

Corso di Complementi di Linguaggi di Programmazione



Progetto SimpLanPlus

Partecipanti Gruppo

Cannas Elia [0001097520]

Abruzzese Michele [0001097676]

Corso di Laurea Magistrale in Informatica
Anno Accademico 2022/23

Indice

1. Introduzione	2
2. Analisi Lessicale	3
3. Analisi Semantica	5
3.1. CheckSemantic	6
3.1.1. Uso di variabili non inizializzate	7
3.2. TypeCheck	8
4. Generazione di codice intermedio	13

1 Introduzione

Lo scopo di questo progetto è stato quello di implementare un compilatore e interprete per un linguaggio di programmazione chiamato “SimplanPlus”. Il progetto in questione è parte integrante dell’esame di “Complementi di linguaggi di programmazione” per l’anno 2022/2023.

SimplanPlus è un semplice linguaggio imperativo, in cui:

- I tipi includono anche il tipo void
- Le dichiarazioni di variabili sono: type ID (senza inizializzazione)
- Le funzioni possono essere ricorsive (ma non mutuamente)
- Ci sono dei comandi (un programma o il corpo di una funzione può essere *stm* oppure *dec stm*)
- Sono ammessi corpi di funzioni del tipo $\{stm; exp\}$ e in questo caso la funzione, dopo aver valutato *stm* restituisce il valore di *exp*.

In seguito all’installazione e configurazione di tutti i componenti necessari (ANTLR, Java e IntelliJ) è stato realizzato il sistema.

La consegna riporta 4 punti principali da seguire per il completamento:

1. Analisi lessicale che restituisce la lista degli errori lessicali in un file di output;
2. Sviluppo della Symbol Table del programma e verifica degli identificatori/funzioni **non dichiarati** e identificatori/funzioni **dichiarati più volte** nello stesso ambiente;
3. Sviluppo di un’analisi semantica che verifichi sia la **correttezza dei tipi** come numero e tipo di parametro attuale se conformi a quelli formali e **uso di variabili non inizializzate** assumendo che le funzioni non accedono mai alle variabili globali;
4. Implementazione dell’interprete di SimplanPlus;

2 Analisi Lessicale

L'analisi lessicale fa riferimento al primo esercizio della consegna del progetto e consiste nel creare un analizzatore lessicale che ritorni una lista di errori lessicali all'interno di un file di output.

Codice utilizzato

```
simplanPlusLexer lexer = new simplanPlusLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);

tokens.fill();

List<Token> errorLexer = new ArrayList<>();
for(Token t : tokens.getTokens()){
    if(t.getType() == simplanPlusLexer.ERR){
        errorLexer.add(t);
        System.out.print(t);
    }
}
```

Scorriamo la lista dei token e per ogni token, controlliamo se il tipo del token è uguale a simplanPlusLexer.ERR.

Con questo controllo ci accertiamo che il token non è riconosciuto dalla grammatica e lo aggiungiamo alla lista degli errori lessicali.

```
for(int i=0; i<errorLexer.size();i++){

    int lineErr = errorLexer.get(i).getLine();
    String strErr = errorLexer.get(i).getText();
    int posErr = errorLexer.get(i).getCharPositionInLine()+1;

    String write = "Errore "+i+1+" :linea "+lineErr+" carattere numero "+posErr+" = "+strErr+"\n";

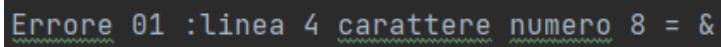
    Files.write(Paths.get( first: "out/errors.txt"), write.getBytes(), StandardOpenOption.APPEND);

}
```

Scriviamo in un file l'errore con le proprie caratteristiche.

3 Esempi di codici testati:

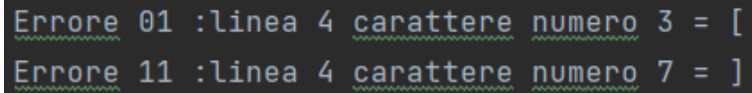
```
1) void h(bool n){  
    int x; int y;  
}  
h(true & true);
```



Errore 01 :linea 4 carattere numero 8 = &

In questo caso il token "&" non è riconosciuto.

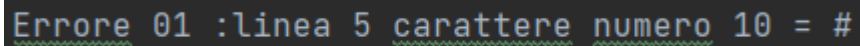
```
2) int a;  
a = 3;  
  
if[a>1]{  
    a  
}
```



Errore 01 :linea 4 carattere numero 3 = [
Errore 11 :linea 4 carattere numero 7 =]

In questo caso i token "[" "]" non sono riconosciuti.

```
3) int a;  
a = 3;  
  
if(a>1){  
    a=5;  #  
}
```



Errore 01 :linea 5 carattere numero 10 = #

In questo caso il token "#" non è riconosciuto.

3 *Analisi Semantica*

Il secondo esercizio consisteva nella creazione di una tabella dei simboli del programma in cui venissero controllati:

- Identificatori/funzioni **non dichiarati**
- Identificatori/funzioni **dichiarati più volte** nello stesso ambiente

La Symbol Table ricopre un ruolo importante nell'analisi di scope e identificatori poiché ci consente di poter monitorare le variabili e i loro livelli di annidamento. Il suo funzionamento è uguale a quello della funzione Γ , che accetta un **Id** in input e restituisce la corrispondente **STentry** che conterrà informazioni come Tipo, NestingLevel e Offset. Abbiamo anche utilizzato un attributo booleano "init" che sfruttiamo per controllare se le variabili utilizzate sono state inizializzate (Esercizio 3.2).

La SymbolTable è stata implementata come un ArrayList di HashMap. Nella ArrayList, l'indice rappresenta il livello di annidamento, mentre l'hashmap rappresenta lo scope associato a quel livello di annidamento.

SymbolTable		
symbol_table	ArrayList<HashMap<String, STentry>>	
offset	ArrayList<Integer>	
add(HashMap<String, STentry>)		void
common(SymbolTable)	HashMap<String, STentry>	
equals(Object)		boolean
getOffset()	ArrayList<Integer>	
getSymbol_table()	ArrayList<HashMap<String, STentry>>	
hashCode()		int
increaseoffset()		void
insert(String, Type, int, String)		void
lookup(String)		STentry
nesting()		Integer
nslookup(String)		Integer
remove()		void
setOffset(ArrayList<Integer>)		void
setSymbol_table(ArrayList<HashMap<String, STentry>>)		void
top_lookup(String)		boolean

STentry		
type	Type	
offset	int	
nesting	int	
label	String	
init	Boolean	
equals(Object)	boolean	
getInit()	Boolean	
getlabel()	String	
getnesting()	int	
getoffset()	int	
gettype()	Type	
hashCode()	int	
initVar()	void	
isInit()	boolean	
setInit(Boolean)	void	
toString()	String	

Nelle immagini sopra possiamo vedere le due classi SymbolTable e STentry.

Le operazioni principali di **SymbolTable** sono le seguenti:

- **add**. Estende la SymbolTable con un nuovo scope;
- **insert**. Inserisce l'id di tipo T nella SymbleTable;
- **lookup**. Restituisce la STentry associata all'id se è presente nella SymbleTable;
- **top_lookup**. Restituisce la STentry associata all'id se è presente nella SymbleTable nell'ultimo scope;
- **remove**. Esce dallo scope corrente;

3.1 CheckSemantics

Dopo aver costruito l'AST tramite la visita implementata in **simplanPlusVisitorImpl**, eseguiamo la CheckSemantics sull'albero che ci restituisce un ArrayList di errori semantici se ci sono.

I principali aggiornamenti per quanto riguarda lo **scope** avvengono in **ProgDecStmNode** (nodo che descrive la regola del programma) e **FunNode** (nodo che descrive la dichiarazione di funzione). In queste due classi viene utilizzato il metodo **add()**, per aggiungere un nuovo scope alla SymbolTable. All'uscita viene utilizzato il metodo **remove()** per uscire dallo scope.

Per quanto riguarda il controllo di identificatore o funzione **non dichiarato**, utilizziamo il metodo **lookup** che, se ci restituisce un oggetto null, creiamo un errore semantico di variabile non dichiarata. (es. CallNode e IdNode)

Per controllare che le variabili o le funzioni non siano state **dichiarate più volte** nello stesso scope utilizziamo il metodo **top_lookup** che, se ci restituisce un STentry non nullo, creiamo un errore semantico di variabile già dichiarata nello stesso scope.

Nei nodi **DecNode** e **FunNode** utilizziamo il metodo **insert()** in quanto stiamo dichiarando rispettivamente variabili e funzioni.

3.1.1 Uso di variabili non inizializzate

Nella consegna si chiede, come punto opzionale, di verificare l'utilizzo di variabili non inizializzate. Per implementare questo controllo ci siamo avvalsi dell'attributo booleano **init** nella classe **STentry**.

Un primo controllo avviene nella classe **AssNode** (classe che rappresenta la regola dell'assegnamento). In questo caso dobbiamo impostare l'attributo **init** dell'id del nodo sinistro a **true**, in modo tale che venga marcato come inizializzato. Facciamo questo dopo aver controllato che la variabile non è stata ancora inizializzata, altrimenti andiamo avanti.

Controlliamo invece che in **IdNode** la variabile sia stata già inizializzata, altrimenti non può essere utilizzata (**ST.lookup(id).init()**).

Un altro punto importante per far sì che non vengano utilizzate variabili non inizializzate è il controllo nell'**if**.

- Nell'**if** è possibile che una variabile venga **inizializzata in uno solo dei due rami**, quindi la variabile è inizializzata solo ed esclusivamente per quel ramo, ma non può essere utilizzata né nell'altro ramo né all'uscita dell'**if**.
- Un altro caso possibile nell'**if** è rappresentato dall'**inizializzazione della stessa variabile in entrambi i rami**, in questo caso la variabile è inizializzata per entrambi i rami e anche fuori dall'**if**.
- Possiamo avere l'**if** solo con il ramo **then**, in questo caso se **inizializziamo una variabile all'interno del ramo then**, non possiamo utilizzarla fuori perchè è inizializzata esclusivamente nel **then** e non sappiamo se si entra nel ramo o meno. In questo caso, la **SymbolTable** all'uscita del ramo sarà quella iniziale;

Per implementare le situazioni sopra descritte, andiamo a fare **due copie della SymbolTable**, una che utilizzeremo per fare la **CheckSemantics** per il ramo **then** e una che utilizzeremo per fare la **CheckSemantics** per il ramo **else**.

Dopo aver eseguito le **CheckSemantics** dei due rami possiamo avere vari scenari:

- Le **tabelle sono uguali** quindi abbiamo iniziato solo ed esclusivamente le stesse variabili da entrambe le parti quindi dopo l'**if** le variabili devono

risultare inizializzate. A questo punto assegnamo alla SymbolTable iniziale, di cui avevamo fatto una copia, una delle due SymbolTable dei due rami.

- Le **tabelle sono diverse ma hanno qualcosa in comune**, quindi significa che alcune variabili sono state inizializzate in entrambi i rami, mentre altre variabili sono state inizializzate solo in uno dei due rami. In questo caso assegno alla SymbolTable iniziale, di cui avevamo fatto una copia, una SymbolTable costituita solo dalle variabili che hanno in comune i due rami. (per controllare le cose in comune utilizziamo il metodo **common()** di SymbolTable)
- Le **tabelle sono diverse**, questo significa che abbiamo inizializzato variabili diverse in entrambi i rami, quindi fuori dall'if non saranno inizializzate.

3.2 TypeCheck

In questa fase andiamo a verificare la correttezza dei tipi.

La grammatica SimpLanPlus prevede tre tipi che sono: **void**, **int**, e **bool**.

Il tipo void è possibile utilizzarlo solo per le funzioni, per una nostra scelta, in quanto non vi è motivo di utilizzarlo per le variabili.

Per ogni nodo implementiamo il metodo typeCheck() che restituisce un **Type** o un **ErrorType**. Passiamo come **parametro** a typeCheck() un **ArrayList di Type** per inserire tutti gli ErrorType in modo tale da non far continuare l'esecuzione del programma se ci sono errori di tipo.

Le regole di inferenza utilizzate sono le seguenti:

$$\frac{\emptyset \vdash exp:T}{\emptyset \vdash exp:T} [progSimple] \quad \frac{\emptyset \bullet [] \vdash dec:\Gamma \quad \Gamma \vdash stm:\Gamma' \quad \Gamma' \vdash exp:T}{\emptyset \vdash dec \quad stm \quad exp:T} [progComplex]$$

$$\frac{\emptyset \bullet [] \vdash dec:\Gamma \quad \Gamma \vdash stm}{\emptyset \vdash dec \quad stm:void} [progDecStm] \quad \frac{\emptyset \bullet [] \vdash dec:\Gamma \quad \Gamma \vdash exp:T}{\emptyset \vdash dec \quad exp:T} [progDecExp]$$

$$\frac{\emptyset \bullet [] \vdash dec: \Gamma}{\emptyset \vdash dec: void} [progDec]$$

$$\frac{Id \notin dom(top(\Gamma))}{\Gamma \vdash T Id: \Gamma[Id \rightarrow T]} [decVar]$$

$$\frac{\Gamma[f \mapsto (T1, \dots, Tn) \rightarrow T, x1 \mapsto T1, \dots, xn \mapsto Tn] \vdash body: T' \quad T' = T \quad f \notin dom(top(\Gamma))}{\Gamma \vdash T f(T1 x1, \dots, Tn xn) \{body\}: \Gamma[f \mapsto (T1, \dots, Tn) \rightarrow T]} [funDec]$$

$$\frac{\Gamma \bullet [] \vdash dec: \Gamma' \quad \Gamma' \vdash stm: \Gamma'' \quad \Gamma'' \vdash exp: T}{\Gamma \vdash dec \quad stm \quad exp: T} [body]$$

$$\frac{\Gamma \vdash stm: \Gamma' \quad \Gamma' \vdash exp: T}{\Gamma \vdash stm \quad exp: T} [bodyStmExp]$$

$$\frac{\Gamma \vdash exp: T}{\Gamma \vdash exp: T} [bodyExp]$$

$$\frac{\Gamma \vdash stm}{\Gamma \vdash stm: void} [bodyStm]$$

$$\frac{\Gamma \bullet [] \vdash dec: \Gamma'}{\Gamma \vdash dec: void} [bodyDec]$$

$$\frac{\Gamma \bullet [] \vdash dec: \Gamma' \quad \Gamma' \vdash stm: \Gamma''}{\Gamma \vdash dec \quad stm: void} [bodyDecStm]$$

$$\frac{\Gamma \bullet [] \vdash dec: \Gamma' \quad \Gamma' \vdash exp: T}{\Gamma \vdash dec \quad exp: T} [bodyDecExp]$$

$$\frac{\Gamma \vdash \{ \quad \}}{\Gamma \vdash \{ \quad \}: void} [bodyEmpty]$$

$$\frac{\Gamma \vdash Id: T \quad \Gamma \vdash exp: T' \quad T' = T}{\Gamma \vdash Id = exp: \Gamma'} [asgn]$$

Funzione che ritorna Γ' : se Id è init allora $\Gamma' = \Gamma$ (non cambia l'ambiente) altrimenti, id diventa init quindi torno Γ' (si modifica l'ambiente). Vedi par. 3.1.1.

$$\frac{\Gamma \vdash f: T1 \times \dots \times Tn \rightarrow T \quad (\Gamma \vdash ei: Ti') \quad i \in 1..n \quad (Ti = Ti') \quad i \in 1..n}{\Gamma \vdash f(e1, \dots, en): \Gamma} [funCallStm]$$

$$\frac{\Gamma \vdash f: T_1 \times \dots \times T_n \rightarrow T \quad (\Gamma \vdash e_i: T_i') \quad i \in 1..n \quad (T_i = T_i') \quad i \in 1..n}{\Gamma \vdash f(e_1, \dots, e_n); : T} [funCallExp]$$

$$\frac{\Gamma \vdash exp: bool \quad \Gamma \vdash branchStm: \Gamma' \quad \Gamma \vdash branchStm: \Gamma''}{\Gamma \vdash if \exp \{ branchStm \} else \{ branchStm \}: \Gamma'''} [ifStm]$$

Funzione che ritorna Γ''' : se $\Gamma' = \Gamma''$ allora $\Gamma''' = \Gamma'$

se $\Gamma' \neq \Gamma''$ allora

- se hanno variabili in comune init allora $\Gamma''' = \Gamma' \cap \Gamma''$
(l'intersezione prende le variabili in comune che sono init in entrambi i rami)
- se non hanno variabili int in comune allora $\Gamma''' = \Gamma$

$$\frac{\Gamma \vdash exp: bool \quad \Gamma \vdash branchStm: \Gamma'}{\Gamma \vdash if \exp \{ branchStm \} : \Gamma} [ifStmThen] \quad \frac{\Gamma \vdash stm: \Gamma'}{\Gamma \vdash stm: \Gamma'} [branchStm]$$

$$\frac{\Gamma \vdash exp: bool \quad \Gamma \vdash branchExp: T \quad \Gamma \vdash branchExp: T' \quad T = T'}{\Gamma \vdash if \exp \{ branchExp \} else \{ branchExp \}: T} [ifExp]$$

$$\frac{\Gamma \vdash stm: \Gamma' \quad \Gamma' \vdash exp: T}{\Gamma \vdash stm \exp: T} [branchExp] \quad \frac{\Gamma(Id): T}{\Gamma \vdash Id: T} [var]$$

$$\frac{}{\Gamma \vdash num: int} [num] \quad \frac{}{\Gamma \vdash true: bool} [true] \quad \frac{}{\Gamma \vdash false: bool} [false]$$

$$\frac{\Gamma \vdash exp: T}{\Gamma \vdash (' exp'): T} [param] \quad \frac{\Gamma \vdash d: \Gamma' \quad \Gamma' \vdash D: \Gamma''}{\Gamma \vdash dD: \Gamma''} [seqDec]$$

$$\frac{\Gamma \vdash s: \Gamma' \quad \Gamma' \vdash S: \Gamma''}{\Gamma \vdash sS: \Gamma''} [seqStm]$$

$$\frac{\Gamma \vdash \text{exp}:\text{bool}}{\Gamma \vdash !\text{exp}:\text{bool}} [\text{not}] \quad \frac{\Gamma \vdash e1:\text{int} \quad \Gamma \vdash e2:\text{int} \quad + \mid - \mid * \mid / \text{ int } x \text{ int } \rightarrow \text{int}}{\Gamma \vdash e1 (+ \mid - \mid * \mid /) e2:\text{int}} [\text{opAritm}]$$

$$\frac{\Gamma \vdash e1:\text{int} \quad \Gamma \vdash e2:\text{int} \quad > \mid < \mid \geq \mid \leq \text{ int } x \text{ int } \rightarrow \text{bool}}{\Gamma \vdash e1 (> \mid < \mid \geq \mid \leq) e2:\text{bool}} [\text{compar}]$$

$$\frac{\Gamma \vdash e1:T \quad \Gamma \vdash e2:T \quad T x T \rightarrow \text{bool}}{\Gamma \vdash e1 == e2:\text{bool}} [\text{equals}]$$

$$\frac{\Gamma \vdash e1:\text{bool} \quad \Gamma \vdash e2:\text{bool} \quad \text{bool } x \text{ bool } \rightarrow \text{bool}}{\Gamma \vdash e1(\&\& \mid ||) e2:\text{bool}} [\text{boolean}]$$

Codici testati:

```
1) int a; int b; int c ; c = 2 ;
    if (c > 1) { b = c ; } else { a = b ; }
    } // ERRORE perché b è usata ma non inizializzata
```

```
You had: 1 errors:
+++++++ Attenzione ++++++ Id b non inizializzata
```

```
2) 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) ; // ERRORE perchè x è usata ma non
    inizializzata (ramo else)
```

```
You had: 1 errors:
+++++++ Attenzione ++++++ Id x non inizializzata
```

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

```
h(5) ; // CORRETTA
```

4) int a;

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

You had: 1 errors:

+++++++ Attenzione ++++++++ Id x non inizializzata

4 *Generazione codice intermedio*

Il punto 4 della consegna chiede di estendere l'interprete di SimpLan, per implementare l'interprete per il nostro linguaggio.

In questa fase viene utilizzata la grammatica SVM.gr4, prima importandola e poi generando i file antlr.

A questo punto, per ogni classe Node, abbiamo implementato il metodo **codeGeneration()** che ci restituisce il codice generato tramite la visita dell'ast. Il codice generato lo salviamo in un file di output.

Il file di output viene utilizzato come input per la virtual machine che esegue il codice.

I codici testati sono i seguenti:

Per eseguire codici che non rispettano il punto bonus è stato aggiunto un flag nel main **initError** che se impostato a true blocca l'esecuzione del programma se ci sono errori di inizializzazione, altrimenti segnala l'errore e prosegue.

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

In questo caso il risultato è 0 (salvato nel registro AO) in quanto la funzione chiamata è di tipo void.

```

2) 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)

```

Risultato 720 (salvato nel registro AO)

```

3) int u;
   void f(int m, int n){
       if (m>n) { u = m+n; }
       else { int x ; x = 1 ; f(m+1,n+1); }
   }
   f(5,4;
   u

```

*** ERRORE linea 4 posizione 8: extraneous input 'int' expecting {'if', ID}**

L'errore si verifica nel ramo "then" in quanto vi è una dichiarazione di variabile, non consentita dalla grammatica.