

# **Report Progetto SimpLanPlus**

Complementi di Linguaggi di Programmazione

Michele Bianco - Nicolò Buscaroli

Unibo 2022/2023

## **Introduzione**

SLP (SimpLanPlus) è un linguaggio imperativo in cui vengono utilizzati i tipi intero, booleano e void e le dichiarazioni non includono anche l'inizializzazione; inoltre, le funzioni possono essere ricorsive ma non mutuamente ricorsive.

Come IDE è stato utilizzato IntelliJ IDEA Community Edition. Installata l'estensione di ANTLR per l'ambiente di sviluppo, il gruppo ha studiato la sintassi di SLP sperimentando con diversi programmi e osservando i vari alberi sintattici risultanti.

Il progetto del compilatore è stato suddiviso nei seguenti package:

- *ast*, contiene le implementazioni dei nodi dell'albero sintattico e del visitor;
- *evaluator*, contiene le classi necessarie per l'interprete;
- *main*, contiene il runner del progetto;
- *parser*, contiene il file della grammatica e le varie classi generate da ANTLR sulla base di essa;
- *utils*, contiene le classi per la gestione della Symbol Table, degli errori e delle labels;
- *tests*, contiene i file di test.

Nel file *Main.java*, nelle prime righe, viene implementata attraverso uno scanner la possibilità dell'utente di inserire il nome del file contenente il codice che si vuole analizzare, il quale deve essere presente su un file nella cartella *tests*.

## **Esercizio 1 - Analisi Lessicale**

È ANTLR ad occuparsi di generare il lexer ed il parser. Al generatore di lexer viene passato come parametro il file contenente il codice da analizzare; al generatore di parser, invece, l'insieme dei token generati dal lexer appena creato.

In seguito l'error listener di default per il lexer e parser vengono sostituiti con una versione estesa chiamata *SLPErrorHandler*, il quale inserisce in una lista ogni errore lessicale incontrato; questa lista di errori viene poi inserita attraverso la funzione *toLog()* all'interno del file di log all'interno della cartella tests.

All'esecuzione del primo comando del parser (illustrata nel prossimo capitolo), la lista verrà popolata e si procederà con l'esecuzione del programma solo se questa lista risulterà vuota.

### **Esempi - Errori Lessicali**

<b>Codice SLP</b>	<b>Output errore</b>
int a; void d_1; d_1 = a;	Character 7 in line 2 generate an error: no viable alternative at input 'voidd_1'
int 12x; 12x = 4;	Character 4 in line 1 generate an error: no viable alternative at input 'int12'
bool x; x= £;	Character 4 in line 2 generate an error: mismatched input ';' expecting {'(', 'if', 'true', 'false', '!', INTEGER, ID}

Nel primo esempio, viene dichiarata una variabile di tipo *void* di nome *d\_1*. Il simbolo *\_* non è 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, non viene riconosciuto l'identificatore dichiarato a causa del fatto che inizia con un numero: viene violata la grammatica.

Nel terzo esempio, viene sollevato un errore in quanto la grammatica non accetta il simbolo £: non è una delle possibili produzioni di *exp*.

## **Esercizio 2 - Analisi Semantica**

La symbol table è stata implementata come una lista di Hash Map. Ogni Hash Map corrisponde ad uno scope. La scelta è stata fatta tenendo conto che sia l'approccio scelto che quello scartato sono equivalenti, avendo entrambi vantaggi e svantaggi.

La classe *SymbolTable* è costituita da questi componenti principali:

- *ArrayList<HashMap<String, STEntry>> table*: lista di hashmap che rappresenta la Symbol Table;
- *ArrayList<Integer> offsetList*: lista degli offset per ogni scope;
- *SymbolTable()*: è il costruttore, crea una Symbol Table vuota;
- *SymbolTable(SymbolTable symbolTable)*: crea una copia della Symbol Table passata come parametro;
- *STEntry lookup(String id)*: cerca all'interno della Symbol Table la dichiarazione dell'identificatore passato come parametro (ritorna l'oggetto STEntry, se trovato, altrimenti null);
- *STEntry lookup(String id, int nestingLevel)*: versione ottimizzata del lookup precedente che cerca solo nello scope dato come parametro;
- *void add(String id, Type type)*: funzione utilizzata per inserire una nuova entry nella Symbol Table, nello scope più interno (è utilizzata per le variabili);
- *void add(String id, Type type, String label)*: uguale alla precedente, ma utilizzata per le funzioni;
- *int newScope()*: serve per entrare in uno nuovo scope (viene aggiunto un nuovo hashmap alla symbol table);
- *void exitScope()*: è l'operazione inversa della precedente;
- *ArrayList<String> getInitializedEntries()*: produce una lista di identificatori di entry inizializzate;
- *void increaseOffset()*: incrementa l'offset dello scope più interno.

La classe *STEntry* rappresenta le entry della symbol table. È costituita dai seguenti campi:

- *Type type*: tipo della entry;
- *int nestingLevel*: livello di profondità della symbol table nel quale si trova lo scope dove la entry è stata dichiarata;
- *String label*: label della funzione, serve per la Code Generation;

- *boolean initialized*: tiene traccia del fatto che la entry sia stata inizializzata oppure no (utilizzata nel punto opzionale);
- *boolean conditionInitializedWarning*: tiene traccia del fatto che una variabile sia stata inizializzata solo in uno dei due rami di un if (utilizzata nel punto opzionale);
- *int offset*: offset rispetto al frame pointer.

Nella classe Main viene istanziato il *Visitor* e si passa alla generazione dell'albero di sintassi astratto. Questa generazione avviene in seguito all'effettiva esecuzione del parser.

```
// Generate abstract tree
Node ast = visitor.visit(parser.prog());
```

Si noti che *prog* è la prima regola della grammatica, ovvero quella più generale: corrisponderà alla radice dell'albero.

*Visit()* è una funzione che viene eseguita su ogni Nodo dell'albero di sintassi astratto e si comporta in modo diverso a seconda del tipo di Nodo. In generale, il suo scopo è quello di istanziare nuovi tipi di nodo. Perciò, grazie alla riga di codice riportata sopra, l'ast viene generato.

Da notare che la grammatica data nella consegna è stata modificata: sono stati aggiunti i tag per ogni regola, che servono nel visitor a differenziare le visite. Ad esempio, all'interno del visitor, la funzione *visitSimpleProg()*

```
@Override
public Node visitSimpleProg(SimpLanPlusParser.SimpleProgContext ctx) {
    return new ProgSimpleNode(this.visit(ctx.exp()));
}
```

verrà utilizzata per la regola *prog*: *exp*, cioè la versione più semplice della regola *prog*, associata appunto al tag *SimpleProg*; viene creato il nodo corrispondente all'*exp* trovato nel codice. Tramite un processo a catena, tutti i nodi vengono implementati. Inoltre, nella grammatica sono state inserite delle label (ad esempio *left* e *right*) che servono per distinguere gli elementi dello stesso tipo, ad esempio, nella somma, l'*exp* di sinistra da quella di destra. Altre label che sono state introdotte sono *then* e *else* utilizzate all'interno del costrutto If per distinguere il ramo then ed il ramo else:

```
| 'if' '(' exp ')' then=ifBodyS ('else' else=ifBodyS)?
```

*ifBodyS* (ed il suo corrispettivo *ifBodyE* all'interno della regola per l'*exp*) serve per far sì che la label *then* (e quella *else*) si riferisca all'intera sequenza di *stm* presente all'interno di *IfBodyS* e non solo al suo primo elemento, cosa che accadrebbe in questo caso:

```
'if' '(' exp ')' then='{ (stm)+ }' ('else' else='{ (stm)+ }' )?
```

Se il programma è libero da errori viene generata la symbol table, la quale viene popolata all'esecuzione di *checkSemantic()* ricorsivamente su tutti i nodi dell'albero per trovare gli errori semantici; gli eventuali errori vengono stampati. Ogni *checkSemantic()* è implementato in modo diverso e specificatamente per il tipo di nodo che deve controllare.

### **Esercizio 3 - Type Checking**

Se dopo la chiamata a *checkSemantic()* non emergono errori semantici, si procede alla stampa dell'albero di sintassi astratto.

Successivamente si passa al Type Checking, che si occupa principalmente del controllo di tipo per assegnazioni, passaggio di parametri, valori di ritorno di funzioni, corrispondenza tra valore di ritorno di una funzione e valore ritornato da un'invocazione. Ricorsivamente viene chiamata la funzione *typeCheck()* su ogni nodo dell'albero astratto: se termina con successo, *typeCheck()* ritorna il tipo del programma oppure *Void* (se il programma non ha tipo); altrimenti, se sono stati rilevati degli errori, essi vengono stampati.

Sono state create le seguenti classi per rappresentare i vari nodi di tipi diversi:

- *BoolType*, per il tipo bool;
- *IntType*, per il tipo int;
- *VoidType*, per il tipo void;
- *FunType*, per il tipo della funzione (è costituito dalla lista dei tipi degli argomenti e dal tipo di ritorno);
- *ErrorType*, per gli errori, contiene una stringa che rappresenta il messaggio di errore.

Tutte queste classi estendono la classe *Type*, che a sua volta implementa la classe *Node*.

### **Regole di inferenza**

Valori:

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

$$\frac{}{\Gamma, n \vdash \text{true} : \text{bool}} \quad [\text{True}]$$

$$\frac{}{\Gamma, n \vdash \text{false} : \text{bool}} \quad [\text{False}]$$

$$\frac{\Gamma(id) = T}{\Gamma, n \vdash id : T} \quad [\text{Id}]$$

Operatori aritmetici:

$$\frac{\Gamma, n \vdash e_1 : T_1 \quad \Gamma, n \vdash e_2 : T_2 \quad T_1 = \text{int} = T_2 \quad \theta : \text{int} \times \text{int} \rightarrow \text{int}}{\Gamma, n \vdash e_1 \ \theta \ e_2 : \text{int}} \quad [\text{name}(\theta)]$$

*Dove name(θ) indica a quale operazione ci si sta riferendo (“+”, “-”, “\*”, “/”).*

Operatori relazionali:

$$\frac{\Gamma, n \vdash e_1 : T_1 \quad \Gamma, n \vdash e_2 : T_2 \quad T_1 = \text{int} = T_2 \quad \theta : \text{int} \times \text{int} \rightarrow \text{bool}}{\Gamma, n \vdash e_1 \ \theta \ e_2 : \text{bool}} \quad [\text{name}(\theta)]$$

*Dove name(θ) indica a quale operazione ci si sta riferendo (“>”, “<”, “>=”, “<=”).*

Operatori logici:

$$\frac{\Gamma, n \vdash e_1 : T_1 \quad \Gamma, n \vdash e_2 : T_2 \quad T_1 = \text{bool} = T_2 \quad \theta : \text{bool} \times \text{bool} \rightarrow \text{bool}}{\Gamma, n \vdash e_1 \ \theta \ e_2 : \text{bool}} \quad [\text{name}(\theta)]$$

*Dove name(θ) indica a quale operazione ci si sta riferendo (“or”, “and”).*

Comparazione:

$$\frac{\Gamma, n \vdash e_1 : T_1 \quad \Gamma, n \vdash e_2 : T_2 \quad T_1 = T_2 \quad == : T_1 \times T_1 \rightarrow \text{bool}}{\Gamma, n \vdash e_1 == e_2 : \text{bool}} \quad [\text{Comparison}]$$

Negazione:

$$\frac{\Gamma, n \vdash e : T \quad T = \text{bool} \quad ! : \text{bool} \rightarrow \text{bool}}{\Gamma, n \vdash !e : \text{bool}} \quad [\text{Not}]$$

Dichiarazione di variabili:

$$\frac{\text{id} \notin \text{dom}(\text{top}(\Gamma))}{\Gamma, n \vdash T \ \text{id}; : \Gamma[x \mapsto T], n + 1} \quad [\text{VarDeclaration}]$$

Assegnazione di variabile:

$$\frac{\Gamma, n \vdash \text{id} : T_1 \quad \Gamma, n \vdash e : T_2 \quad T_1 = T_2}{\Gamma, n \vdash \text{id} = e : \text{void}} \quad [\text{VarAssignment}]$$

Sequenza di dichiarazioni:

$$\frac{\Gamma, n \vdash d : \Gamma', n' \quad \Gamma', n' \vdash D : \Gamma'', n''}{\Gamma, n \vdash d D : \Gamma'', n''} \quad [\text{DecSequence}]$$

Sequenza di statement:

$$\frac{\Gamma, n \vdash s : T \quad \Gamma, n \vdash S : T' \quad T = \text{void} = T'}{\Gamma, n \vdash s S : \text{void}} \quad [\text{StmSequence}]$$

Dichiarazione di funzione:

$$\frac{\begin{array}{c} f \notin \text{dom}(\text{top}(\Gamma)) \\ \Gamma \cdot [f \mapsto (T_1, \dots, T_m) \rightarrow T, x_1 \mapsto T_1, \dots, x_m \mapsto T_m], m+1 \vdash \text{body} : T' \end{array} \quad T = T'}{\Gamma, n \vdash T f(T_1 \ x_1, \dots, T_i \ x_i) \{ \text{body} \} : \Gamma[f \mapsto (T_1, \dots, T_i) \rightarrow T], n} \quad [\text{FunDec}]$$

Invocazione di funzione:

$$\frac{\Gamma, n \vdash f : (T_1, \dots, T_m) \rightarrow T \quad (\Gamma \vdash e_i : T'_i)_{i=1}^m \quad (T_i = T'_i)_{i=1}^m}{\Gamma \vdash f(e_1, \dots, e_m) : T} \quad [\text{FunCall}]$$

Invocazione di funzione come statement:

$$\frac{\Gamma, n \vdash f : (T_1, \dots, T_m) \rightarrow T \quad (\Gamma \vdash e_i : T'_i)_{i=1}^m \quad (T_i = T'_i)_{i=1}^m}{\Gamma \vdash f(e_1, \dots, e_m); : \text{void}} \quad [\text{FunStm}]$$

If con espressione:

$$\frac{\Gamma, n \vdash e : T_1 \quad \Gamma, n \vdash \text{bodyE}_1 : T_2 \quad \Gamma, n \vdash \text{bodyE}_2 : T_3 \quad T_1 = \text{bool} \quad T_2 = T_3}{\Gamma, n \vdash \text{if } (e) \text{ bodyE}_1 \text{ else bodyE}_2 : T_2} \quad [\text{IfExp}]$$

*Dove bodyE è rappresentato come:*

$$\frac{\Gamma, n \vdash e : T_1 \quad \Gamma, n \vdash S : T_2 \quad T_2 = \text{void}}{\Gamma, n \vdash e : T_1} \quad [\text{IfBodyExp}]$$

If con solo sequenza di statement:

$$\frac{\Gamma, n \vdash e : T_1 \quad \Gamma, n \vdash S_1 : T_2 \quad \Gamma, n \vdash S_2 : T_3 \quad T_1 = \text{bool} \quad T_2 = \text{void} = T_3}{\Gamma, n \vdash \text{if } (e) \ S_1 \text{ else } S_2 : T_2} \quad [\text{IfStm}]$$



Programma completo:

$$\frac{\Gamma \bullet [], n \vdash D : \Gamma', n' \quad \Gamma', n' \vdash S : T_1 \quad \Gamma', n' \vdash e : T_2 \quad T_1 = \text{void}}{\Gamma, n \vdash D \ S \ e : T_2} \quad [\text{ProgComplex}]$$

Programma con semplice espressione:

$$\frac{\Gamma \bullet [], n \vdash e : T}{\Gamma, n \vdash e : T} \quad [\text{ProgSimple}]$$

## Punto opzionale

Come punto opzionale è richiesto il controllo dell'utilizzo di variabili non inizializzate (si assume che le funzioni non accedano mai a variabili globali).

All'interno di *IdNode* (astrazione della produzione di exp per gli identificatori), la funzione *checkSemantic()* è strutturata in modo da effettuare tale controllo:

```
@Override
public ArrayList<SemanticError> checkSemantics(SymbolTable symbolTable, int nestingLevel) {
    ArrayList<SemanticError> errors = new ArrayList<>();
    this.nestingUsage = nestingLevel;
    // Check for variable id in the symbol table
    entry = symbolTable.lookup(id);

    if (entry == null)
        errors.add(new SemanticError("Id " + id + " is not declared."));
    // Check if the variable is initialized
    else if (!entry.isInitialized())
        errors.add(new SemanticError("Id " + id + " used before initialization."));
    // Check if variable has been initialized only in one if branch.
    else if (entry.hasConditionWarning())
        errors.add(new SemanticError("Id " + id + " may not be initialized."));

    return errors;
}
```

Prima di tutto viene controllato che l'*Id* sia presente all'interno della Symbol Table (tramite un lookup); successivamente, si controlla che la variabile *isInitialized()* dell'entry sia *false*, in quel caso si genera l'errore opportuno. Si noti che è necessario gestire *isInitialized()* in modo che venga aggiornata opportunamente quando la variabile viene inizializzata. Il secondo ramo *else if* gestisce un caso particolare:

quando, all'interno di un costrutto *if then else*, una variabile viene inizializzata solo in uno dei due branch (*then* o *else*) ma non nella controparte, in questo caso un ulteriore flag di STEntry chiamato *conditionInitializedWarning* viene posto a *true*; se il valore di una variabile viene utilizzata con questo flag a *true*, viene generato un errore opportuno che indica che la variabile potrebbe non essere inizializzata.

```
// Check if then branch and else branch initialized the same variables
if (stInitialized.size() != estInitialized.size() || !stInitialized.containsAll(estInitialized)) {
    // Make a list of all unique id initialized in only one branch
    ArrayList<String> difference1 = new ArrayList<>(stInitialized);
    ArrayList<String> difference2 = new ArrayList<>(estInitialized);
    difference1.removeAll(estInitialized);
    difference2.removeAll(stInitialized);

    ArrayList<STEntry> difference = new ArrayList<>();

    for (String id : difference1)
        difference.add(symbolTable.lookup(id));

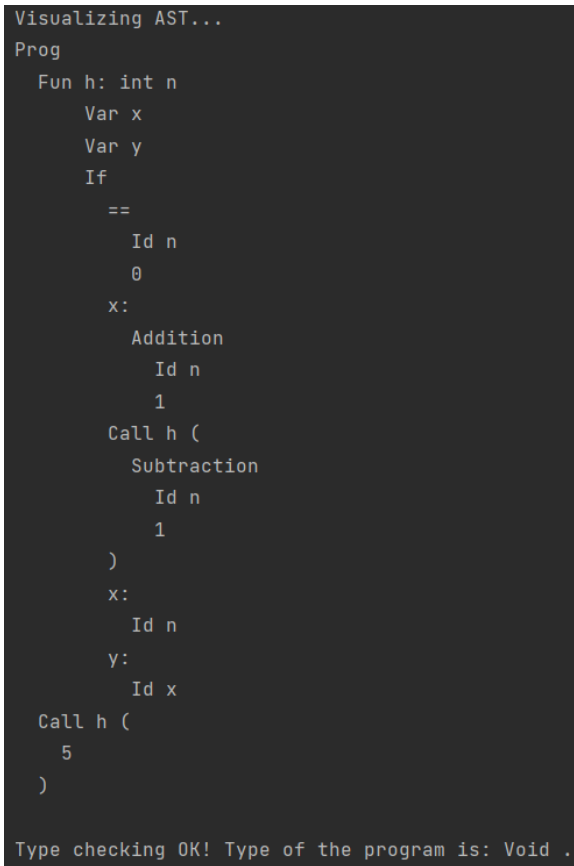
    for (String id : difference2)
        difference.add(symbolTable.lookup(id));

    // Mark entries with conditional warning
    for (STEntry entry : difference)
        entry.markConditionWarning();
}
```

In questo frammento di codice della classe *IfExpNode* si può vedere come la variabile *difference* sia una lista di tutte le entry relative alle variabili che sono state inizializzate nel ramo *then* ma non nel ramo *else* o viceversa (si noti che *stInitialized* e *estInitialized* sono rispettivamente la lista delle variabili inizializzate nel ramo *then* e nel ramo *else*); nelle ultime due righe la variabile *conditionInitializedWarning* viene settata a *true* in ciascuna entry dentro a *difference*.

## Esempi

Codice SLP	Output Errore
<pre>int a; int b; int c ; c = 2 ; if (c &gt; 1) { b = c ; } else { a = b ; } // ERRORE perché b è usata ma non inizializzata</pre>	<pre>You had 1 errors:       Id b used before initialization.</pre>

<pre>int a; int b; int c ; void f(int n){     int x ; int y ;     if (n &gt; 0) { x = n ;} else { y = n+x ;} } c = 1 ; f(0) ; // ERRORE perché x è usata ma non inizializzata (ramo else)</pre>	<p>You had 1 errors:</p> <p>Id x used before initialization.</p>
<pre>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</pre>	 <pre>Visualizing AST... Prog   Fun h: int n     Var x     Var y     If       ==         Id n         0       x:         Addition           Id n           1         Call h (           Subtraction             Id n             1           )       y:         Id n     y:       Id x     Call h (       5     ) Type checking OK! Type of the program is: Void .</pre>
<pre>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</pre>	<p>You had 1 errors:</p> <p>Id x used before initialization.</p>

## Esercizio 4 - Code Generation

La generazione delle label viene gestita dalla classe *CodeGenSupport* del package *utils*, che consente di creare e memorizzare nuove label assicurando l'univocità di esse grazie ad un contatore, il cui valore viene inserito all'interno della label che viene restituita.

La macchina virtuale utilizzata è la stessa del *SimpLan* di base mentre la *codeGeneration*, che ha lo scopo di generare la serie di istruzioni in bytecode per l'interprete di *SimpLanPlus*, è stata rivista per ogni nodo, in particolare per i seguenti:

### FunDecNode

```
public String codeGeneration() {
    CodeGenSupport.addFunctionCode(
        label + ": \n" +
        "pushr RA \n" +
        body.codeGeneration() +
        "popr RA \n" +
        "addi SP " + paramList.size() + " \n" +
        "pop \n" +
        "store FP 0(FP) \n" +
        "move FP AL \n" +
        "subi AL 1 \n" +
        "pop \n" +
        "rsub RA \n"
    );

    return "push " + label + " \n";
}
```

È molto simile alla code generation di *SimpLan*: l'unica differenza è che non sembra esserci il comando che fa spazio nello stack per tutte le dichiarazioni fatte all'interno del *body*. In realtà, nel nostro progetto tale comando è all'interno della code generation del *body*.

Si noti che la funzione *addFunctionCode()* serve per inserire il codice della funzione dichiarata all'interno del campo *functionsCode* della classe *CodeGenSupport*.

## GreatEqualNode

```
public String codeGeneration() {  
    String labelTrue = CodeGenSupport.newLabel();  
    String labelEnd = CodeGenSupport.newLabel();  
  
    return left.codeGeneration() +  
        "pushr A0 \n" +  
        right.codeGeneration() +  
        "popr T1 \n" +  
        "bleq A0 T1 " + labelTrue + "\n" +  
        "storei A0 0 \n" +  
        "b " + labelEnd + "\n" +  
        labelTrue + ": \n" +  
        "storei A0 1 \n" +  
        labelEnd + ": \n";  
}
```

Questo nodo è relativo all'operatore  $\geq$ . Esiste un'istruzione bytecode per controllare se due elementi sono minori o uguali (*bleq*) e, nel caso lo siano, far ripartire l'esecuzione dalla label indicata: questa istruzione viene utilizzata ma sui due elementi da confrontare scambiati di posizione, in modo da controllare se sono maggiori o uguali.

## GreaterNode

```
public String codeGeneration() {
    String labelFalse = CodeGenSupport.newLabel();
    String labelEnd = CodeGenSupport.newLabel();

    return left.codeGeneration() +
        "pushr A0 \n" +
        right.codeGeneration() +
        "popr T1 \n" +
        "bleq T1 A0 " + labelFalse + "\n" +
        "storei A0 1 \n" +
        "b " + labelEnd + "\n" +
        labelFalse + ": \n" +
        "storei A0 0 \n" +
        labelEnd + ": \n";
}
```

Se l'elemento di sinistra è minore o uguale a quello di destra allora si mette 0 nel registro *A0*; altrimenti si mette 1, in quanto l'elemento di sinistra sarà maggiore di quello di destra.

## MinEqualNode

```
public String codeGeneration() {
    String labelTrue = CodeGenSupport.newLabel();
    String labelEnd = CodeGenSupport.newLabel();

    return left.codeGeneration() +
        "pushr A0 \n" +
        right.codeGeneration() +
        "popr T1 \n" +
        "bleq T1 A0 " + labelTrue + "\n" +
        "storei A0 0 \n" +
        "b " + labelEnd + "\n" +
        labelTrue + ": \n" +
        "storei A0 1 \n" +
        labelEnd + ": \n";
}
```

## MinorNode

```
public String codeGeneration() {  
    String labelTrue = CodeGenSupport.newLabel();  
    String labelFalse = CodeGenSupport.newLabel();  
    String labelEnd = CodeGenSupport.newLabel();  
  
    return left.codeGeneration() +  
        "pushr A0 \n" +  
        right.codeGeneration() +  
        "popr T1 \n" +  
        // Check if left is <= to right  
        "bleq T1 A0 " + labelTrue + "\n" +  
        labelFalse + ": \n" +  
        "storei A0 0 \n" +  
        "b " + labelEnd + "\n" +  
        labelTrue + ": \n" +  
        // Check if right is equal to left  
        "beq A0 T1 " + labelFalse + "\n" +  
        "storei A0 1 \n" +  
        labelEnd + ": \n";  
}
```

Questo è il caso più complesso. Viene utilizzata un'altra istruzione bytecode, *beq*, che se i due elementi sono uguali fa ripartire l'esecuzione dalla label indicata. Se *left* non è minore o uguale a *right* si inserisce 0 in *A0* e si passa direttamente a *labelEnd*, terminando l'esecuzione. Se *left*  $\leq$  *right* allora si passa al controllo se *left* è uguale a *right*; in quel caso viene inserito 0 in *A0* e si passa alla *labelEnd* terminando l'esecuzione. Se invece *left* e *right* sono diversi si inserisce 1 in *A0* e termina l'esecuzione.

## NotNode

```
public String codeGeneration() {
    String labelEnd = CodeGenSupport.newLabel();
    String labelTrue = CodeGenSupport.newLabel();

    return exp.codeGeneration() +
        "storei T1 1 \n" +
        "beq A0 T1 " + labelTrue + "\n" +
        "storei A0 1 \n" +
        "b " + labelEnd + "\n" +
        labelTrue + ": \n" +
        "storei A0 0 \n" +
        labelEnd + ": \n";
}
```

Dopo aver fatto la code generation dell'espressione, si controlla se il valore salvato su *A0* è 1; in quel caso si passa a *labelTrue* e si inserisce in *A0* 0. Altrimenti si fa l'opposto.

## AndNode

```
public String codeGeneration() {
    String labelEnd = CodeGenSupport.newLabel();

    return left.codeGeneration() +
        "storei T1 0 \n" +
        "beq A0 T1 " + labelEnd + "\n" +
        right.codeGeneration() +
        labelEnd + ": \n";
}
```

Implementazione lazy del costrutto *and*: nel caso la prima espressione risulti falsa, la seconda non viene valutata.

Si fa la code generation di *left* e si confronta il suo esito salvato su *A0* con 0; se sono uguali si termina l'esecuzione, altrimenti si fa la code generation di *right*, inserendo in *A0* il risultato corretto dell'*And*.



## OrNode

```
public String codeGeneration() {
    String labelEnd = CodeGenSupport.newLabel();

    return left.codeGeneration() +
        "storei T1 1 \n" +
        // If true, A0 already contains 1
        "beq A0 T1 " + labelEnd + "\n" +
        right.codeGeneration() +
        labelEnd + ": \n";
}
```

Implementazione lazy del costrutto *or*: nel caso la prima espressione risulti vera, la seconda non viene valutata.

Se *left* è uguale ad 1, allora si termina l'esecuzione in quanto il risultato dell'*Or* è 1 e su *A0* è già inserito il valore corretto; altrimenti, si fa la code generation di *right* ed il valore salvato su *A0* sarà il risultato corretto dell'*Or*.

## VarDeclarationNode

```
public String codeGeneration() {
    // Leave the space for new variable
    return "subi SP 1 \n";
}
```

Questa istruzione serve per creare lo spazio nello stack per una nuova variabile.

## VarAssNode

```
public String codeGeneration() {
    return exp.codeGeneration() +
        "move AL T1 \n" +
        "store T1 0(T1) \n".repeat(Math.max(0, nestingUsage - entry.getNesting())) +
        "subi T1 " + entry.getOffset() + " \n" +
        "load A0 0(T1) \n";
}
```

Recupera la posizione della variabile dallo stack e ne altera il valore con quello ottenuto dall'espressione.

## ProgComplexNode

```
public String codeGeneration() {
    StringBuilder decStmCode = new StringBuilder();

    for (Node declaration : declarationList)
        decStmCode.append(declaration.codeGeneration());

    for (Node statement : statementList)
        decStmCode.append(statement.codeGeneration());

    if (exp != null)
        decStmCode.append(exp.codeGeneration());

    return "pushr FP \n" +
        "pushr AL \n" +
        decStmCode +
        "halt \n" +
        CodeGenSupport.getFunctionsCode();
}
```

Dopo aver inserito nella pila il Frame Pointer e l'Access Link, viene fatta la code generation di tutte le dichiarazioni e di tutti gli statement (e dell'eventuale *exp* finale). Dopo aver terminato l'esecuzione con *halt*, viene inserito il codice delle funzioni dichiarate presente all'interno del campo *functionsCode* della classe CodeGenSupport; in questo modo si evita di eseguirlo erroneamente.

## Esempio 1

### Codice SLP:

```
int x ;  
void f(int n){  
    if (n == 0) { n = 0 ; }    // n e` gia` uguale a 0; equivale a fare skip  
    else { x = x * n ; f(n-1) ; }  
}  
x = 1 ;  
f(10)
```

### Output:

```
You had 1 errors:  
  
Id x used before initialization.
```

All'interno della dichiarazione di *f()* viene utilizzata *x* prima dell'inizializzazione, perciò viene restituito un errore semantico dovuto all'implementazione del punto opzionale.

È stato però aggiunto uno switch nel *Main* che permette di silenziare gli errori e quindi proseguire nella generazione di codice per verificarne il risultato.

```
// Used to mute errors if needed  
boolean showError = true;
```

Dopo aver settato questo flag a *false*, eseguiamo nuovamente il programma:

Prima di tutto viene stampato l'ast:

```

Prog
  Var x
  Fun f: int n
    If
      ==
      Id n
      0
    n:
      0
    x:
      Multiplication
        Id x
        Id n
      Call f (
        Subtraction
          Id n
          1
        )
  x:|
  1
  Call f (
    10
  )

```

Poi il Type Checking ritorna il tipo Void.

Viene successivamente generato il codice intermedio e mostrato passo dopo passo lo stato dello stack.

Come risultato finale viene riportato il valore 0.

## Esempio 2

Codice SLP:

```
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)
```

Output:

L'ast è:

```
Prog
  Var u
  Fun f: int n
    Var y
    y:
      1
    If
      ==
      Id n
      0
      Id y
      y:
        Call f (
          Subtraction
            Id n
            1
          )
        Multiplication
          Id y
          Id n
    u:
      6
    Call f (
      Id u
    )
```

Il tipo del programma è Int; il risultato finale è 720.

### Esempio 3

Codice SLP:

```
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  
// cosa succede se la invoco con f(4,5)?
```

Output:

Viene sollevato un errore in quanto per la grammatica data non è possibile che ci sia una dichiarazione di variabile all'interno di un ramo *else*.

```
Character 8 in line 4 generate an error: extraneous input 'int' expecting {'}', 'if', ID}  
Character 14 in line 4 generate an error: no viable alternative at input 'x;'
```

### Esempio 3 - Bis

Si ripropone il codice dell'esempio 3 adattandolo alle istruzioni consentite dalla grammatica di SLP.

Codice SLP:

```
int u ;  
void f(int m, int n){  
    int x ; // dichiarazione spostata prima dell'if  
    if (m>n) { u = m+n ;}  
    else { x = 1 ; f(m+1,n+1) ; }  
}  
f(5,4) ; u  
// cosa succede se la invoco con f(4,5)?
```

Output:

Anche in questo caso viene stampato un errore dovuto all'implementazione del punto opzionale che segnala che una variabile è stata inizializzata solo in uno dei rami del costrutto *if*.

```
You had 1 errors:  
    Id u may not be initialized.
```

Dopo aver settato il flag *showError* a *false* come nell'esempio 1, eseguiamo nuovamente il programma:

```
Prog
  Var u
  Fun f: int m int n
    Var x
    If
      Greater
        Id m

        Id n
      u:
        Addition
          Id m
          Id n
      x:
        1
    Call f (
      Addition
        Id m
        1
      Addition
        Id n
        1
    )
  Call f (
    5
    4
  )
  Id u
```

Il tipo ritornato del programma è *Int* ed il risultato è 9.

Ora proviamo ad invocare *f(4,5)*: il risultato è un errore di memoria.

Infatti, il programma cicla all'infinito e lo stack supera i limiti consentiti dalla macchina virtuale, provando ad accedere a celle di memoria non allocate.