } else {
                    return isEvens(num - 2);
                }
            }
        }

        bool num = 10;          // L'errore di tipo non viene rilevato
        print (isEven(value));
    }
```

Funzione ricorsiva per la valutazione di valore pari o dispari.

Output generato dalla compilazione del file "Test_1.slp":

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[WARN] You had 3 semantic errors:
  Fun isEvens does not exist in scope.
  Var num not declared.
  Var value not declared.
[ERROR] Program parsing failed.
```

Output generato dal codice

3.3.2 Esempio 2

```
// Funzione ricorsiva per la stampa dell'ennesimo numero di Fibonacci
{
    int num;
    int fibonacciSequence (int value){
        if (value <= 1){
            return value;
        }
        return fib(value-1) + fib(value-2);
    }

    int num = 12;
    print (fibonaciSequence(num));
}
```

Funzione ricorsiva per la valutazione dell'ennesimo numero della sequenza di Fibonacci.

Output generato dalla compilazione del file "Test_2.slp":

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[WARN] You had 4 semantic errors:
  Fun fib does not exist in scope.
  Fun fib does not exist in scope.
  Var num already declared.
  Fun fibonaciSequence does not exist in scope.
[ERROR] Program parsing failed.
```

Output generato dal codice

3.3.3 Esempio 3

```
1     // Funzione ricorsiva per la stampa della somma dei primi n numeri.
2     {
3         int addNumbers(int n) {
4             int print5(){ // Le funzioni non possono essere annidate,
      questo dara' errore.
5                 print 5;
6             }
7             if (n != 0) return n + addNumbers(n - 1);
8             else return n;
9         }
10
11        int addNumbers(int a, int b) {
12            return a + b;
13        }
14
15      int num = 10;
16      int result = addNumbers(num);
17      print(result);
18    }
```

Funzione ricorsiva per la stampa della somma dei primi n numeri.

Output generato dalla compilazione del file "Test_2.slp":

```
1     [INFO] Starting lexical verification.
2     line 4:18 mismatched input '(' expecting {';', '='}
3     [WARN] You had 1 errors while parsing the program:
4     Errors:
5       [ERROR] Error at pos 4:18. mismatched input '(' expecting {';',
      '='}
6     [ERROR] Program parsing failed.
```

Output generato dal codice

4 Esercizio 3

4.1 Obiettivo

Nel seguente esercizio viene richiesto di implementare nel linguaggio un'analisi semantica che verifichi:

- Correttezza dei tipi (in particolare numero e tipo dei parametri attuali se conformi al numero e tipo dei parametri formali);
- Utilizzo di variabili non inizializzate;
- Dichiarazione di variabili non utilizzate.

4.2 Prototipi inference rule

Per lo sviluppo delle regole di inferenza si è fatto riferimento ai seguenti prototipi:

$\Gamma : \Gamma_1 \times \Gamma_2 ... \times \Gamma_n$

$\Gamma : ID \rightarrow Type \times Offset \times \{Declared, Initialized, Used\}$

$\Gamma : ID_{Function} \rightarrow TypeFunction \times (TypeParam_1, TypeParam_2, TypeParam_3, ...) \times Offset$

$\Gamma, n \vdash Dec : \Gamma_1, n_1$

$\Gamma \vdash Stm : \Gamma_1, Type$

$\Gamma \vdash Exp : \Gamma_1, Type$

4.3 Inference rule

BlockProgram

$$\frac{[\,], 0 \vdash Dec : \Gamma_{Dec}, n_{Dec} \qquad \Gamma_{Dec} \vdash Stm : \Gamma_{Stm}, T_{Stm}}{\vdash \{Dec\ Stm\}}\ \text{[BlockProgram]}$$

Regola utilizzata nella gestione del blocco di un programma. I giudizi riportati permettono di gestire dichiarazioni ed istruzioni.

NOTA: Ad inizio programma l'ambiente è vuoto e l'offset parte da valore 0.

stmBlock

$$\frac{\Gamma \times [\,], 0 \vdash Dec : \Gamma_{Dec}, n_{Dec} \qquad \Gamma_{Dec} \vdash Stm : \Gamma_{Stm}, T_{Stm}}{\Gamma \vdash \{Dec\ Stm\} : pop(\Gamma_{Stm}), T_{Stm}}\ \text{[stmBlock]}$$

Regola utilizzata nella gestione dei blocchi di nuovi programmi interni al blocco programma principale. Mediante tale regola si gestiscono le dichiarazioni e le istruzioni del blocco interno. Se non sono individuati errori, la regola restituisce un ambiente contenente solo gli effetti prodotti dai giudizi, mediante l'utilizzo della funzione "pop(...)", ed il tipo ritornato dalla valutazione delle istruzioni.

decVarInt

$$\frac{ID \notin dom(top(\Gamma))}{\Gamma, n \vdash int\ ID\,; : \Gamma[ID \rightarrow (int, n, Declared)], n + 4}\ \text{[decVarInt]}$$

Regola utilizzata nella gestione di dichiarazione di variabili di tipo intero. Mediante tale regola si verifica che non vi siano nell'ambiente corrente variabili aventi il medesimo identificativo, in caso di riscontro positivo si procede con l'aggiornamento dell'ambiente e del valore di offset.

decVarBool

$$\frac{ID \notin dom(top(\Gamma))}{\Gamma, n \vdash bool\ ID\,; : \Gamma[ID \rightarrow (bool, n, Declared)], n + 1}\ \text{[decVarBool]}$$

Regola utilizzata nella gestione di dichiarazione di variabili di tipo booleano. Mediante tale regola si verifica che non vi siano nell'ambiente corrente variabili aventi il medesimo identificativo, in caso di riscontro positivo si procede con l'aggiornamento dell'ambiente e del valore di offset.

decVarIntAsgm

$$\frac{ID \notin dom(top(\Gamma)) \qquad \Gamma \vdash Exp : \Gamma_1, int}{\Gamma, n \vdash int\ ID = Exp\,;\ : \Gamma_1[ID \rightarrow (int, n, Initialized)], n + 4} \text{ [decVarIntAsgm]}$$

Regola utilizzata nella gestione di dichiarazione con inizializzazione di variabili di tipo intero. Mediante tale regola si verifica che non vi siano nell'ambiente corrente variabili aventi il medesimo identificativo e che il tipo dell'espressione sia intero (come il tipo della variabile). In caso di riscontro positivo si procede con l'aggiornamento dell'ambiente, il quale corrisponde a quello modificato dall'espressione, e del valore di offset.

decVarBoolAsgm

$$\frac{ID \notin dom(top(\Gamma)) \qquad \Gamma \vdash Exp : \Gamma_1, bool}{\Gamma, n \vdash bool\ ID = Exp\,;\ : \Gamma_1[ID \rightarrow (bool, n, Initialized)], n + 1} \text{ [decVarBoolAsgm]}$$

Regola utilizzata nella gestione di dichiarazione con inizializzazione di variabili di tipo booleano. Mediante tale regola si verifica che non vi siano nell'ambiente corrente variabili aventi il medesimo identificativo e che il tipo dell'espressione sia booleano (come il tipo della variabile). In caso di riscontro positivo si procede con l'aggiornamento dell'ambiente, il quale corrisponde a quello modificato dall'espressione, e del valore di offset.

decVars

$$\frac{\Gamma, n \vdash Dec_1 : \Gamma_{Dec_1}, n_{Dec_1} \qquad \Gamma_{Dec_1}, n_{Dec_1} \vdash Dec_2 : \Gamma_{Dec_2}, n_{Dec_2}}{\Gamma, n \vdash Dec_1\,;\ Dec_2\ : \Gamma_{Dec_2}, n_{Dec_2}} \text{ [decVars]}$$

Regola utilizzata nella gestione di dichiarazione multiple.

stmAsgm

$$\frac{ID \in dom(\Gamma) \quad \Gamma(ID) = (T_{Id}, Offset_{Id}, Status_{Id}) \quad \Gamma \vdash Exp : \Gamma_{Exp}, T_{Exp} \quad T_{Id} = T_{Exp}}{\Gamma \vdash ID = Exp\,; : \Gamma_{Exp}[ID \rightarrow (T_{Id}, Offset_{Id}, max(Initialized, Status_{Id}))], Void} \text{ [stmAsgm]}$$

Regola utilizzata nella gestione di assegnamento di un'espressione ad una variabile. Mediante tale regola si verifica che vi sia nell'ambiente la variabile avente l'identificativo riportato e che il tipo dell'espressione sia concorde. In caso di riscontro positivo si procede con l'aggiornamento dell'ambiente, il quale corrisponde a quello modificato dall'espressione, andando a modificare in caso l'effetto associato alla variabile.

NOTE: La funzione "max(...)" restituisce lo stato maggiore rispetto lo stato presente nell'ambiente e lo stato di "Inizialized".

stmPrint

$$\frac{\Gamma \vdash Exp : \Gamma_{Exp}, T_{Exp}}{\Gamma \vdash printExp : \Gamma', Void} \text{ [stmPrint]}$$

Regola utilizzata nella gestione di stampa di un'espressione. Mediante tale regola si controlla quanto contenuto nell'espressione. In caso di riscontro positivo si procede con l'aggiornamento dell'ambiente, il quale corrisponde a quello modificato dall'espressione.

stmEmptyReturn

$$\frac{}{\Gamma \vdash Return : \Gamma, Void} \text{ [stmEmptyReturn]}$$

Regola utilizzata nella gestione del ritorno di una funzione mancate di un'espressione.

stmReturn

$$\frac{\Gamma \vdash Exp : \Gamma_{Exp}, T_{Exp}}{\Gamma \vdash ReturnExp : \Gamma_{Exp}, T_{Exp}} \text{ [stmReturn]}$$

Regola utilizzata nella gestione del ritorno di una funzione avente un'espressione. Mediante tale regola si controlla quanto contenuto nell'espressione, si procede con l'aggiornamento dell'ambiente, il quale corrisponde a quello modificato dall'espressione, ed il ritorno del tipo valutato.

stmIfThenElse

$$\frac{\begin{array}{c}\Gamma \vdash Exp : \Gamma_{Exp}, bool \\ \Gamma_{Exp} \vdash Stm_1 : \Gamma_{Stm_1}, T_{Stm_1} \qquad \Gamma_{Exp} \vdash Stm_2 : \Gamma_{Stm_2}, T_{Stm_2}\end{array}}{\Gamma \vdash if(Exp)\ Stm_1\ else\ Stm_2\ :\ evAmbiente(\Gamma_{Stm_1}, \Gamma_{Stm_2}), evTipo(T_{Stm1}, T_{Stm2})} \text{ [stmIfThenElse]}$$

Regola utilizzata nella gestione dell'if-then-else. Mediante tale regola si controlla quanto contenuto nell'espressione imponendo il tipo booleano. In caso di condizione verificata si procede alla valutazione del contenuto dei due branch, dando in ingresso l'ambiente aggiornato dalla precedente espressione. In caso di condizioni verificate, viene valutato l'ambiente da ritornare ed il tipo attraverso due funzioni personalizzate.

NOTA:

Tabella evAmbiente per l'ambiente aggiornato.

$evAmbiente$	Declared	Initialized	Used
Declared	Declared	Declared	Declared
Initialized	Declared	Initialized	Used
Used	Declared	Used	Used

Tabella evTipo per il tipo aggiornato.

$evTipo$	void	int	bool
void	void	voidable(int)	voidable(bool)
int	voidable(int)	int	Error
bool	voidable(bool)	Error	bool

Di base il codice supporta 3 tipi principali: *Void* (solo per le funzioni), *Int* e *Bool*.
Per il corretto controllo dei tipi è stato necessario introdurre un tipo intermedio, capace di rappresentare un elemento il cui tipo di ritorno, in fase di compilazione, può essere allo stesso tempo void e int (o bool).
Questa situazione si ha principalmente in caso di ITE in cui un solo ramo ha un *return*.

stmIfThen

$$\frac{\Gamma \vdash Exp : \Gamma_{Exp}, bool \qquad \Gamma_{Exp} \vdash Stm : \Gamma_{Stm}, T_{Stm}}{\Gamma \vdash if(Exp)\ Stm\ :\ \Gamma_{Stm}, T_{Stm}}\ \text{[stmIfThen]}$$

Regola utilizzata nella gestione dell'if-then. Mediante tale regola si controlla quanto contenuto nell'espressione imponendo il tipo booleano. In caso di condizione verificata si procede alla valutazione del contenuto del branch, dando in ingresso l'ambiente aggiornato dalla precedente espressione. In caso di condizioni verificate, viene valutato l'ambiente da ritornare ed il tipo.

Stms

$$\frac{\Gamma \vdash Stm_1 : \Gamma_{Stm_1}, T_{Stm_1} \qquad \Gamma_{Stm_1} \vdash Stm_2 : \Gamma_{Stm_2}, T_{Stm_2}}{\Gamma \vdash Stm_1 \,;\, Stm_2 \;:\; \Gamma_{Stm_2}, max^*(T_{Stm_1}, T_{Stm_2})} \text{ [Stms]}$$

Regola utilizzata nella gestione della concatenazione di diverse istruzioni.

max^*	void	int	bool	voidable(int)	voidable(bool)
void	void	int	bool	voidable(int)	voidable(bool)
int	int	int	Error	int	Error
bool	bool	Error	bool	Error	bool
voidable(int)	voidable(int)	int	Error	voidable(int)	Error
voidable(bool)	voidable(bool)	Error	bool	Error	voidable(bool)

Num

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

Regola utilizzata per la gestione degli assiomi relativi a numeri interi.

Bool

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

Regola utilizzata per la gestione degli assiomi relativi a valori booleani.

ID

$$\frac{ID \in dom(\Gamma) \qquad \Gamma(ID) = (T_{Id}, Offset_{Id}, Status_{Id} > Declared)}{\Gamma \vdash ID : \Gamma[ID \rightarrow (T_{Id}, Offset_{Id}, Used)], T_{Id}} \text{ [ID]}$$

Regola utilizzata per la gestione degli assiomi relativi a identificativi di variabili. Mediante tale regola si controlla se la variabile appartiene al dominio dell'ambiente e se il suo stato è "Initialized" o "Used". In caso di condizione verificata si aggiorna l'ambiente, modificando lo stato della variabili ad "Used" e si ritorna il suo tipo.

AND

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, bool \qquad \Gamma_{Exp_1} \vdash Exp_2 : \Gamma_{Exp_2}, bool}{\Gamma \vdash Exp_1 \ \&\& \ Exp_2 \ : \ \Gamma_{Exp_2}, bool} \text{ [AND]}$$

Regola utilizzata per la gestione dell'operatore "&&". Mediante tale regola si verifica il tipo ed il contenuto delle espressioni, aggiornando di conseguenza l'ambiente. In caso di condizione verificata si ritorna l'ultimo ambiente aggiornato e il tipo boolean.

OR

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, bool \qquad \Gamma_{Exp_1} \vdash Exp_2 : \Gamma_{Exp_2}, bool}{\Gamma \vdash Exp_1 \ || \ Exp_2 \ : \ \Gamma_{Exp_2}, bool} \text{ [OR]}$$

Regola utilizzata per la gestione dell'operatore "||". Mediante tale regola si verifica il tipo ed il contenuto delle espressioni, aggiornando di conseguenza l'ambiente. In caso di condizione verificata si ritorna l'ultimo ambiente aggiornato e il tipo boolean.

arithmeticOp

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, int \qquad \Gamma_{Exp_1} \vdash Exp_2 : \Gamma_{Exp_2}, int}{\Gamma \vdash Exp_1 \ \#_1 \ Exp_2 : \Gamma_{Exp_2}, int} \quad [+, -, *, /, \% \,]$$

Regola utilizzata per la gestione degli operatori aritmetici. Mediante tale regola si verifica il tipo ed il contenuto delle espressioni, aggiornando di conseguenza l'ambiente. In caso di condizione verificata si ritorna l'ultimo ambiente aggiornato e il tipo intero.

logicOp

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, int \qquad \Gamma_{Exp_1} \vdash Exp_2 : \Gamma_{Exp_2}, int}{\Gamma \vdash Exp_1 \ \#_2 \ Exp_2 : \Gamma_{Exp_2}, bool} \quad [>, <, >=, <= \,]$$

Regola utilizzata per la gestione degli operatori logici. Mediante tale regola si verifica il tipo ed il contenuto delle espressioni, aggiornando di conseguenza l'ambiente. In caso di condizione verificata si ritorna l'ultimo ambiente aggiornato e il tipo boolean.

eqOp

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, T_{Exp_1} \qquad \Gamma_{Exp_1} \vdash Exp_2 : \Gamma_{Exp_2}, T_{Exp_2} \qquad T_{Exp_1} = T_{Exp_2}}{\Gamma \vdash Exp_1 \ \#_3 \ Exp_2 : \Gamma_{Exp_2}, bool} \quad [!=, ==]$$

Regola utilizzata per la gestione degli operatori di uguaglianza. Mediante tale regola si verifica il contenuto delle espressioni, aggiornando di conseguenza l'ambiente, ed il tipo, controllando che siano concordi. In caso di condizione verificata si ritorna l'ultimo ambiente aggiornato e il tipo boolean.

NOT

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, bool}{\Gamma \vdash !Exp_1 : \Gamma_{Exp_1}, bool} \text{ [NOT]}$$

Regola utilizzata per la gestione degli operatore di negazione. Mediante tale regola si verifica il contenuto dell'espressione, aggiornando di conseguenza l'ambiente, e verificando che sia di tipo boolean. In caso di condizione verificata si ritorna ambiente aggiornato e il tipo boolean.

NEG

$$\frac{\Gamma \vdash Exp_1 : \Gamma_{Exp_1}, int}{\Gamma \vdash -Exp_1 : \Gamma_{Exp_1}, int} \text{ [NEG]}$$

Regola utilizzata per la gestione degli operatore di inversione di segno. Mediante tale regola si verifica il contenuto dell'espressione, aggiornando di conseguenza l'ambiente, e verificando che sia di tipo intero. In caso di condizione verificata si ritorna ambiente aggiornato e il tipo intero.

decFun

$$\frac{\begin{array}{c} Fun_{ID} \notin dom(top(\Gamma)) \\ \Gamma' = \Gamma[Fun_{ID} \rightarrow (T_{ID}, (T_i, ..., T_i'...), Offset_{Fun_{ID}}), \\ x_i \rightarrow \{T_i, Initialized, Offset_{x_i}\}, ..., \\ x_i' \rightarrow \{T_i', Initialized, Offset_{x_i'}\}, ...] \\ n' = 0 + \sum_{i=1}^{n} dim(x_i) + \sum_{i=1}^{m} dim(x_i') \\ \Gamma', n' \vdash DecsVar : \Gamma'', n'' \qquad \Gamma'' \vdash Stms : \Gamma''', T_{Stm} \\ T_{ID} = T_{Stm} \qquad \Gamma_{ID} = \Gamma[Fun_{ID} \rightarrow (T_{ID}, (T_i, ..., T_i'...))] \end{array}}{\Gamma \vdash T_{ID} Fun_{ID}(T_1 x_1, ..., T_n x_n, var T_1' x_1', ..., T_m' x_m') \{DecsVar\ Stms\} : \Gamma_{ID}} \text{ [decFun]}$$

Regola utilizzata per la definizione di una funzione. Mediante tale regola si verifica in ordine l'unicità della funzione nel dominio dell'ambiente e lo stato dei parametri richiesto. Viene valutato l'ambiente creato dalla funzione, verificando che il tipo di ritorno corrisponda a quello definito dalla funzione. In caso di condizione verificata si ritorna ambiente aggiornato contente la firma della funzione.
NOTA: L'offset di una funzione ha valore -1.
NOTA: n' non calcola lo spazio necessaria al salvataggio del registro $ra.
NOTA: La funzione "dim" calcola lo spazio occupato nello stack per ogni variabile passa in input.

callExp

$$\frac{Fun_{ID} \in dom(\Gamma) \qquad \Gamma, n \vdash Exp_1 : \Gamma_1, T_{e_1} \qquad \Gamma_i, n \vdash Exp_{i+1} : \Gamma_{i+1}, T_{e_{i+1}} \qquad T_{pi} = T_{ei}}{\Gamma \vdash ID(Exp_1, ..., Exp_n) : \Gamma', T_{Fun_{ID}}} \text{ [CallExp]}$$

Regola utilizzata per il richiamo di una funzione all'interno di un'espressione. Mediante tale regola si verifica in ordine l'esistenza della funzione nel dominio dell'ambiente e il contenuto delle espressioni con il conseguente aggiornamento. In caso di condizione verificata si ritorna ambiente aggiornato e il tipo di ritorno della funzione.

CallStm

$$\frac{Fun_{ID} \in dom(\Gamma) \qquad \Gamma, n \vdash Exp_1 : \Gamma_1, T_{e_1} \qquad \Gamma_i, n \vdash Exp_{i+1} : \Gamma_{i+1}, T_{e_{i+1}} \qquad T_{pi} = T_{ei}}{\Gamma \vdash ID(Exp_1, ..., Exp_n) : \Gamma', Void} \text{ [callStm]}$$

Regola utilizzata per il richiamo di una funzione. Mediante tale regola si verifica in ordine l'esistenza della funzione nel dominio dell'ambiente e il contenuto delle espressioni con il conseguente aggiornamento. In caso di condizione verificata si ritorna ambiente aggiornato.

4.4 Esempi di typecheck

4.4.1 Esempio 1

```
1   {
2       int a;
3       int b;
4       int c = 1;
5
6       if(c > 1){
7           b = c;
8       }else{
9           a = b;
10      }
11  }
```

Test typecheck 1.

Errore riportato all'interno del terminale:

```
1   [INFO] Starting lexical verification.
2   [INFO] No lexical error found.
3   [INFO] Starting semantic verification.
4   [INFO] No semantic error found.
5   [SUCCESS] Program parsing completed.
6   [INFO] Starting type check.
7   [ERROR] Error while running type check: Variable not initialized: b
```

Output per il typecheck 1.

Il risultato è in linea con quanto atteso. La variabile con identificativo "b" viene dichiarata alla linea 3. Quando viene effettuata la valutazione dell'if-then-else e si valutano singolarmente i due branch si incorre nell'errore:

- Valutazione branch true
  Nella valutazione della riga 7 si verifica la presenza della variabile "b" nell'ambiente, l'espressione (in questo caso il richiamo di una variabile che deve verificare la presenza della variabile "c" ed il suo stato deve essere "Initialized" o "Used") e la corrispondenza dei tipi. Dal momento che la valutazione dell'assegnamento non presenta errori, la variabile "b" da dichiarata passa allo stato di inizializzato.

- Valutazione branch false
  Nella valutazione della riga 9 si verifica la presenza della variabile "a" nell'ambiente, l'espressione (in questo caso il richiamo di una variabile che deve verificare la presenza della variabile "b" ed il suo stato deve essere "Initialized" o "Used") e la corrispondenza dei tipi. Dal momento che la valutazione dell'assegnamento presenta errori (la variabile "b" ha lo stato "Declared"), viene riportato il problema ed il typecheck termina.

4.4.2 Esempio 2

```
1     {
2         int a;
3         int b;
4         int c;
5
6         void f(int n, var int x){
7             x = n ;
8         }
9     }
10
11    f(1,a) ; f(2,b) ; f(3,c);
12    }
```

Test typecheck 2.

Errore riportato all'interno del terminale:

```
1     [INFO] Starting lexical verification.
2     [INFO] No lexical error found.
3     [INFO] Starting semantic verification.
4     [INFO] No semantic error found.
5     [SUCCESS] Program parsing completed.
6     [INFO] Starting type check.
7     [WARNING] Variable x not used
8
9     [ERROR] Error while running type check: Variable not initialized: a
```

Output per il typecheck 2.

Il risultato è in linea con quanto atteso. La variabile con identificativo "a" viene dichiarata alla linea 2. Quando viene effettuata la valutazione del richiamo di funzione si incorre nell'errore. Questo è dovuto al fatto che la variabile richiamata non risulta essere inizializzata.

Per scelta progettuale si è deciso di imporre per i parametri attuali l'utilizzo di costanti o di variabili obbligatoriamente inizializzate.

4.4.3 Esempio 3

```
1   {
2       int a;
3       int b;
4       int c = 1;
5
6       void h(int n, var int x, var int y, var int z){
7           if(n==0) return ;
8           else{
9               x = y;
10              h(n-1,y,z,x);
11          }
12      }
13
14      h(5,a,b,c);
15  }
```

Test typecheck 3.

Errore riportato all'interno del terminale:

```
1   [INFO] Starting lexical verification.
2   [INFO] No lexical error found.
3   [INFO] Starting semantic verification.
4   [INFO] No semantic error found.
5   [SUCCESS] Program parsing completed.
6   [INFO] Starting type check.
7   [ERROR] Error while running type check: Variable not initialized: a
```

Output per il typecheck 3.

Il risultato è in linea con quanto atteso. Come nell'esempio precedente, la variabile con identificativo "a" viene dichiarata alla linea 2. Quando viene effettuata la valutazione del richiamo di funzione si incorre nell'errore. Questo è dovuto al fatto che la variabile richiamata non risulta essere inizializzata.

Per scelta progettuale si è deciso di imporre per i parametri attuali l'utilizzo di costanti o di variabili obbligatoriamente inizializzate.

5 Esercizio 4

5.1 Obiettivo

Nel seguente esercizio viene richiesto di definire un linguaggio bytecode per eseguire programmi in SimpLanPlus ed implementare l'interprete. In particolare devono essere rispettate le seguenti condizioni:

- Il bytecode deve avere istruzioni per una macchina a pila che memorizza in un apposito registro il valore dell'ultima istruzione calcolata;
- Implementare l'interprete per il bytecode;
- Compilare ed eseguire i programmi del linguaggio ad alto livello.

5.2 Note sul linguaggio bytecode implementato

5.2.1 Sintassi di riferimento

Il linguaggio implementato cerca di seguire il più possibile la struttura implementata nel linguaggio Assembly MIPS[1].

In primis questo significa adottare una sintassi *Intel* e non *AT&T*. La differenza principale tra le due si trova nell'ordine in cui i parametri delle operazioni vengono forniti:

- **Intel**: operator destination source
- **AT&T**: operator source destination

5.2.2 Differenze rispetto a sintassi MIPS

- La sintassi da noi sviluppata non presenta operazioni unsigned.
- Salviamo il risultato di *mult* nel registro destinazione specificato invece che nei registri *high* (32 bit più significativi) e *low* (32 bit meno significativi).
- Salviamo il risultato di *div* nel registro destinazione specificato (applicando una divisione tra interi) invece che nei registri *high* (quoziente) e *low* (resto).
  Questo cambiamento ha portato all'aggiunta dell'operatore *mod*, il quale salva nel registro destinazione specificato il resto della divisione.
- Le operazioni di confronto aritmetiche (<, <=, >, >=) presentano un nome diverso rispetto a quello implementato in MIPS (perdono la s iniziale) e non presentano la versione "immediate".
- La nostra sintassi implementa solo l'operazione di break *beq*, tutte le altre operazioni si possono ottenere con una combinazione di operazioni di confronto aritmetiche e break (*ble* nella nostra sintassi diventa le + beq).

[1] MIPS Instruction Set: https://www.dsi.unive.it/ gasparetto/materials/MIPS_Instruction_Set.pdf
MIPS user manual: https://www.cs.unibo.it/ solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf

5.3 Note sulla code generation

5.3.1 Code Generation DerExp

La code generation delle *DerExp* è particolare dato che deve gestire variabili sia passate per copia che per riferimento.

La funzione prende in input un parametro di tipo stringa contenente le eventuali opzioni necessarie.

Passaggio di una variabile come parametro attuale con modificatore *var* si ha se alla funzione viene passata l'opzione *getAddress*.

Se la variabile è un parametro attuale con modificatore *var* viene caricato nel registro di destinazione il puntatore a cui sta puntando la variabile attuale.

Se la variabile non è un parametro attuale con modificatore *var* viene calcolato l'indirizzo della variabile e viene salvato nel registro destinazione.

Lettura di una variabile si ha se non viene passata nessuna opzione alla funzione.

Se la variabile è un parametro attuale con modificatore *var*

1. viene calcolato l'indirizzo a cui punta la variabile attuale
2. viene salvato nel registro destinazione il valore contenuto nella cella puntata

Se la variabile non è un parametro attuale con modificatore *var* viene salvato nel registro destinazione il valore contenuto nella cella.

5.3.2 Code Generation Return

Il return oltre a salvare l'eventuale valore da ritornare deve eseguire tutte le operazioni necessarie per terminare la funzione, evitando l'esecuzione di istruzioni successive.

Per terminare correttamente una funzione è necessario rimuovere tutti gli ambienti da essa creati. Per ottenere tale risultato si è deciso di inserire una *label* prima del codice che gestisce l'uscita dall'ambiente della funzione e fare si che il return salti direttamente a questa label.
Da qui in avanti, si farà riferimento alla porzione di codice atta alla terminazione di un ambiente (o di una funzione) come *footer* di quest'ultima.
Al fine di consentire al return di saltare al *footer* della funzione, quest'ultimo inserisce una label fittizia, denominata *RETURN_CHAIN_PLACEHOLDER*, la quale verrà sostituita nel footer della funzione con la corretta label.

Questa implementazione porta alla corretta terminazione dell'ambiente della funzione, ma non a quella degli eventuali ambienti annidati in essa.

Per risolvere questo problema è stata inserita una label nel *footer* di ogni ambiente e, utilizzando la stessa logica descritta sopra, le *jump* sono state configurate per saltare al *footer* dell'ambiente subito precedente utilizzando la label fittizia *BLOCK_CHAIN_PLACEHOLDER*.

La soluzione descritta porta ad un funzionamento corretto del return nella maggior parte dei casi, ma introduce un problema nel caso in cui vi sia un return che non è detto venga raggiunto, come nel caso mostrato di seguito:

```
1  {
2      // Ritorna true se n e' multiplo di 3 e di 5
3      bool f(int n){
4          if (n%3==0) {
5              if (n%5==0) {
6                  return true;
7              }
8          }
9          return false;
10     }
11 }
```

La funzione ritorna true se il numero inserito è multiplo sia di 3 che di 5, false altrimenti.

Se il numero inserito non è multiplo di 3 o è multiplo sia di 3 che di 5, questo programma funziona, se però il numero è multiplo di 3 ma non di 5, il programma arriva alla fine del primo *if* e, essendoci un return all'interno, procede a saltare al *footer* della funzione, ignorando il return false.

Questo problema non è risolvibile in fase di code generation, poichè dipende da una condizione nota solo in fase di esecuzione.

Per questo motivo è stato aggiunto un registro speciale $ret, questo registro verrà impostato a true ogni volta che si raggiunge un istruzione return e a false ogni volta che si raggiunge il *footer* di una funzione.

I footer degli ambienti intermedi effettueranno la *jump* al *footer* dell'ambiente precedente solo se il registro $ret ha valore true (ovvero se l'esecuzione ha raggiunto un return).

Con questo accorgimento il sistema precedentemente descritto è in grado di terminare in qualunque punto della funzione e a qualunque nesting level.

5.4 Struttura memoria del programma

5.4.1 Ambiente

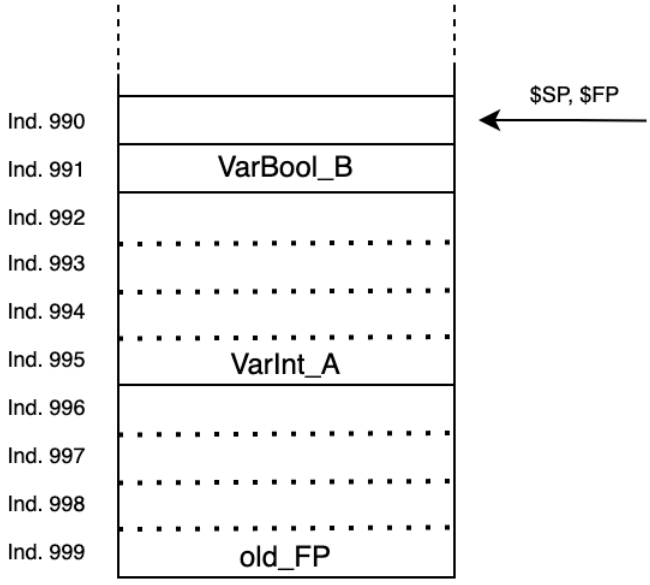

Figura 2: Struttura della memoria all'inizio di un ambiente.

5.4.2 Funzione

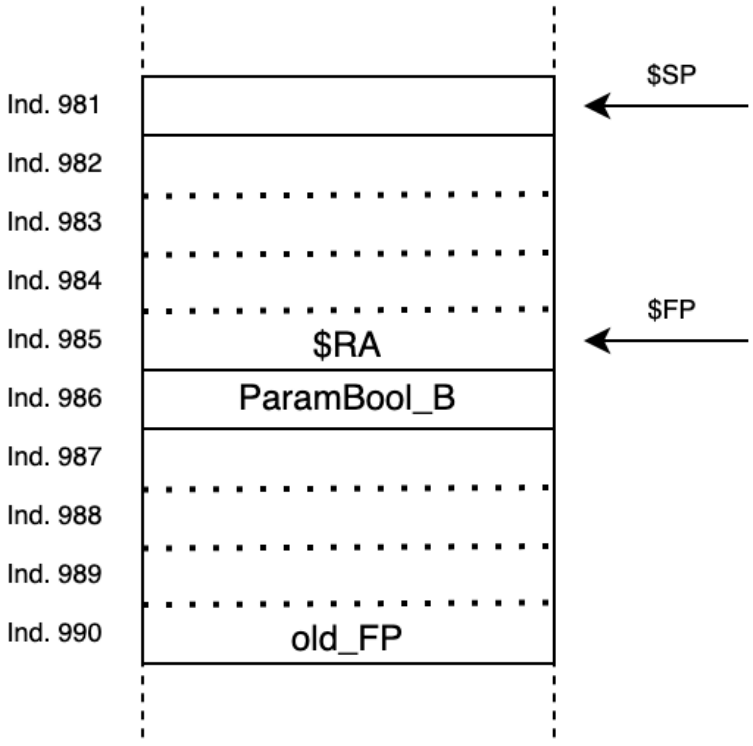

Figura 3: Struttura della memoria a seguito di una chiamata di funzione *f(bool b)*.

5.5 Esempi di esecuzione

5.5.1 Esempio 1 versione 1

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

    f(10);
}
```

Codegen 1.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
You had 3 semantic errors:
  Var x not declared.
  Var x not declared.
  Var x not declared.
[ERROR] Program parsing failed
```

Generazione del codice intermedio per la Codegen 1 versione 1.

Il codice in esame prova ad accedere ad una variabile globale dall'interno di una funzione, essendo questa un'operazione non consentita, il programma termina in fase di compilazione e non viene generato alcun file di output.

5.5.2 Esempio 1 versione 2

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

    f(10, x);
}
```

Codegen 1.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.
```

Generazione del codice intermedio per la Codegen 1 versione 2.

Generato il codice intermedio nel file “test1_2.asm” si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifElse_0 not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:
       3628800

[INFO] Program terminated correctly.
[INFO] Your program used 230/10000 bytes of memory. (2.3 %)
[INFO] Your program took ~1 ms to complete.
```

Esecuzione del codice intermedio per la Codegen 1 versione 2.

5.5.3 Esempio 2

```
{
    int u = 1 ;
    void f(var int x, int n){
        if (n == 0) {
            print(x) ;
        } else {
            int y = x * n ;
            f(y,n-1);
        }
    }

    f(u,6) ;
}
```

Codegen 2.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.
```

Generazione del codice intermedio per la Codegen 2.

Generato il codice intermedio nel file “test2.asm” si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifElse_0 not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:
       720

[INFO] Program terminated correctly.
[INFO] Your program used 174/10000 bytes of memory.
(1.7399999999999998 %)
[INFO] Your program took ~1 ms to complete.
```

Esecuzione del codice intermedio per la Codegen 2.

5.5.4 Esempio 3 versione 1

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

    f(5,4);
}
```

Codegen 3 versione 1.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[WARNING] Variable x not used

[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.
```

Generazione del codice intermedio per la Codegen 3 versione 1.

Generato il codice intermedio nel file “test3_1.asm” si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifElse_0 not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:
       9

[INFO] Program terminated correctly.
[INFO] Your program used 30/10000 bytes of memory. (0.3 %)
[INFO] Your program took ~1 ms to complete.
```

Esecuzione del codice intermedio per la Codegen 3 versione 1.

5.5.5 Esempio 3 versione 2

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

    f(4,5);
}
```

Codegen 3 versione 2.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[WARNING] Variable x not used

[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.
```

Generazione del codice intermedio per la Codegen 3 versione 2.

Generato il codice intermedio nel file “test3_2.asm” si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifElse_0 not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:

[ERROR] Out of memory with instruction PushInt.
```

Esecuzione del codice intermedio per la Codegen 3 versione 2.

Il programma finisce per dare un errore di “Out of memory” a causa di un loop infinito dato dai parametri in input alla funzione e dalle operazione che quest’ultima esegue, le quali portano il programma ad allocare ad ogni ciclo nuovo spazio in memoria fino al superamento del limite.

6 Esempi di funzionamento

Nella seguente sezione sono presentati esempi di codice supplementari che permettono di evidenziare le potenzialità di SimpLanPlus.

6.1 Successione di Fibonacci

```
1   {
2       int fib(int n){
3           if (n<=1) return n;
4           else return fib(n-1) + fib(n-2);
5       }
6
7       int res;
8       int val;
9       int correct;
10      int errorCount = 0;
11
12      // Se i test risultano corretti il programma stampa true,
    altrimenti stampa:
13      // valore input
14      // valore atteso
15      // valore ottenuto
16
17      val = 5;
18      correct = 5;
19      res = fib(val);
20
21      if (res != correct) {
22          print(val);
23          print(correct);
24          print(res);
25          errorCount = errorCount + 1;
26      }
27
28      // NOTA: Altro codice simile a quello riportato sopra, ma
    riportante valori differenti.
29
30      if (errorCount == 0) print(true);
31      else print(false);
32  }
```

Successione di Fibonacci.

Il codice riportato nel documento, come indicato nel commento alla riga 28, non coincide con quello testato effettivamente.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.
```

Generazione del codice intermedio per la Successione di Fibonacci.

Generato il codice intermedio nel file “fibonacci.asm” si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifElse_0 not found, generating placeholder
[WARNING] Label fib_0_footer not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
[WARNING] Label ifEnd_1 not found, generating placeholder
[WARNING] Label ifEnd_2 not found, generating placeholder
[WARNING] Label ifEnd_3 not found, generating placeholder
[WARNING] Label ifEnd_4 not found, generating placeholder
[WARNING] Label ifElse_5 not found, generating placeholder
[WARNING] Label ifEnd_5 not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:
true

[INFO] Program terminated correctly.
[INFO] Your program used 330/10000 bytes of memory.
(3.3000000000000003 %)
[INFO] Your program took ~322 ms to complete.
```

Esecuzione del codice intermedio per la Successione di Fibonacci.

6.2 Minimo comune multiplo

```
{

    // ATTENZIONE: Questo codice di test richiede l'operatore % che non e' implementato di default in SLP

    void mcm1Aux(int n1, int n2, int i, var int gcd) {
        if (i> n1 || i> n2) return;
        if (n1 % i == 0 && n2 % i == 0) gcd = i;
        mcm1Aux(n1, n2, i+1, gcd);
    }

    int mcm1(int n1, int n2){
        int gcd = 0;
        if (n1 == n2) return n1;
        mcm1Aux(n1, n2, 1, gcd);
        return (n1*n2/gcd);
    }

    int mcm2Aux(int n1, int n2, int max) {
        if (max % n1 == 0 && max % n2 == 0) return max;
        return mcm2Aux(n1, n2, max+1);
    }

    int mcm2(int n1, int n2) {
        if (n1 == n2) return n1;
        if (n1 >= n2) return mcm2Aux(n1, n2, n1);
        else return mcm2Aux(n1, n2, n2);
    }

    print(mcm1(72,120));
    print(mcm2(72,120));
}
```

Minimo comune multiplo.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.
```

Generazione del codice intermedio per il Minimo comune multiplo.

Generato il codice intermedio nel file "mcm.asm" si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
[WARNING] Label mcm1Aux_0_footer not found, generating placeholder
[WARNING] Label ifEnd_1 not found, generating placeholder
[WARNING] Label ifEnd_2 not found, generating placeholder
[WARNING] Label mcm1_0_footer not found, generating placeholder
[WARNING] Label ifEnd_3 not found, generating placeholder
[WARNING] Label mcm2Aux_0_footer not found, generating placeholder
[WARNING] Label ifEnd_4 not found, generating placeholder
[WARNING] Label mcm2_0_footer not found, generating placeholder
[WARNING] Label ifElse_5 not found, generating placeholder
[WARNING] Label ifEnd_5 not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:
       360
       360

[INFO] Program terminated correctly.
[INFO] Your program used 4847/10000 bytes of memory. (48.47 %)
[INFO] Your program took ~12 ms to complete.
```

Esecuzione del codice intermedio per il Minimo comune multiplo.

6.3 Congettura di Goldbach

```
/*
    La congettura di Goldbach afferma che ogni numero intero pari maggiore di 2 puo' essere scritto come somma di due numeri primi (non necessariamente distinti).
    Scrivere una funzione che dato un numero n ritorni:
    - 0 se n non e' un intero pari maggiore di 2
    - il piu' piccolo numero primo p1 tale per cui p2 = n - p1 sia un numero primo
*/

{
    bool isPrimeAux(int n, int i){
        if(n==i || n==1) return true;
        if(n%i==0 || n<i) return false;
        return isPrimeAux(n, i+1);
    }

    bool isPrime(int n){
        return isPrimeAux(n, 2);
    }

    int goldbachAux(int n, int i){
        if (i>=n) return 0;
        if (isPrime(i)){
            if(isPrime(n-i)) {
                return i;
            }
        }
        return goldbachAux(n, i+1);
    }

    int goldbach(int n){
        if ((n<=2) || (n%2!=0)) return 0;
        return goldbachAux(n, 2);
    }

    int i = 14;
    int n = goldbach(i);

    print(n);
    print(i-n);
}
```

Congettura di Goldbach.

Output riportato nel terminale in fase di generazione del codice intermedio:

```
[INFO] Starting lexical verification.
[INFO] No lexical error found.
[INFO] Starting semantic verification.
[INFO] No semantic error found.
[SUCCESS] Program parsing completed.
[INFO] Starting type check.
[INFO] No type check errors found.
[SUCCESS] Code generated! Assembling and running generated code.

```

Generazione del codice intermedio per il congettura di Goldbach.

Generato il codice intermedio nel file “goldbach.asm” si procede alla sua valutazione ed esecuzione.

```
[INFO] Starting asm code lexical verification.
[WARNING] Label block_0 not found, generating placeholder
[WARNING] Label ifEnd_0 not found, generating placeholder
[WARNING] Label isPrimeAux_0_footer not found, generating
placeholder
[WARNING] Label ifEnd_1 not found, generating placeholder
[WARNING] Label isPrime_0_footer not found, generating placeholder
[WARNING] Label ifEnd_2 not found, generating placeholder
[WARNING] Label goldbachAux_0_footer not found, generating
placeholder
[WARNING] Label ifEnd_3 not found, generating placeholder
[WARNING] Label ifEnd_4 not found, generating placeholder
[WARNING] Label block_5_footer not found, generating placeholder
[WARNING] Label block_5_ret not found, generating placeholder
[WARNING] Label block_4_footer not found, generating placeholder
[WARNING] Label block_4_ret not found, generating placeholder
[WARNING] Label ifEnd_5 not found, generating placeholder
[WARNING] Label goldbach_0_footer not found, generating placeholder
You had: 0 lexical errors and 0 syntax errors.
[INFO] Starting Virtual Machine...

[INFO] Program Output:
       3
       11

[INFO] Program terminated correctly.
[INFO] Your program used 239/10000 bytes of memory. (2.39 %)
[INFO] Your program took ~2 ms to complete.
```

Esecuzione del codice intermedio per il congettura di Goldbach.

Il valore i-esimo per il calcolo di goldbach può essere modificato. Si consiglia di non superare il valore di 624 per non incorrere in problemi di “Out of memory”.