Compilatore per SimpLanPlus

Simone Branchetti - 0001038801
simone.branchetti2@studio.unibo.it

Complementi di Linguaggi di Programmazione

Abstract

SimpLanPlus è un linguaggio imperativo dove i tipi possibili sono int, bool e void, è possibile dichiarare una variabile senza inizializzarla, ammette funzioni ricorsive (ma non mutualmente ricorsive), un programma o il corpo di una funzione può essere stm oppure dec stm e sono ammessi corpi di funzioni del tipo { stm ; exp }. In tal caso la funzione, dopo aver valutato stm, ritorna il valore di exp.
Lo scopo del progetto è quello di creare un compilatore per SimpLanPlus attraverso la risoluzione di 4 esercizi. Il resto del report è strutturato in 5 capitoli, uno per esercizio preceduti da un capitolo che spiega la struttura del progetto.

1 Struttura del progetto

Il progetto è stato realizzato con IntelliJ IDEA e l'estensione ANTLR 4.12.0 per generare i file delle grammatiche SimpLan e SVM. Nella cartella principale si trovano questa relazione e 4 sottocartelle:

- **gen**: contiene i file generati da ANTLR per la grammatica SimplanPlus.g4 e per la grammatica SVM.g4.
- **lib**: contiene il file .jar di ANTLR versione 4.12.0.
- **out**: output per la compilazione e il programma in sé.
- **src**: cartella principale del progetto, contiene quattro pacchetti: *ast* dove si trovano le implementazioni dei nodi dell'albero sintattico, *interpreter* contiene l'implementazione dell'interprete di SimpLanPlus, *sem_an* dove troviamo l'implementazione della symbol table e *simpLanPlusLib* che contiene funzioni utili per il resto del progetto.

Prima di parlare degli esercizi assegnati è importante illustrare alcune scelte progettuali per quanto riguarda l'albero sintattico e i nodi che lo compongono.

1.1 Albero Sintattico

La grammatica originale è stata ampliata usando degli identificatori e delle etichette in modo da poter più facilmente capire il contesto della produzione. Ad esempio in una produzione come
`exp: e1=exp (op='*' | op='/') e2=exp #mulDivExp`
si possono notare le etichette `e1` e `e2` e l'identificatore `#mulDivExp`. Tutte queste modifiche servono per implementare la classe Visitor che si occupa di creare l'albero sintattico vero e proprio attraverso chiamate ricorsive ai componenti di un programma preso in analisi. Quando un programma viene analizzato il primo nodo da cui parte la visita è prog, ogni elemento genera un nodo corrispondente al suo tipo e ricorsivamente espande la creazione dell'albero ai suoi figli sfruttando ancora una volta la funzione visit di ANTLR.

1.2 Nodi

Tutti i nodi dell'albero sintattico implementano l'intefaccia Node che contiene qualche definizione di metodi che vengono poi implementati in modo diverso in ogni nodo. I quattro metodi contenuti nell'interfacia sono:

- **checkSemantics**: controlla se il nodo corrente presenta errori semantici. In caso di errori, essi vengono aggiunti ad un ArrayList che verrà infine ritornato.
- **typeCheck**: type checking del nodo corrente mediante regole di inferenza.
- **printNode**: semplice stampa delle informazioni del nodo.
- **codeGeneration**: genera il codice intermedio per il nodo.

In totale ci sono 17 tipi di nodo che implementano la stessa intefaccia; nel seguente elenco sarà data una breve descrizione per ognuno di questi, in ordine alfabetico:

- **ArgNode**: nodo per gli argomenti di una funzione. Contiene l'ID dell'argomento e l'espressione che gli si sta assegnando.
- **AsgNode**: nodo per lo statement di assegnamento. Contiene l'ID dell'argomento e l'espressione che gli si sta assegnando.
- **BoolNode**: contiene 1 se il valore booleano è vero oppure 0 altrimenti.
- **CfrExpNode**: coniene due nodi per i due operatori e una stringa per il tipo di operazione. Questo nodo si occupa di tutte le operazioni di confronto ($>$, $\geq$, $\leq$, $\leq$, $==$).
- **DecFunctionNode**: questo nodo si occupa delle dichiarazioni delle funzioni e contiene il TypeNode per accedere alle informazioni sul tipo della funzione, il suo nome e diversi ArrayList di Node per i parametri, dichiarazioni e statement della funzione più l'eventuale espressione di ritorno.

- **DecNode**: viene usato per le dichiarazioni di variabili e contiene TypeNode che a sua volta contiene id e tipo della variabile dichiarata.
- **FunctionCallNode**: usato per le chiamate di funzioni, contiene l'id della funzione chiamata e ArrayList per i parametri e il corpo della funzione, in aggiunta all'eventuale return.
- **IdNode**: contiene l'id come stringa.
- **IfExpNode**: nodo per l'if quando prodotto dalla produzione exp. Contiene il Node per l'espressione di controllo e gli statement per il ramo then e il ramo else in due ArrayLists. Può contenere le eventuali espressioni di ritorno dei rami.
- **IfStmNode**: nodo per l'if quando prodotto dalla produzione stm. Uguale al precedente tranne per le espressioni di ritorno che qui mancano siccome questa produzione fa parte degli stm.
- **IntNode**: contiene il valore del nodo sotto forma di numero intero
- **LogicalExpNode**: gestisce le operazioni logiche come && e || quindi contiene due nodi per gli operandi e una stinga che definisce l'operatore.
- **MulDivNode**: gestisce le operazioni di moltiplicazione e divisione quindi contiene due nodi per gli operandi e una stinga che definisce l'operatore.
- **NotExpNode**: gestisce l'operatore ! e contiene il nodo per l'operando.
- **PlusMinusNode**: gestisce le operazioni di addizione e sottrazione quindi contiene due nodi per gli operandi e una stinga che definisce l'operatore.
- **ProgramNode**: se il programma è di tipo singeExp allora contiene quella espressione, altrimenti contiene la lista di dichiarazioni, la lista di statement e l'eventuale espressione alla fine.
- **TypeNode**: identifica il tipo di dato di un nodo, contiene una stringa con il nome del tipo identificato e un ArrayList con i tipi dei parametri nel caso il nodo sia una funzione.

1.3 Symbol Table

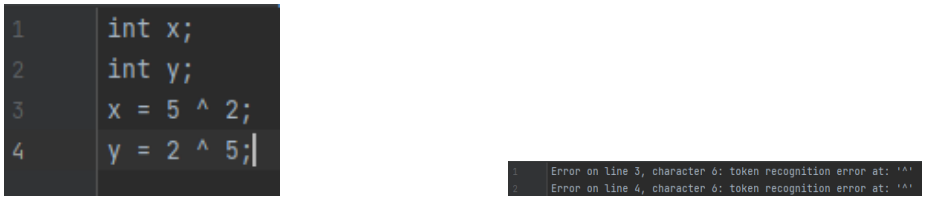
Dopo aver costruito l'albero sintattico e averne terminato con successo l'analisi, il programma costruisce la tabella dei simboli per il codice inserito attraverso un oggetto di tipo Environment che, partendo da vuoto, viene passato come parametro alla funzione `checkSemantics` che popola la tabella in base a quale nodo incontra e gestisce gli offset. Al termine di questa operazione ricorsiva, la symbol table si presenta come una pila di tabelle di hash dove ognuna di queste è formata da una coppia {chiave, valore} dove la prima è l'id della variabile e la seconda è un oggetto di tipo SymbolTableEntry composto da una label, un tipo, un offset e uno status, che serve a verificare se questa entry è dichiarata

o inizializzata. Quando il codice da analizzare entra in un nuovo ambiente si aggiunge un elemento alla pila di tabelle e quando esce da un ambiente fa il pop del primo elemento della pila, rendendo più semplice la gestione del livello di annidamento degli ambienti, che corrisponde alla profondità nella pila. Un caso particolare che vale la pena esplorare è quello del costrutto `if then else` dove, per assicurarsi che l'ambiente in cui vengono valutati entrambi i rami sia lo stesso è necesario fare una deep copy dell'ambiente di partenza prima di entrare nei rami. Questo comporta che, se un ramo inizializza una variabile che non viene inizializzata nell'altro, si debba sollevare un'errore perchè a livello statico non è possibile sapere quale ambiente verrà effettivamente ritornato dopo l'esecuzione dell'if. Alcuni esercizi richiedono che questo comportamento venga meno: in tal caso si può settare la variabile `skipTypeCheck` nel file Main a `true` per saltare il processo di type checking.

2 Esercizio 1

`L'analizzatore lessicale deve ritornare la lista degli errori lessicali in un file di output. Il report deve contenere la discussione di tre esempi e degli errori segnalati`

Allo scopo di risolvere questo esercizio è stato esteso il BaseErrorListener di ANTLR nella classe errorCheck che viene settato come listener di errori per il lexer e il parser di SimpLanPlus cosí da gestire gli errori lessicali e sintattici. Questi errori vengono immagazzinati in una lista mentre viene compilato il file e poi vengono salvati nel file "errors.txt" contenuto nella cartella out del progetto. Di seguito vengono mostrati tre esempi di errori lessicali e i relativi messaggi stampati su file.



Codice Errore

Figura 1: Primo esempio di errore nell'analisi lessicale. Il token "∧" non appartiene alla grammatica di SimpLanPlus.

3 Esercizio 2

`Sviluppare la tabella dei simboli del programma. Il codice sviluppato deve controllare`

- `identificatori/funzioni non dichiarati,`

Codice Errore

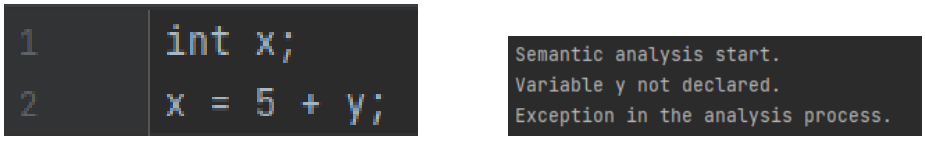
Figura 2: Secondo esempio di errore nell'analisi lessicale. Il token "|" non appartiene alla grammatica di SimpLanPlus

Codice Errore

Figura 3: Terzo esempio di errore nell'analisi lessicale. Il token "_" non appartiene alla grammatica di SimpLanPlus

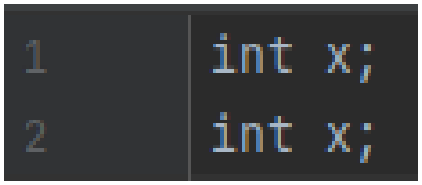
- `identificatori/funzioni dichiarate più volte nello stesso ambiente.`

Per dimostrare la capacità del compilatore di identificare correttamente identificatori o funzioni non dichiarate o dichiarate più volte si fa affidamento agli esempi seguenti:



Codice Errore

Figura 4: Esempio di errore per variabile non dichiarata.

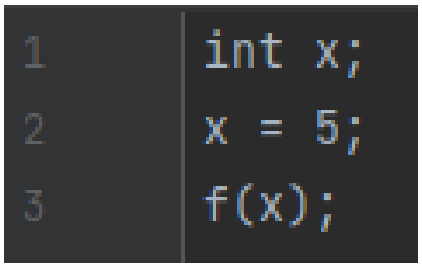


Codice

```
Semantic analysis start.
Variable x already declared in this scope.
Exception in the analysis process.
```

Errore

Figura 5: Esempio di errore per variabile dichiarata più volte.

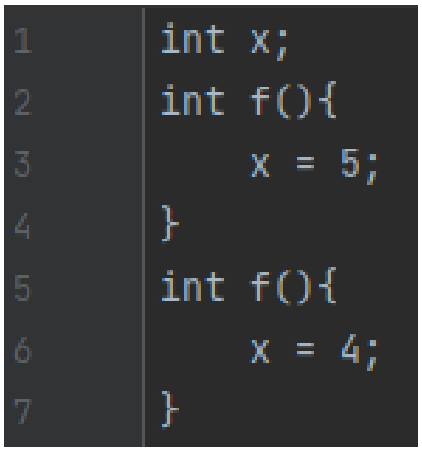


Codice

```
Semantic analysis start.
Function f was not declared.
Exception in the analysis process.
```

Errore

Figura 6: Esempio di errore per funzione non dichiarata.



Codice

```
Semantic analysis start.
Function name f already used in this scope.
Exception in the analysis process.
```

Errore

Figura 7: Esempio di errore per funzione dichiarata più volte.

4 Esercizio 3

`Sviluppare un'analisi semantica che verifica`

- `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 assumendo che le funzioni non accedano mai a variabili globali.`

L'analisi semantica è divisa in sottosezioni, ognuna che esplora una parte della grammatica più una sezione riguardante gli esempi proposti da risolvere e le loro . Il focus principale di questo esercizio è l'uso di variabili non inizializzate che, nel progetto, viene risolto modificando come è costruito un ambiente:

$$\Gamma = [ID \to (T, n, init), ...]$$

In ogni ambiente, un ID è collegato non solo al suo tipo T e all'offset in memoria n ma anche a una variabile booleana init che rappresenta lo stato della variabile: se è falsa la variabile è stata dichiarata ma non ha ancora nessun valore assegnato mentre se è verala variabile è inizializzata correttamente ed è printa all'uso. Questo verrà usato nelle regole di dichiarazione dove verrà settata a false init per la variabile dichiarata, nella regola di assegnamento dove verrà invece settata a true e infine nell'espressione idExp dove una premessa sarà appunto che init sia true, in modo da assicurarsi che la variabile sia stata usata in almeno un assegnamento.

4.1 Giudizi

$$\Gamma, n \vdash \texttt{exp} : T \qquad \Gamma, n \vdash \texttt{stm} : \Gamma', n$$
$$\Gamma, n \vdash \texttt{dec} : \Gamma', n' \qquad \emptyset, 0 \vdash \texttt{prog} : T$$
$$\Gamma, n \vdash \texttt{body} : T$$

4.2 Programma

$$\frac{[], 0 \vdash \texttt{e} : T}{\emptyset, 0 \vdash \texttt{e} : T}\ \text{singleExp}$$

$$\frac{[], 0 \vdash d : \Gamma, n \quad \Gamma, n \vdash D : \Gamma', n' \quad \Gamma', n' \vdash s : \Gamma'', n' \quad \Gamma'', n' \vdash S : \Gamma''', n' \quad \Gamma''', n' \vdash e : T}{\emptyset \vdash dD\ sS\ e : T}\ \text{multExp}$$

4.3 Body

$$\frac{\Gamma, n \vdash d : \Gamma', n' \quad \Gamma', n' \vdash D : \Gamma'', n'' \quad \Gamma'', n'' \vdash s : \Gamma''', n'' \quad \Gamma''', n'' \vdash S : \Gamma'''', n'' \quad \Gamma'''', n'' \vdash e : T}{\Gamma, n \vdash dD\ sS\ e : T}\ \text{body}$$

4.4 Dichiarazioni

$$\frac{ID \notin dom(top(\Gamma))}{\Gamma, n \vdash T\ ID : \Gamma[ID \to (T, n, init = false)], n'}\ \text{varDec}$$

$$\frac{f \notin top(dom(\Gamma)) \quad (\Gamma, n \vdash ID_i : T_i')^{i \in 1,...,m} \quad (T_i = T_i')^{i \in 1,...,m} \quad \Gamma', n \vdash body : T}{\Gamma, n \vdash Tf(T_1\ ID_1, ..., T_m\ ID_m)\{body\} : \Gamma[f \to T_1 \times ...T_m \to T], n}\ \text{funDec}$$

4.5 Statements

$$\frac{\Gamma(ID) = T, init \quad \Gamma \vdash e : T}{\Gamma, n \vdash ID = e : \Gamma[ID \to (T, m, init = true)], n}\ \text{asgStm}$$

$$\frac{\Gamma, n \vdash f : T_1 \times ... \times T_n \to T \quad (\Gamma, n \vdash e_i : T_i')^{i \in 1,...,m} \quad (T_i = T_i')^{i \in 1,...,m}}{\Gamma, n \vdash f(e_1, ..., e_m) : \Gamma, n}\ \text{funCallStm}$$

$$\frac{\Gamma, n \vdash e : bool \quad \Gamma, n \vdash thenStm : \Gamma', n \quad \Gamma, n \vdash elseStm : \Gamma'', n \quad \Gamma''' = \Gamma' \vee \Gamma''}{\Gamma, n \vdash if\ then\ thenStm\ else\ elseStm : \Gamma''', n}\ \text{ifStm}$$

$$\frac{\Gamma, n \vdash s : \Gamma', n \quad \Gamma', n \vdash S : \Gamma'', n}{\Gamma, n \vdash sS : \Gamma'', n}\ \text{thenElseStmBranch}$$

4.6 Espressioni

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool \quad \&\& : bool \times bool \to bool}{\Gamma \vdash e_1\ \&\&\ e_2 : bool}\ \text{LogicalExpAND}$$

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool \quad \| : bool \times bool \to bool}{\Gamma \vdash e_1\ \|\ e_2 : bool}\ \text{LogicalExpOR}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad + : int \times int \to int}{\Gamma \vdash e_1 + e_2 : int}\ \text{plusMinusExpPlus}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad - : int \times int \to int}{\Gamma \vdash e_1 - e_2 : int}\ \text{plusMinusExpMinus}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad * : int \times int \to int}{\Gamma \vdash e_1 * e_2 : int}\ \text{multDivExpMult}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad / : int \times int \to int}{\Gamma \vdash e_1/e_2 : int}\ \text{multDivExpDiv}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad >: int \times int \to bool}{\Gamma \vdash e_1 > e_2 : bool} \text{ cfrExpGT}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad <: int \times int \to bool}{\Gamma \vdash e_1 < e_2 : bool} \text{ cfrExpLT}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad >=: int \times int \to bool}{\Gamma \vdash e_1 >= e_2 : bool} \text{ cfrExpGTE}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad <=: int \times int \to bool}{\Gamma \vdash e_1 <= e_2 : bool} \text{ cfrExpLTE}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad ==: int \times int \to bool}{\Gamma \vdash e_1 == e_2 : bool} \text{ cfrExpEquals}$$

$$\frac{\Gamma \vdash e : bool \quad ! : bool \to bool}{\Gamma \vdash \ ! \ e : bool} \text{ notIdExp}$$

$$\frac{\Gamma(ID) = (T, init) \quad init = true}{\Gamma \vdash ID : T} \text{ idExp}$$

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

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash (e) : T} \text{ bracketExp}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash thenBranch : T \quad \Gamma \vdash elseBranch : T' \quad T = T'}{\Gamma \vdash if(e) \ then \ thenBranch \ else \ elseBranch : T} \text{ ifExp}$$

$$\frac{\Gamma, n \vdash s : \Gamma', n \quad \Gamma', n \vdash S : \Gamma'', n \quad \Gamma'', n \vdash e : T}{\Gamma, n \vdash sS \ e : T} \text{ thenElseExpBranch}$$

4.7 Tipi

$$\frac{}{\Gamma \vdash number : int} \text{ num}$$

$$\frac{}{\Gamma \vdash true : bool} \text{ boolTrue}$$

$$\frac{}{\Gamma \vdash false : bool} \text{ boolFalse}$$

4.8 Discussione Esempi

La tabella seguente mostra gli esempi da discutere e gli errori rilevati dal programma.

Esempi da discutere	Errori

`int a; int b; int c ; c = 2 ;`
`if (c > 1) { b = c ; }`
`else { a = b ; }`
	TypeChecking Error:
	variable b was declared but not initialized.

`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) ;`
	TypeChecking Error:
	variable x was declared but not initialized.

`void h(int n){`
`int x ; int y ;`
`if (n==0){ x = n+1 ;}`
`else { h(n-1) ; x = n ; y = x ;}`
`}`
`h(5) ;`
	Semantic analysis completed.
	TypeChecking started
	TypeChecking completed.

`int a;`
`void h(int n){`
`int x ; int y ;`
`if (n==0){ x = n+1 ;}`
`else { h(n-1) ; y = x ;}`
`}`
`h(5) ;`
	TypeChecking Error:
	variable x was declared but not initialized.

Tabella 1: Esempi di analisi semantica: in tutti i casi con errori il ramo else contiene una variabile dichiarata ma non inizializzata usata come right side expression in degli assegnamenti, portando appunto a un errore di type checking.

5 Esercizio 4

`Estendendo l'interprete di SimpLan, implementare l'interprete di SimpLanPlus.`
Ogni nodo dell'albero sintattico ha una funzione `codeGen` che genera il codice per l'interprete relativo a quel nodo. Percorrendo l'albero, ogni nodo genera la sua porzione di codice che verrà poi eseguita dalla classe ExecuteVm. La grammatica dell'interprete aggiornata è nel file SVM.g4 e contiene, oltre ai comandi già presenti tre comandi che servono per il controllo del flusso nel caso di tre operazioni di disuguaglianza ($>, <, \geq$) che permettono di coprire tutti i casi. I registri utilizzati sono un Frame Pointer (FP), uno Stack Pointer (SP), un Access Link (AL), un registro per l'indirizzo di return delle funzioni (RA), un Instruction Pointer (IP), un accumulatore (A0) e due registri temporanei (T1 e T2). Il tutto viene eseguito su una memoria composta da 1000 celle, in modo da non sovraccaricare troppo il sistema. I comandi sono quindi:

- 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 (NUMBER — LABEL)
- pushr REG
- pop
- popr REG
- b LABEL
- beq REG REG LABEL
- bleq REG REG LABEL

- jsub LABEL
- rsub REG
- l=LABEL ':'
- halt
- bgt REG REG LABEL
- blt REG REG LABEL
- bgte REG REG LABEL

Questo interprete permette di eseguire il codice scritto nel file input.txt e di restituire il risultato dell'espressione alla fine di quest'ultimo. I seguenti esempi verificano i codici della traccia:

Esempi da discutere	Output

`int x ;`
`void f(int n){`
`  if (n == 0) { n = 0 ;}`
`  else { x = x * n ; f(n-1) ;}`
`}`
`x = 1 ;`
`f(10)`
	TypeChecking skipped.
	Code generation started.
	Code successfully generated.
	Result : 0

`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)`
	TypeChecking skipped.
	Code generation started.
	Code successfully generated.
	Result : 720

`int u ;`
`void f (int m, int n){`
`  int x;`
`  if (m>n) { u = m+n ;}`
`  else { x = 1 ; f(m+1,n+1) ; }`
`}`
`f(5,4); u`
	TypeChecking skipped.
	Code generation started.
	Code successfully generated.
	Result : 9

`int u ;`
`void f (int m, int n){`
`  int x;`
`  if (m>n) { u = m+n ;}`
`  else { x = 1 ; f(m+1,n+1) ; }`
`}`
`f(4,5); u`
	Index -1 out of bounds for length 1000.
	Questo significa che l'interprete ha finito
	lo spazio in memoria a causa
	di un ciclo infinito nel codice.

Tabella 2: Esempi del funzionamento dell'interprete.

Gli ultimi due esercizi presentavano una dichiarazione di variabile dentro il ramo else dell'if che è stata tolta e rimessa all'inizio del corpo della funzione f in modo da far proseguire l'esecuzione del programma senza l'errore di una dichiarazione fuori posto. Nella tabella è stata riportata la forma corretta in modo da non allungare troppo questo report.