Corso di Complementi di Linguaggi di Programmazione

Relazione Progetto d'Esame

# Compilatore SimpLanPlus

Anno Accademico 2022/2023

Autori:
Raciti Gabriele - Mat: 0001102147
(raciti.gabriele2@studio.unibo.it)
Tamai Leonardo - Mat: 0001098711
(tamai.leonardo@studio.unibo.it)

# 0. INDICE

# 1. INTRODUZIONE

SimpLanPlus è un linguaggio imperativo in cui è possibile dichiarare variabili di tipo booleano ed intero e funzioni di tipo booleano, intero e void. Le funzioni possono essere ricorsive ma non mutuamente ricorsive ed il loro corpo può contenere dichiarazioni iniziali, altri statement e un'espressione in caso dovesse ritornare un valore.

Oltre al codice sorgente del compilatore/interprete, codici di esempio e alla stessa relazione, all'interno del progetto consegnato è presente un file .jar, che consente di provare rapidamente il progetto. È sufficiente aprire un terminale nella cartella principale del progetto e inserire il comando "java -jar .\SimpLanPlus.jar", per eseguire il progetto.
In più, è possibile personalizzare l'esecuzione del progetto con il comando "java -jar SimPlanPlus.jar [1/0] [1/0]", dove il primo numero indica se deve (1) o non deve (0) essere eseguita la Type Check, mentre il secondo numero indica se deve (1) o non deve (0) essere eseguita la generazione di codice. Questa personalizzazione è stata inserita per permettere di testare al meglio il progetto.

## 1.1. AMBIENTE DI SVILUPPO

Il gruppo ha deciso di utilizzare come IDE Eclipse ed utilizzare GitHub per coordinare al meglio l'attività lavorativa. Una volta installata l'estensione di ANTLR il progetto è stato subito suddiviso nei seguenti package:

- **Ast**: contiene le implementazioni di tutti i nodi dell'albero sintattico.
- **Evaluator**: contiene delle classi per gestire le etichette generate dal codice e delle altre per la gestione della generazione di codice.
- **MainPackage**: contiene la classe Main da cui avviare il progetto.
- **Parser**: contiene le classi delle grammatiche e tutte quelle generate automaticamente da ANTLR sulla base di esse.
- **SemanticAnalysis**: contiene le classi necessarie all'implementazione della Symbol Table.
- **InputOutput**: contiene il file "input.txt" da cui viene preso il codice sorgente da analizzare, il file "syntax_errors.txt" in cui verranno riportati tutti gli errori sintattici/lessicali ed il file "Esercizi.txt" in cui sono contenuti i codici di esempio analizzati in questo documento.

Al fine di rispettare i requisiti del linguaggio, la grammatica è stata modificata introducendo gli elementi "BlockSeqStm" e "BlockSeqStmExp" che rappresentano rispettivamente il contenuto dei rami THEN ed ELSE dei costrutti IF "ifStm" e "ifExp".
Non sono state necessarie ulteriori modifiche per impedire la mutua ricorsione, in quanto già priva di un costrutto in grado di definire prototipi. Non verrà quindi svolta una visita preliminare dall'analizzatore semantico per raccogliere gli id di tutte le funzioni.

## 1.2. CONTENUTO DELLA RELAZIONE

La presente relazione è suddivisa in capitoli in cui vengono descritte le diverse sezioni del progetto, ovvero: l’analisi lessicale, l’implementazione della Symbol Table, l'analisi semantica ed infine l’interprete. In ogni capitolo verranno discusse le scelte implementative adottate dal gruppo insieme ad alcuni esempi del funzionamento di ciò che è stato implementato.

# 2. ESERCIZIO 1 - ANALIZZATORE LESSICALE

Una volta eseguite le modifiche alla grammatica iniziale descritte nel capitolo precedente, è stato utilizzato ANTLR per auto-generare le classi di parser e lexer. Successivamente è stata predisposta la lettura del file di input “input.txt” per estrarne i token ed effettuare l’analisi lessicale.
Sono stati poi definiti tutti gli elementi della grammatica sotto forma di nodi, in modo da avere ognuno di essi come classe che implementa l’interfaccia comune “Node”. Ultimato il lavoro sui nodi, ci si è soffermati sull’implementare il visitor per SimpLanPlus: questa classe, contenuta nel package “ast”, contiene gli override dei metodi predefiniti della classe generica generata da ANTLR. Questi vengono chiamati quando si esegue la funzione “visit()” su un elemento trovato dal parser, e consente di costruire l’albero della sintassi.
Si è anche implementata la funzione “toPrint” di ogni nodo, in modo da ottenere una rappresentazione grafica dell’albero prodotto.
Si è inoltre implementata una visita iniziale per ogni token e, utilizzando il carattere della grammatica “ERR” (che raccoglie al suo interno tutti i simboli accettati dalla grammatica), si è implementata un’analisi sintattica, che esegue la “cattura” di ogni carattere non riconosciuto dalla grammatica e li aggiunge a un ArrayList di stringhe, per poi inserirle all’interno di un file “syntax_errors.txt” che fornirà informazioni riguardo il carattere non riconosciuto e la posizione nel codice in cui si trova

## 2.1. ESEMPI

Di seguito una tabella che contiene nella prima colonna dei possibili esempi di codice erroneo e nella seconda colonna il rispettivo errore che viene visualizzato.

| **Input Erroneo** | **Errore in Output** |
|---|---|
| int a_;<br>int b;<br>a_ = 2;<br>b = 3; | [!] Error in line 1 character 6, symbol "_" not recognized.<br>[!] Error in line 3 character 2, symbol "_" not recognized. |
| int a;<br>int f(int x, int y){<br>  x<br>}<br>b = false; | You had 1 errors:<br>  Var id 'b' not declared |
| int a;<br>void f(int x){<br>  int x;<br>  x = 3;<br>} | You had 1 errors:<br>  Var id 'x' already declared |

Nel primo esempio viene mostrato il comportamento del compilatore in presenza di un simbolo (“_”) non presente all’interno della grammatica (errore lessicale). Il messaggio d’errore, infatti, indica un token sconosciuto e l’impossibilità di collegarlo ad una grammatica nota.
Nel secondo esempio viene mostrato il comportamento del compilatore in presenza di una variabile “b” invocata ma non dichiarata. Il compilatore segnala quindi l’assenza di dichiarazione della variabile.
Nel terzo ed ultimo esempio viene mostrato il comportamento del compilatore in presenza di una dichiarazione di una variabile “x” già dichiarata precedentemente. Il compilatore segnala quindi che la variabile è già stata dichiarata.

# 3. ESERCIZIO 2 - TABELLA DEI SIMBOLI

## 3.1. IMPLEMENTAZIONE

La Symbol Table è stata implementata come una ArrayList di HashMap, le quali rappresentano ognuna uno scope differente interno a quello globale e mappano ogni entry della tabella (STentry) in corrispondenza dell'ID della variabile o funzione dichiarata.
Nel caso delle funzioni, ovvero di ambienti "usa e getta", i rispettivi contesti vengono creati, controllati e rimossi non appena si esce dal blocco.
La realizzazione della Symbol Table è affidata ai metodi "checkSemantics" di tutti i nodi, che verificano la validità della Symbol Table scorrendo l'albero durante una visita. Al loro interno, vengono eseguite tutte le operazioni per popolare la Symbol Table, quali aggiunte di STentry in caso di dichiarazione, creazione ed eliminazione di nuovi livelli, e così via. Gli errori raccolti durante la visita vengono poi mostrati all'utente, e nel caso in cui ve ne siano la compilazione viene interrotta e il programma rigettato.
Di seguito un esempio di Symbol Table:

| **Programma in Input** | **Symbol Table** |
|---|---|
| int a;<br>bool b;<br><br>int f(int n, bool c){<br>int x;<br>x = 2;<br>x<br>}<br><br>b = true; | Entry ID: a<br>Tipo: IntType<br>Offset: 1<br>Nesting: 0<br>Label:<br>Initialized: false<br>Entry ID: b<br>Tipo: BoolType<br>Offset: 2<br>Nesting: 0<br>Label:<br>Initialized: true<br>Entry ID: f<br>Tipo: ArrowType<br>Offset: 3<br>Nesting: 0<br>Label: function0<br>Initialized: true |

Come si può notare dall'esempio precedente, nella Symbol Table finale sono presenti gli identificatori 'a', 'b' ed 'f', che sono stati dichiarati nell'ambiente più esterno, cosa che non accade nel caso di 'x', in quanto è stata dichiarata all'interno della funzione 'f' ed è quindi visibile solo al suo interno. Possiamo notare che, correttamente, la variabile 'b' appare inizializzata, mentre 'a' no.

## 3.2. STRUTTURA DI STENTRY

La classe STentry rappresenta una singola entry della Symbol Table, ovvero ogni variabile che viene dichiarata in un certo scope. Le informazioni salvate per ognuna di essa sono le seguenti:

- Type: rappresenta il tipo della variabile (int/bool) o della funzione (int/bool/void).
- Offset: rappresenta la posizione della variabile nello scope in cui viene dichiarata.
- Nesting: indica il livello di nesting in cui è stata dichiarata la variabile/funzione (l'ambiente globale ha livello di nesting uguale a zero).
- Label: rappresenta l'etichetta che è stata attribuita alla specifica funzione.
- Initialized: booleano che indica se la variabile è stata anche inizializzata (true) o meno (false) oltre che dichiarata.

## 3.3. ESEMPI

In questa sezione è stata posta particolare attenzione nell'individuazione delle variabili/identificatori non dichiarati oppure dichiarati molteplici volte. Nella seguente tabella sono riportati degli esempi di segnalazione di questi errori:

| **Codice** | **Output** |
|---|---|
| int a;<br>int b;<br>b = 2;<br>b | Type checking ok! |
| int a;<br>int f(int x){<br>x<br>}<br><br>b = f(1); | You had 1 errors:<br>Var id 'b' not declared |
| int a;<br>int b;<br><br>bool a;<br>a = true; | You had 1 errors:<br>Var id 'a' already declared |

Nel primo esempio è possibile notare in azione il compilatore che analizza un programma corretto, restituendo quindi un messaggio positivo e mostrando che tutto è andato per il verso giusto.
Nel secondo esempio è possibile notare il compilatore che segnala come la variabile "b" non sia dichiarata. Essa infatti è oggetto di un assegnamento senza però essere prima dichiarata. Il compilatore segnala quindi l'errore.
Nel terzo ed ultimo esempio possiamo notare il compilatore che segnala la molteplice dichiarazione della variabile "a".

# 4. ESERCIZIO 3 - ANALISI SEMANTICA

Una volta completata l'implementazione della Symbol Table è stata sviluppata la parte di analisi semantica, ovvero verificare la correttezza dei tipi ed in particolare la correttezza del numero e del tipo dei parametri attuali nel caso di chiamate di funzione. Per fare ciò, è stata implementata per tutti i nodi dell'AST la funzione typeCheck.

## 4.1. PREMESSE E FUNZIONI DI SUPPORTO

Funzione **top**: dato in input una pila di ambienti Γ:Γ1x…xΓn restituisce l'ultimo ambiente inserito.
Funzione **choose(**Γ1,Γ2**)**: dati due ambienti ritorna un terzo ambiente Γ3 che contiene lo "scenario peggiore" per le variabili che vengono inizializzate all'interno dei rami THEN o ELSE di un if, seguendo le regole della tabella sottostante:

| **Stato variabile in Γ1** | **Stato variabile in Γ2** | **Stato variabile in Γ3** |
|---|---|---|
| dec | dec | dec |
| init | dec | dec |
| dec | init | dec |
| init | init | init |

dove lo stato 'dec' indica che la variabile in questione è stata solamente dichiarata, mentre lo stato 'init' indica che la variabile in questione è stata sia dichiarata che inizializzata.

## 4.2. REGOLE DI INFERENZA

Per le seguenti regole mostrate, si assume non sia necessario scrivere tutte le regole per ogni possibile combinazione causata dalle chiusure di Kleene.

### 4.2.1. PROG:

a) 'singleExp', rappresenta un programma costituito solo da un elemento di tipo EXP.

$$\frac{\emptyset \circ [\,],0 \vdash e : init, T}{\emptyset, 0 \vdash e : T}$$

b) ‘decstmExp’, rappresenta un programma costituito da almeno un elemento DEC seguito da un qualsiasi numero di elementi STM ed infine una EXP facoltativa.

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

### 4.2.2. DEC:

a) ‘idDec’, rappresenta la dichiarazione di variabili.

$$\frac{id \notin dom\big(top(\Gamma, n)\big)}{\Gamma, n \vdash T\ id : \Gamma[id \to dec, T], n + 1}$$

$$\frac{\Gamma,\ n \vdash d : \Gamma',\ n' \qquad \Gamma',\ n' \vdash D :\ \Gamma'',\ n''}{\Gamma,\ n\ \vdash d\ D : \Gamma'',\ n''}$$

b) ‘funDec’, rappresenta la dichiarazione di funzione.

$$\frac{\begin{array}{c} f \notin dom(top(\ \Gamma, n)) \\ \Gamma[f \to (T1, \ldots, Tn) \to T,\ x1 \to T1,\ \ldots\ ,\ xn \to Tn\,],\ n \vdash body : T' \quad T = T' \end{array}}{\Gamma,\ n \vdash T\ f(T1\ x1,\ \ldots\ ,\ Tn\ xn)\{body\} :\ \Gamma[f \to (T1,\ \ldots\ ,\ Tn) \to T],\ n + 1}$$

### 4.2.3. BODY:

Rappresenta il corpo di una funzione, composto da un numero qualsiasi di elementi DEC seguito da un numero qualsiasi di STM ed infine una EXP facoltativo.

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

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

### 4.2.4. STM:

a) ‘varStm’, rappresenta l’assegnamento di un elemento EXP ad una variabile.

$$\frac{\Gamma, n \vdash id : S, T \quad \Gamma, n \vdash exp : init, T' \quad T = T'}{\Gamma, n \vdash \ id = exp : \Gamma[id \rightarrow init, T], n}$$

dove S ∈ {dec, init}

b) ‘funStm’, rappresenta la chiamata di funzione.

$$\frac{f \in dom\big(top(\Gamma, n)\big) \quad \Gamma, n \vdash f : T1x \ldots xTn \rightarrow T \quad (\Gamma, n \vdash ei : Ti')i \in 1 \ldots n \quad (Ti = Ti')i \in 1 \ldots n}{\Gamma, n \vdash f(e1, \ldots, e_n) : \Gamma, n}$$

c) ‘ifStm’, rappresenta il costrutto ‘IF’(guardia){ramoThen}’ELSE’{ramoElse}, in cui la guardia deve essere un elemento EXP di tipo bool ed i due rami then ed else sono degli elementi BlockSeqStm. Il ramo else in questo caso può essere omesso.

$$\frac{\begin{array}{c}\Gamma, n \vdash e : init, bool \quad \Gamma, n \vdash BlockSeqStm1 : \Gamma', n \\ \Gamma, n \vdash BlockSeqStm2 : \Gamma'', n \\ \Gamma''' = choose(\Gamma', \Gamma'')\end{array}}{\Gamma, n \vdash if(e)\{BlockSeqStm1\}else\{BlockSeqStm2\} : \Gamma''', n}$$

### 4.2.5. BlockSeqStm:

Rappresenta il possibile contenuto dei rami then ed else dell’elemento ‘ifStm’ ed è composto da una sequenza di almeno un elemento STM.

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

### 4.2.6. EXP:

a) ‘intExp’, rappresenta un qualsiasi numero intero.

$$\frac{}{\Gamma,\ n\ \vdash Int : init\ ,\ int}$$

b) ‘trueExp’, rappresenta il valore booleano true.

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

c) ‘falseExp’, rappresenta il valore booleano false.

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

d) ‘varExp’, rappresenta un ID, che identifica una variabile.

$$\frac{\Gamma, n(id) = S, T}{\Gamma, n \vdash id: S, T}$$

dove S ∈ {dec, init}

e) ‘notExp’, rappresenta l’operatore logico NOT fra elementi EXP di tipo bool e restituisce un bool.

$$\frac{\Gamma, n \vdash exp: init, bool \quad !: bool \rightarrow bool}{\Gamma, n \vdash !\, exp : init, bool}$$

f) ‘muldivExp’, rappresenta l'operazione di moltiplicazione e divisione fra elementi EXP di tipo int e restituisce un int.

$$\frac{\Gamma, n \vdash exp1: init, int \quad \Gamma, n \vdash exp2: init, int \quad *: intxint \rightarrow int}{\Gamma, n \vdash exp1 * exp2 : init, int}$$

$$\frac{\Gamma, n \vdash exp1: init, int \quad \Gamma, n \vdash exp2: init, int \quad /: intxint \rightarrow int}{\Gamma, n \vdash exp1 \,/\, exp2 : init, int}$$

g) ‘addsubExp’, rappresenta l'operazione di addizione e sottrazione fra elementi EXP di tipo int e restituisce un int.

$$\frac{\Gamma, n \vdash exp1 : init, int \quad \Gamma, n \vdash exp2 : init, int \quad + : intxint \rightarrow int}{\Gamma, n \vdash exp1 + exp2 : init, int}$$

$$\frac{\Gamma, n \vdash exp1 : init, int \quad \Gamma, n \vdash exp2 : init, int \quad - : intxint \rightarrow int}{\Gamma, n \vdash exp1 - exp2 : init, int}$$

h) ‘compareExp’, rappresenta gli operatori di confronto, ovvero maggiore/ maggiore-uguale/minore/minore-uguale che confrontano due elementi EXP di tipo int e forniscono un bool, ed infine l’operatore di uguaglianza che confronta due elementi EXP dello stesso tipo (bool o int) e restituisce un bool.

$$\frac{\Gamma, n \vdash exp1 : init, int \quad \Gamma, n \vdash exp2 : init, int \quad \leq : intxint \rightarrow bool}{\Gamma, n \vdash exp1 \leq exp2 : init, bool}$$

$$\frac{\Gamma, n \vdash exp1 : init, int \quad \Gamma, n \vdash exp2 : init, int \quad < : intxint \rightarrow bool}{\Gamma, n \vdash exp1 < exp2 : init, bool}$$

$$\frac{\Gamma, n \vdash exp1 : init, int \quad \Gamma, n \vdash exp2 : init, int \quad \geq : intxint \rightarrow bool}{\Gamma, n \vdash exp1 \geq exp2 : init, bool}$$

$$\frac{\Gamma, n \vdash exp1 : init, int \quad \Gamma, n \vdash exp2 : init, int \quad > : intxint \rightarrow bool}{\Gamma, n \vdash exp1 > exp2 : init, bool}$$

$$\frac{\Gamma, n \vdash exp1 : init, T' \quad \Gamma, n \vdash exp2 : init, T'' \quad T' = T'' \quad == : TxT \rightarrow bool}{\Gamma, n \vdash exp1 == exp2 : init, bool}$$

i) 'andorExp', rappresenta gli operatori logici AND e OR, che prendono due elementi EXP di tipo bool e restituiscono un bool.

$$\frac{\Gamma, n \vdash exp1 : init, bool \quad \Gamma, n \vdash exp2 : init, bool \quad \&\& : boolxbool \rightarrow bool}{\Gamma, n \vdash exp1 \;\&\&\; exp2 : init, bool}$$

$$\frac{\Gamma, n \vdash exp1 : init, bool \quad \Gamma, n \vdash exp2 : init, bool \quad || : boolxbool \rightarrow bool}{\Gamma, n \vdash exp1 \;||\; exp2 : init, bool}$$

j) 'ifExp', rappresenta il costrutto 'IF'(guardia){ramoThen}'ELSE'{ramoElse}, in cui la guardia deve essere un elemento EXP di tipo bool ed i due rami then ed else sono degli elementi BlockSeqStmExp.

$$\frac{\begin{array}{c}\Gamma, n \vdash e : init, bool \quad \Gamma, n \vdash BlockSeqStmExp1 : \Gamma', n, T' \\ \Gamma, n \vdash BlockSeqStmExp2 : \Gamma'', n, T'' \\ T' = T'' \quad \Gamma''' = choose(\Gamma', \Gamma'')\end{array}}{\Gamma, n \vdash if(e)\{BlockSeqStmExp1\}else\{BlockSeqStmExp2\} : init, T'}$$

k) 'funExp', rappresenta la chiamata di funzione utilizzabile come EXP.

$$\frac{\begin{array}{c}f \in dom\big(top(\Gamma, n)\big) \quad \Gamma, n \vdash f : T1x \ldots xTk \rightarrow T \\ (\Gamma, n \vdash ei : Ti')i \in 1 \ldots k \quad (Ti = Ti')i \in 1 \ldots k\end{array}}{\Gamma, n \vdash f(e1, \ldots, ek) : init, T}$$

### 4.2.7. BlockSeqStmExp:

Rappresenta il possibile contenuto dei rami then ed else dell'elemento 'ifExp' ed è composto da una sequenza di un numero qualsiasi di elementi STM seguiti da un elemento EXP.

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

## 4.3. CONTROLLO TIPI

Come anticipato, di seguito una tabella con degli esempi di verifica della correttezza dei tipi, in particolare numero e tipo dei parametri attuali se conformi al numero e tipo dei parametri formali:

| **Codice da Verificare** | **Output** |
|---|---|
| int f(int a, int b){<br>a<br>}<br><br>f(2) | Wrong number of parameters in the invocation of ‘f’.<br>Type checking is WRONG! |
| int g(bool a, int b){<br>b<br>}<br><br>g(2,3) | Wrong type for 1-th parameter in the invocation of 'g'.<br>Type checking is WRONG! |

Nel primo esempio, il compilatore segnala un errore dovuto al fatto che la funzione ‘f’ viene chiamata con un numero di parametri sbagliato. Infatti, come possiamo vedere dalla dichiarazione di ‘f’, essa prende in input due parametri di tipo intero. Successivamente invece essa viene chiamata con un singolo parametro. Il compilatore segnala quindi errore e il Type Checking fallisce.
Nel secondo esempio, il compilatore segnala un errore dovuto al fatto che la funzione ‘g’ viene chiamata con un parametro di tipo sbagliato. Infatti, come possiamo vedere dalla dichiarazione di ‘g’, essa prende in input due parametri di tipo ‘bool’ e ‘int’. Successivamente invece essa viene chiamata con due parametri di tipo ‘int’. Il compilatore segnala quindi errore e il Type Checking fallisce.

# 4.4. CODICE DA CONTROLLARE

| **Codice da Verificare** | **Output** |
|---|---|
| int a;<br>int b;<br>int c;<br>c = 2;<br>if (c > 1) { b = c ; }<br>else { a = b ; } | Type Error: Var ‘b’ not initialized |
| int a;<br>int b;<br>int c ;<br>void f(int n){<br>int x ; int y ;<br>if (n > 0) { x = n ;}<br>else { y = n+x ;}<br>}<br>c = 1;<br>f(0) | Type Error: Var 'x' not initialized |
| void h(int n){<br>int x ; int y ;<br>if (n==0){ x = n+1 ;}<br>else { h(n-1) ; x = n ; y = x ;}<br>}<br>h(5) | Type checking ok! |
| int a;<br>void h(int n){<br>int x ;<br>int y ;<br>if (n==0){ x = n+1 ;}<br>else { h(n-1) ; y = x ;}<br>}<br>h(5) | Type Error: Var 'x' not initialized |

Nel primo esempio il compilatore segnala errore a causa della non-inizializzazione di ‘b’ nell’assegnamento. Essa infatti, nel momento in cui si entra nell’if, non è inizializzata. Il compilatore segnala quindi l’errore, in quanto nel ramo else essa non può essere utilizzata per un assegnamento.
Nel secondo esempio il compilatore segnala errore a causa della non-inizializzazione di ‘x’ nell’assegnamento. Essa infatti, nel momento in cui si entra nell’if, non è inizializzata. Il compilatore segnala quindi l’errore, in quanto nel ramo else essa non può essere utilizzata per un assegnamento.
Nel terzo esempio non è presente nessun errore di tipo. Il Type Checking si conclude quindi positivamente.
Nel quarto ed ultimo esempio il compilatore segnala errore a causa della non-inizializzazione di ‘x’ nell’assegnamento. Essa infatti, all’interno della funzione ‘h’, nel momento in cui si

entra nell’if, non è inizializzata. Il compilatore segnala quindi l’errore, in quanto nel ramo else essa non può essere utilizzata per un assegnamento.

# 5. ESERCIZIO 4 - CODEGEN E INTERPRETE

In questo capitolo viene descritto il linguaggio bytecode con cui è stato generato il codice, com'è stata gestita la memoria ed infine l'esecuzione di alcuni sorgenti di esempio.

## 5.1. LINGUAGGIO BYTECODE

Di seguito la lista delle istruzioni che sono state utilizzate per generare il codice Assembly:

| **Istruzione Assembly** | **Funzionamento** |
|---|---|
| 'load' $R1 NUMBER '(' $R2 ')' | Carica il valore di $R1 all'indirizzo di memoria NUMBER '(' $R2 ')'. |
| 'store' $R1 NUMBER '(' $R2 ')' | Inserisce il valore all'indirizzo di memoria NUMBER '(' $R2 ')' in $R1 |
| 'storei' $R NUMBER | Inserisce NUMBER in $R. |
| 'move $R1 $R2' | Sposta il valore di $R2 in $R1. |
| 'add' $R1 $R2 | Inserisce in testa alla pila il valore $R1 + $R2. |
| 'addi' $R NUMBER | Inserisce in testa alla pila il valore $R1 + NUMBER. |
| 'sub' $R1 $R2 | Inserisce in testa alla pila il valore $R1 - $R2. |
| 'subi' $R NUMBER | Inserisce in testa alla pila il valore $R - NUMBER. |
| 'mul' $R1 $R2 | Inserisce in testa alla pila il valore $R1 * $R2. |
| 'muli' $R NUMBER | Inserisce in testa alla pila il valore $R * NUMBER. |
| 'div' $R1 $R2 | Inserisce in testa alla pila il valore $R1 / $R2. |
| 'divi' $R NUMBER | Inserisce in testa alla pila il valore $R / NUMBER. |
| 'push' (n=NUMBER \| l=LABEL) | memory[sp] = number\|label , sp = sp-1 |
| 'pushr' $R | Inserisce il valore contenuto in $R in testa alla pila. |
| 'pop' | Rimuove l'ultimo elemento salvato nella pila. |
| 'popr' $R | Rimuove l'ultimo elemento salvato nella pila e lo mette in $R. |
| 'b' LABEL | Salta all'istruzione specificata da LABEL. |

| | |
|---|---|
| ‘beq’ $R1 $R2 LABEL | Se $R1 == $R2 salta all’istruzione specificata da LABEL. |
| ‘bleq’ $R1 $R2 LABEL | Se $R1 <= $R2 salta all’istruzione specificata da LABEL. |
| ‘bl’ $R1 $R2 LABEL | Se $R1 < $R2 salta all’istruzione specificata da LABEL. |
| ‘bg’ $R1 $R2 LABEL | Se $R1 > $R2 salta all’istruzione specificata da LABEL. |
| ‘bgeq’ $R1 $R2 LABEL | Se $R1 >= $R2 salta all’istruzione specificata da LABEL. |
| ‘jsub’ LABEL | Usata nella chiamata di funzione, salta all’indirizzo specificato da LABEL. |
| ‘rsub’ $R | Usata nella definizione di funzione, una volta completata l’esecuzione torna all’indirizzo $R. |
| ‘LABEL ':'' | Creazione dell’etichetta. |
| ‘halt’ | Termina l’esecuzione del programma. |

## 5.2. GESTIONE DELLA MEMORIA

Per gestire correttamente la pila sono stati utilizzati i seguenti registri:

| **Registro** | **Utilizzo** |
|---|---|
| $A0 | Contiene il valore dell’ultima istruzione calcolata, utilizzato come salvataggio per ogni valore di ritorno. |
| $T1, $T2 | Registri temporanei. |
| $SP | Stack Pointer: contiene l’ultimo indirizzo di memoria utilizzato, in tutti gli indirizzi più bassi ci sarà memoria da considerarsi libera mentre in tutti gli indirizzi più alti si troverà la pila del programma. |
| $FP | Frame Pointer: contiene l’indirizzo del record di attivazione corrente, ogni volta che viene creato un nuovo record di attivazione il vecchio FP viene salvato nello stack mentre questo registro viene aggiornato con l’indirizzo del nuovo record. |
| $RA | Return Address: registro utilizzato esclusivamente dalla chiamata a funzione; nel momento in cui si presenta l’istruzione di ‘JAL’ viene salvato in questo registro l’indirizzo dell’istruzione successiva in modo da permettere di ritornare al normale flusso del programma una volta terminata la funzione. |
| $AL | Access Link: contiene l’indirizzo del record di attivazione del blocco sintattico che li racchiude. |

All’avvio l’interprete imposta i registri $sp e $fp al valore ‘MEMSIZE’ che indica il valore massimo della memoria.
Al momento della dichiarazione di una variabile o di una funzione viene allocato uno spazio nello Stack, indipendentemente dal loro tipo.

### 5.2.1. VISIBILITÀ

- **Variabili**: sono visibili nel record di attivazione in cui sono state dichiarate. È possibile dichiarare variabili globali visibili da ogni punto del programma, oppure la dichiarazione all’interno del corpo delle funzioni, le quali non sono visibili al di fuori della funzione stessa. Ogni variabile dichiarata, come ogni parametro passato ad una funzione, crea un nuovo spazio in memoria, modificando l’offset del contesto in cui si trova, aumentano quindi l’$SP.
- **Funzioni**: possono essere dichiarate nell’ambiente più esterno e anche all’interno di funzioni stesse. Oltre ad essere visibili in qualsiasi blocco successivo sono anche visibili all’interno di se stesse, consentendo la ricorsione, e all’interno delle funzioni dichiarate successivamente.
- **Label**: la generazione e la gestione delle label è affidata alla classe SimpLanPlusLib, la quale accumula anche il codice di tutte le funzioni dichiarate.

## 5.3. CODICE DA CONTROLLARE

Per la verifica del seguente codice è stata disabilitata la verifica semantica per poter eseguire comunque il codice che altrimenti non rispetterebbe la grammatica fornita.

| **Codice da Verificare** | **Output** |
|---|---|
| int x ;<br>void f(int n){<br>if (n == 0) { n = 0 ; }<br>else { x = x * n ; f(n-1) ; }<br>}<br>x = 1 ;<br>f(10) | Result: 0 |
| int u ;<br>int f(int n){<br>int y ;<br>y = 1 ;<br>if (n == 0) { y }<br>else { y = f(n-1) ; y*n }<br>}<br>u = 6 ;<br>f(u) | Result: 720 |
| int u ;<br>void f(int m, int n){<br>int x;<br>if (m>n) { u = m+n ;}<br>else {x = 1 ; f(m+1,n+1) ; }<br>}<br>f(5,4) ;<br>u | Result: 9 |
| int u ;<br>void f(int m, int n){<br>int x;<br>if (m>n) { u = m+n ;}<br>else {x = 1 ; f(m+1,n+1) ; }<br>}<br>f(4,5) ;<br>u | Index -1 out of bounds for length 1000 |

Nel primo esempio, il compilatore restituisce l'ultimo valore presente in A0, ovvero 0. Infatti, dopo aver effettuato la ricorsione nel ramo else, l'ultima operazione svolta è l'assegnamento 'n = 0' nel ramo then. In A0 quindi l'ultimo valore presente sarà 0.
Nel secondo esempio, il compilatore restituisce l'ultimo valore presente in A0, ovvero 720. Infatti, dopo aver effettuato ricorsivamente i vari assegnamenti ad 'y' nel ramo else, l'ultima operazione svolta è la stampa di 'y' nel ramo then, ovvero 720.
Nel terzo esempio, il compilatore restituisce l'ultimo valore presente in A0, ovvero 9. Infatti, una volta entrati nella funzione, essendo 'm>n' si entrerà nel ramo then. Il valore di A0 sarà quindi il risultato dell'assegnamento a 'u' di 'm+n', ovvero 9.

Nel quarto ed ultimo esempio, richiamando la funzione ‘f’ con i valori [4,5] invece di [5,4], il compilatore andrà in loop infinito, in quanto non si raggiungerà mai la fine della ricorsione, causando quindi un out of bounds.