Corso di Compilatori e Interpreti
Progetto: SimpLanPlus

Report di progetto

Componenti del gruppo:
Alex Citeroni [1052175]
Davide Talevi [1043409]
Kathleen Anne Velasco [1052173]

Corso di studi: Laurea Magistrale in Informatica
Anno accademico 2021/2022

Indice

Introduzione

Il progetto presentato per il corso di Compilatori ed Interpreti, inerente all'anno accademico 2021/2022, prevede la realizzazione di un compilatore per uno specifico linguaggio chiamato **SimpLanPlus**.
Nella realizzazione si sono susseguiti vari passaggi e si è dovuto anche definire un linguaggio bytecode per eseguire programmi in SimpLanPlus e creare il rispettivo interprete.

Definizione del linguaggio

SimpLanPlus è un semplice linguaggio imperativo, in cui i parametri delle funzioni possono essere passati per variabile, le funzioni non sono annidate (cioè una funzione non può essere definita nel corpo di un'altra), ed ammette la ricorsione, ma non la mutua ricorsione.

Fase di Sviluppo

Una volta installati e configurati tutti i componenti necessari (*ANTLR, Java, IntelliJ*), la fase successiva è stata la realizzazione del sistema. Per la sua complessità, l'implementazione è stata divisa in tre fasi:

1. **Analisi:** rappresenta il primo passo che il compilatore deve eseguire. In questo procedimento, avviene il controllo della correttezza del codice sorgente, ossia il sistema ha il compito di esaminare il file e di accertarsi che le righe di codice di tale documento rispettino la grammatica, definita nel file **SimpLanPlus.g4**, e le specifiche regole del linguaggio.

   In questa fase del programma, vengono eseguite 3 tipologie di verifiche:

   - **Analisi lessicale (lexing):** controlla la correttezza lessicale del codice sotto-esame. Più precisamente, il lexer di *ANTLR* ha la funzione di suddividere ciascun elemento di codice in una sequenza di token e rilevare eventuali elementi minimi errati.
   - **Analisi sintattica (parsing):** l'assenza di errori lessicali consente al sistema di avanzare con l'ispezione sintattica. I token precedentemente identificati vengono raggruppati in

frasi grammaticali, rappresentandoli mediante l'AST (*abstract syntax tree*). In questo tipo di analisi, viene valutato se l'espressione fatta dai token è sintatticamente coerente oppure no.
- **Analisi semantica:** fa riferimento all'ultimo processo di controllo dove viene recuperato l'AST e analizzato se questo segue le regole del linguaggio.

Nel corso dello sviluppo sono state considerate le seguenti specifiche:

- Ogni variabile e funzione deve essere dichiarata
- La dichiarazione di una determinata variabile o funzione, all'interno di uno specifico ambiente, non deve essere ripetuta più di una volta
- La correttezza dei tipi, in particolare, il numero e tipo dei parametri attuali devono essere conformi al numero e tipo dei parametri formali
- Le variabili che vengono richiamate (o usate all'interno del blocco) devono essere inizializzate
- Tutte le variabili che sono state dichiarate devono essere usate.

Nel momento in cui viene identificato un errore, viene stampato il messaggio corrispondente e viene anche arrestata l'esecuzione del programma. Altrimenti, se al termine del controllo non viene rilevato alcun tipo di errore, allora viene generato e stampato l'AST in un file apposito e si procede al successivo passaggio ossia la compilazione.

2. **Generazione del bytecode:** a partire dal codice sorgente (di estensione *.simplanplus*) che è stato verificato, viene generato un codice intermedio e salvato in un file *.asm*. In altre parole quest'ultimo documento contiene il codice bytecode (liste di stringhe di comandi bytecode).
3. **Esecuzione:** in questa fase il codice generato viene eseguito e compilato dalla macchina a pila che effettua i calcoli ed infine memorizza in un apposito registro il valore dell'ultima istruzione ottenuta.

Struttura e organizzazione

Definiamo in questa sezione l'organizzazione dei vari moduli del compilatore, illustrando i loro scopi e le loro logiche.

AST

In questo modulo definiamo i componenti che si occuperanno del matching tra i token analizzati e le regole della grammatica per creare l'albero astratto della sintassi.
Troviamo la cartella dei nodi con le varie definizione per regola e il file `SimpLanPlusVisitorImpl.java` che si occupa della visita e dei nodi e di mantenere le strutture per la costruzione del AST.
Le classi che si occupano dei nodi dell'albero AST sono state divise per sezioni, estrapolate dalla semantica grammaticale:

- **declaration:** nodi riguardanti dichiarazioni di variabili e funzioni
- **exp:** nodi riguardanti le espressioni
- **statement:** nodi riguardanti le istruzioni
- **type:** nodi riguardanti il tipo

INTERPRETER

Definisce la macchina virtuale che ha la funzione di eseguire i calcoli e "interpretare" il codice bytecode generato.

MAINPACKAGE

Contiene la classe **Main** del compilatore, dove viene inizializzato il programma, con tutti i vari passaggi. Nella cartella è presente anche la classe che si occupa della gestione degli errori (*SyntaxErrorDetector*)

PARSER

Contiene tutti i file prodotti da ANTLR in relazione alle grammatiche per il linguaggio (SimpLanPlus) e per il bytecode (SVM).

UTIL

È costituito da diverse classi di utilità, tra cui SimpLanPlusLib che definisce varie funzioni di supporto per strutture dati, funzioni e procedure utilizzate durante le varie fasi di analisi.

Flusso di lavoro del compilatore

L'analisi del codice inizia dalla classe **Main**, in cui vengono richiamati in sequenza il **lexer** e il **parser** generati da **ANTLR** a partire dalla grammatica specificata in `SimpLanPlus.g4`**.** Quest'ultima non ha subito modifiche dalla sua versione originale. I due moduli richiamati controllano quindi il file di input, scritto in linguaggio SimpLanPlus, che contiene il codice da **compilare** ed **eseguire**.

In questa fase, il sistema notificherà di un eventuale **errore lessicale** o **sintattico** gestendo l'eccezione e arrestando la compilazione e fino a quando non saranno corretti tutti gli errori lessicali e sintattici non sarà possibile andare avanti nella compilazione. Gli errori riscontrati vengono salvati in un file di log con estensione .log.

Una volta completata l'analisi lessicale e sintattica in assenza di errori, è possibile passare all'analisi semantica, divisa in tre passaggi:

1. **checkSemantics():** controlla la semantica, se è stato scritto tutto correttamente e non vengono quindi trovati errori si procede, altrimenti vengono stampati gli errori e la compilazione si interrompe.
2. **typeCheck():** controlla se il valore di variabili e funzioni dichiarate sono gli stessi dei tipi assegnati loro. Se non vengono trovati errori si procede, altrimenti vengono stampati gli errori e la compilazione si interrompe.
3. **checkEffects():** controlla se gli effetti sono corretti. Se non vengono trovati errori si procede, altrimenti vengono stampati gli errori e la compilazione si interrompe.

Dopo aver eseguito l'analisi semantica e non aver trovato errori, viene stampato l'albero AST tramite il metodo "**toPrint()**".
L'ultimo passo che viene eseguito riguarda la generazione di un file .asm il quale contiene il codice bytecode, questa funzione viene gestita da "**codeGeneration()**".
Definito il bytecode corrispondente al codice sorgente precedentemente analizzato, il sistema invoca l'interprete per avviare l'esecuzione della

Virtual Machine, che con il supporto di SVM.g4 e altri moduli collegati, al termine del suo processo di esecuzione, restituisce i risultati calcolati.
Eventuali errori, l'albero AST e il risultato dell'interprete vengono salvati nel file di log.

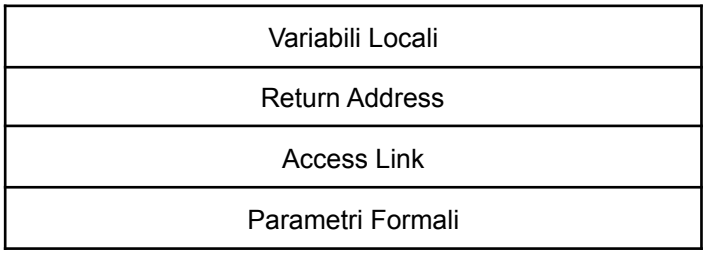
Aliasing

Nel nostro compilatore viene data priorità alle variabili locali rispetto alle variabili globali.
Gestiamo l'aliasing principalmente attraverso la classe CallNode, dove c'è la necessità di confrontare tra loro gli status finali degli alias.
L'approccio utilizzato per risolvere il problema è quello di costruire un dizionario che associ ad un parametro attuale una lista di alias. Tale lista avrà lunghezza minima pari ad 1, nel caso in cui non ci sia aliasing.
Grazie a questo dizionario è dunque possibile eseguire l'operatore par tra gli stati finali di ogni alias associato ad un parametro attuale e poter segnalare eventuali errori.

Activation Record

Un fattore importante che è stato preso in considerazione nel corso dell'implementazione del sistema, riguarda la definizione della struttura dell' Activation Record, nello specifico di una funzione.
Questo ricopre un ruolo fondamentale nel momento in cui termina l'esecuzione del blocco di istruzione di una specifica funzione e riprende il calcolo sospeso prima dell'invocazione.
Tale elemento contiene tutte le informazioni relative alla funzione a cui fa riferimento e anche qualche dato relativo al chiamante.
A livello implementativo, il layout di AR diventa rilevante quando bisogna generare il codice bytecode o di una chiamata di funzione o della sua dichiarazione.

L'Activation Record del sistema è stato strutturato nel seguente modo:



I parametri formali sono gli argomenti della funzione che vengono passati nel momento in cui si ha un invocazione.
Essi sono caratterizzati da offset positivi in quanto sono stati posizionati prima del frame pointer (il quale è stato memorizzato in Access Link).
Per quanto riguarda gli offset usati nella locazione di memoria per le variabili locali, risultano negativi perché, così come si può osservare dall'AR layout, i parametri locali vengono impostati al di sopra di Access Link.

Regole semantiche utilizzate

In questo capitolo vengono elencate tutte le regole semantiche che sono state osservate nel corso dello sviluppo.

$$\frac{}{\Gamma \vdash true: bool}\ [True] \qquad \frac{}{\Gamma \vdash false: bool}\ [False]$$

$$\frac{\Gamma(id) = T}{\Gamma \vdash id: T}\ [Id] \qquad \frac{}{\Gamma \vdash num: int}\ [Num]$$

$$\frac{\Gamma(exp) = bool}{\Gamma \vdash !exp: bool}\ [Not] \qquad \frac{\Gamma(exp) = T}{\Gamma \vdash -exp: T}\ [Neg]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = int = T2 + : intxint -> int}{\Gamma \vdash e1 + e2 : int}\ [Plus]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = int = T2 - : intxint -> int}{\Gamma \vdash e1 - e2 : int}\ [Minus]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = int = T2 / : intxint -> int}{\Gamma \vdash e1 / e2 : int}\ [Div]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = int = T2 * : intxint -> int}{\Gamma \vdash e1 * e2 : int}\ [Mult]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 == : T1xT2 -> bool}{\Gamma \vdash e1 == e2 : bool}\ [Eq]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 != : T1xT2 -> bool}{\Gamma \vdash e1 != e2 : bool}\ [NotEq]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 \&\& : T1xT2 -> bool}{\Gamma \vdash e1 \&\& e2 : bool}\ [And]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 || : T1xT2 -> bool}{\Gamma \vdash e1 || e2 : bool}\ [Or]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 > : T1xT2 -> bool}{\Gamma \vdash e1 > e2 : bool}\ [Major]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 < : T1xT2 -> bool}{\Gamma \vdash e1 < e2 : bool}\ [Minor]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 >= : T1xT2 -> bool}{\Gamma \vdash e1 >= e2 : bool}\ [MajorEq]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2 : T2 \quad T1 = T2 <= : T1xT2 -> bool}{\Gamma \vdash e1 <= e2 : bool}\ [MinorEq]$$

$$\frac{\Gamma \vdash f: T_1 x \dots x T_n -> T \ (\Gamma \vdash e_i: T_i') \ i \in 1..n \quad (T_i = T_i') \ i \in 1..n}{\Gamma \vdash f(e_1, ..., e_n) : T}\ [Call]$$

$$\frac{\Gamma \vdash e : T' \ x \notin dom(top(\Gamma)) \quad T = T'}{\Gamma \vdash T\ x = e\ ; : \Gamma\ [x -> T]}\ [VarD]$$

$$\frac{\Gamma[x_1 -> T_1, ..., x_n -> T_n] \vdash e : T' \ \ T' = T \ \ f \notin dom(top(\Gamma))}{\Gamma \vdash T\ f(T_1\ x_1, .., T_n\ x_n) = e;\ :\ \Gamma[f -> (T_1, ..., T_n) -> T]}\ [FunD]$$

$$\frac{\Gamma(x) \ = \ T \ \ \Gamma \vdash e : T' \quad T = T'}{\Gamma \vdash x = e;\ : void}\ [Asgn]$$

$$\frac{\Gamma \vdash e1:\ T1 \quad \Gamma \vdash e2:\ T2 \quad \Gamma \vdash e3:\ T3 \quad T1 = bool \quad T2 = T3}{\Gamma \vdash if\ (e1)\ e2\ else\ e3 : T2}\ [If] \qquad \frac{\Gamma \vdash exp}{\Gamma \vdash (exp):\ T}\ [Exp]$$

$$\frac{\Gamma \vdash S : void \quad \Gamma \vdash S' : T}{\Gamma \vdash S\ ;\ S' : T}\ [Return] \qquad \frac{\Gamma \vdash e : Int}{\Gamma \vdash print(e);\ : void}\ [Print]$$

$$\frac{\Gamma \cdot [] \vdash D : \Gamma' \quad \Gamma' \vdash S : void}{\Gamma \vdash \{D\ S\} : void}\ [Block] \qquad \frac{\Gamma \vdash d : \Gamma' \quad \Gamma' \vdash D : \Gamma''}{\Gamma \vdash d\ D : \Gamma''}\ [SeqD]$$

$$\frac{\Gamma \vdash s : void \quad \Gamma \vdash S : void}{\Gamma \vdash s\ S : void}\ [SeqS] \qquad \frac{\sum \vdash e : \sum'}{\sum \vdash x = e; : \sum' \rhd [x -> rw]}\ [Asgn - e$$

Qui richiamiamo specifiche regole che richiedono di essere discusse:

- [*VarD*] e [*FunD*]: sono due regole molto simili tra di loro, la prima fa riferimento alla dichiarazione di una variabile, la seconda invece a quella di una funzione. Come si può notare, ad entrambi viene posto il seguente vincolo: la dichiarazione deve essere effettuata una sola volta all'interno di un determinato ambiente, espressa da:

  [VarD] x ∉ dom(top(Γ)) ; [FunD] f ∉ dom(top(Γ))

  dove x e f indicano rispettivamente l'identificatore di una variabile e di una funzione).

  Con i medesimi vincoli, nel caso dovessimo avere una dichiarazione di variabile e di una funzione con lo stesso identico nome, il sistema li identifica come se fossero un unico elemento e non li distingue per tipo o per componente (cioè se una funzione o variabile) di conseguenza restituisce un errore.

  Inoltre, sempre in queste due regole, viene specificato che essi debbano essere inizializzati.
- [*If*]: da sottolineare il fatto che il tipo del ramo then deve coincidere con quello del ramo else (nel caso in cui ci fosse). Questa regola può essere osservata nell'esercizio 2 già analizzata nel capitolo dell'analisi semantica.
- [*Return*]: nel caso in cui all'interno di un blocco ci fosse un return, quest'ultimo nodo deve essere posto come ultimo statement.
- [*Print*]: l'espressione *e,* presente dentro lo statement *print,* è di tipo int anche se l'utente vuole stampare un valore bool (allora il

sistema esegue una conversione: 0 se è false e 1 se è true). In generale lo statement *print* è di tipo void.

- [*SeqS*]: la regola semantica generale quando si ha una sequenza di statement all'interno di un blocco. Bisogna tenere in considerazione il caso in cui si ha uno statement return, allora si riprende la regola di [*Return*].
- [*Asgn-e*]: la regola di inferenza con effetti. Quando viene assegnato un valore ad una determinata variabile, il suo stato diventa READWRITE. Facendo in questo modo, il sistema non restituirà un errore di "non inizializzata" o di "non utilizzata".

Analisi Lessicale

Successivamente allo sviluppo dei moduli per l'analisi lessicale abbiamo trascritto due file di test per verificare il corretto funzionamento del codice. Per l'analisi lessicale è stato analizzato il seguente codice:

```
1. {
2.   //dichiarazione variabili
3.   int nome = (1 - 19);
4.   bool a1;
5.   a1 = true;
6.   // controllo
7.   if(flag == a1){
8.      print a1;
9.      return true;
10.   }
11.   else{
12.     @false;
13.   }
14.   \ /* @FINE*/
15.   {
16.    if (a1]) print true;
17.   }
18.  }
```

Tutti gli errori che vengono rilevati dal programma sono scritti in un file con estensione *nomeFile.log*.
Seguendo la grammatica che abbiamo utilizzato, in questo codice sono presenti tre errori:

```
Error in line 7, character 56: token recognition error at: '@'
Error in line 8, character 3: token recognition error at: '\'
Error in line 10, character 56: token recognition error at:
']'
```

Nella grammatica SimpLanPlus alcuni determinati caratteri speciali (quali: **@, \,]**) non sono riconosciuti e di conseguenza generano un errore sintattico.
Non viene generato alcun tipo di errore se questi caratteri speciali vengono inseriti dentro un commento o un blocco di commenti in quanto

questi non vengono valutati come token dal parser non facendo parte del programma stesso.

L'esercizio n.2 richiede che l'analizzatore fosse in grado di rilevare se variabili e/o funzioni utilizzate fossero state precedentemente dichiarate nel programma, generando errore in caso contrario. Abbiamo deciso di inserire nel file di log anche questi errori.

Per la valutazione abbiamo scritto questo frammento di codice:

```
19.  {
20.    int eccolo;
21.    int funzione(int a){
22.       int pippo = 1a;
23.       in b = 10
24.    }
25.  }
```

Ecco i tre errori presenti nel codice rilevati dal nostro programma:

1. Il primo errore che è stato rilevato è:
   `Error in line 4, character 20 :extraneous input 'a' expecting ';'`
   In questo caso il programma ha rilevato, a destra dell'uguale, un'espressione non accettabile. L'exp non può essere un valore alfanumerico composto da DIGIT e CHAR (a…z| A…Z). Invece, se scrivessimo "int pippo = a1;" per il programma, questo risulta "corretto".

2. Il secondo errore che è stato rilevato è:
   `Error in line 5, character 10 :no viable alternative at input 'inb'`
   In questo caso il programma non conosce un tipo di dato come "in" e quindi non riesce a capire di che tipo è la variabile "b".

3. Il terzo errore che è stato rilevato è:
   `Error in line 6, character 4 :missing ';' at '}'`
   Qui l'errore sta nel fatto che manca il carattere ";" alla fine della dichiarazione della variabile.

Analisi semantica

Abbiamo verificato i seguenti codici:

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

L’analisi semantica di questo codice ha riportato come risultato la seguente dichiarazione: “L'Id 'b' non è stato inizializzato!”.
L’errore viene causato dalla riga 8 del codice, ossia quando viene assegnata una determinata variabile ad un’altra che non risulta inizializzata. Difatti anche se c fosse uguale ad un valore maggiore di 1, appare lo stesso messaggio di errore. Generalmente, una variabile viene identificata “non inizializzata” quando essa viene assegnata avendo come stato “DECLARED”.

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

In questo caso abbiamo avuto come risultato 2 errori:

1. L'Id 'a' non è stato inizializzato!
2. L'Id 'b' non è stato inizializzato!

Abbiamo questi errori perchè richiamiamo la funzione “h” dove vengono utilizzate le variabili “a” e ”b” a cui non è stato assegnato nessun valore.

Se inseriamo un valore o una variabile dopo il return e prima del “;”, ad esempio “1” abbiamo questo errore: “Tipi incompatibili in altri rami”. Questo perché nel then il valore di ritorno è un intero, mentre nell’else il valore di ritorno è di tipo Void. Invece, riprendendo il codice sorgente originale, l’errore “Tipi incompatibili in altri rami” non appare poichè entrambi i rami sono dello stesso tipo di ritorno, ovvero void.

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

In questo esempio è presente alla riga 8 la fine del corpo del codice, quindi viene eseguito riportando questi 3 errori:

```
Errore: la variabile 'a' è stata dichiarata, ma non viene utilizzata
Errore: la variabile 'b' è stata dichiarata, ma non viene utilizzata
Errore: la variabile 'c' è stata dichiarata, ma non viene utilizzata
```

Questi messaggi di errore appaiono in quanto tutte e tre le variabili, al termine dell’esecuzione, sono caratterizzate da stato DECLARED. E siccome uno dei requisiti del progetto richiede di rilevare variabili non usate, il sistema restituisce tale messaggio di errore nel momento in cui una specifica variabile al termine del programma ha mantenuto uno stato DECLARED.
Se togliamo la “}” presente nella riga 8, abbiamo invece 3 errori, dati dal mancato assegnamento di valore alle variabili “a”, “b” e “c”:

1. L'Id 'a' non è stato inizializzato!
2. L'Id 'b' non è stato inizializzato!
3. L'Id 'c' non è stato inizializzato!

Evento simile agli esercizi già spiegati sopra.

Albero AST

Codice utilizzato:

```
1. {
2.    void f(int m, int n) {
3.       if (m > n) {
4.           print(m + n);
5.       }
6.       else {
7.           int x = 1;
8.           f(m + 1, n + 1);
9.       }
10.  }
11.  f(5,4);
12.  f(4,5);
13.  }
```

Albero prodotto:

```
Block
 DecFun: 'f' -> VoidType
  Arg: 'm' -> IntType
  Arg: 'n' -> IntType
   Block
     ITE
      BinExp[>]
        DerExpNode: 'm' -> at nesting level 1 with offset 1
        DerExpNode: 'n' -> at nesting level 1 with offset 2
      Block
       PrintNode
         BaseExpNode
            BinExp[+]
              DerExpNode: 'm' -> at nesting level 1 with offset 1
              DerExpNode: 'n' -> at nesting level 1 with offset 2
      Block
        DecVar: 'x' -> IntType
         ValExpNode: 1
        Call: 'f' -> at nesting level 0 with offset 0
         BinExp[+]
           DerExpNode: 'm' -> at nesting level 1 with offset 1
           ValExpNode: 1
         BinExp[+]
           DerExpNode: 'n' -> at nesting level 1 with offset 2
           ValExpNode: 1
 Call: 'f' -> at nesting level 0 with offset 0
  ValExpNode: 5
  ValExpNode: 4
 Call: 'f' -> at nesting level 0 with offset 0
  ValExpNode: 4
  ValExpNode: 5
```

Esecuzione dell’interprete

Sono stati eseguiti i seguenti codici:

```
1.  {
2.   int x = 1;
3.   void f(int n){
4.      if (n == 0) {
5.          print(x);
6.      } else {
7.          x = x * n;
8.          f(n - 1);
9.      }
10.    }
11.    f(10);
12.  }
```

Il risultato dell’esecuzione di questo codice non riporta nessun errore e il valore di ritorno risulta “0”.

```
1.  {
2.   int u = 1;
3.   void f(var int x, int n) {
4.      if (n == 0) {
5.          print(x);
6.      }
7.      else {
8.          int y = x * n;
9.          f(y, n - 1);
10.       }
11.    }
12.    f(u, 6);
13.  }
```

Nemmeno il risultato dell’esecuzione di questo codice riporta un errore, ma invece ritorna comunque un valore intero, “720”.

```
1.  {
2.   void f(int m, int n) {
3.      if (m > n) {
4.          print(m + n);
5.      }
6.      else {
7.          int x = 1;
```

```
8.            f(m + 1, n + 1);
9.        }
10.      }
11.      f(5,4);
12.      f(4,5);
13.  }
```

Abbiamo 2 risultati in questa esecuzione:

1. Il primo riguarda la chiamata di funzione “f(5, 4);”: in questo caso abbiamo come risultato il valore intero “9”.
2. Il secondo risultato riguarda la chiamata di funzione “f(4, 5);”, qui troviamo un’errore: “Errore: memoria insufficiente”. Succede questo perchè il codice produce un loop infinito, dato che m=4 e n=5, quindi non soddisfa i requisiti dell’if (all’interno della funzione void) ed entra nell’else, dove viene richiamata la stessa funzione ma con i valori aumentati di 1; questa operazione viene ripetuta all’infinito perché i requisiti dell’if non vengono mai soddisfatti.