Progetto SimpLanPlus

Indice

1 Esercizio 1

Traccia: *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.*

Dalla documentazione ufficiale della libreria *ANTLR* è stato trovato un modo efficiente per intercettare gli errori lessicali senza l'ausilio di variabili all'interno della grammatica. La soluzione proposta consiste nell'implementazione del metodo *syntaxError()* dell'interfaccia *ANTLRErrorListener* che permette di catturare principalmente: gli errori sintattici generati dal lexer; il messaggio di errore; la linea di codice che contiene l'errore; la posizione del carattere errato. Per fare questo abbiamo creato la classe *ErrorsListener* che implementa l'interfaccia in questione e contiene il metodo *printErrorsToFile()* che, tramite la classe *FileWriter*, crea il file di output *syntactic_errors.txt* dove verranno inseriti la linea di codice che contiene l'errore, la posizione del carattere e il messaggio di errore. In aggiunta alla traccia dell'esercizio, gli errori sono stati stampati anche nella console.
All'interno della classe *TestSimpLanPlus* è stata creata l'istanza della classe *ErrorsListener*, dopodiché, tramite la funzione *removeErrorListeners()*, ci siamo assicurati che non ci fossero già dei *listener* in ascolto e abbiamo aggiunto la nostra istanza tramite il metodo *addErrorListener()*.

1.1 Esempi di funzionamento

Esempio 1

```
1    { int x = 5 }
```

In questo esempio, il lexer genera un errore perché non è presente il '*;*' alla fine della dichiarazione. All'interno del file di output è presente la seguente riga:

Line: 1, Char Position: 12, missing ';' at '}'

Esempio 2

```
{ int x = 5; if(x = 5){int y = 4;} }
```

Nel secondo esempio viene generato un errore nella condizione dell'if in quanto il lexer si aspetta un'espressione (vedi Figura 1) ma si ritrova un assegnamento.

```
ite         : 'if' '(' exp ')' statement ('else' statement)?;
exp         : '(' exp ')'                                          #baseExp
            | left=exp op=('*' | '/')              right=exp       #binExp
            | left=exp op=('+' | '-')              right=exp       #binExp
            | left=exp op=('<' | '<=' | '>' | '>=') right=exp       #binExp
            | left=exp op=('==' | '!=')            right=exp       #binExp
```

Figura 1: Grammatica if-then-else

L'errore stampato è il seguente:

```
Line: 1, Char Position: 18, mismatched input '='
expecting {')', '-', '*', '/', '+', '<', '<=', '>', '>=', '==', '!=', '&&', '||'}
```

Esempio 3

```
bool example3(){
    int p = 5;
}
```

In quest'ultimo esempio l'errore viene generato dall'assenza delle parentesi graffe iniziali, come viene specificato dal non terminale *block* (Figura 2):

```
block       : '{' declaration* statement* '}';
```

Figura 2: Grammatica block

Gli errori risultanti sono:

```
Line: 1, Char Position: 0, missing '{' at 'bool'
Line: 3, Char Position: 1, extraneous input '<EOF>'
expecting {'{', '}', 'void', 'int', 'bool', 'print', 'return', 'if', ID}
```

2 Esercizio 2

Traccia: *Sviluppare la tabella dei simboli del programma. Decidere se implementarlo come lista di hash-table o come hash-table di liste. Il codice sviluppato deve controllare:*

- *variabili/funzioni non dichiarate;*
- *variabili/funzioni dichiarate più volte nello stesso ambiente.*

Dopo aver verificato l'assenza di errori sintattici si è passati al controllo di quelli semantici. Abbiamo deciso di implementare la tabella dei simboli del programma come lista di *hash-table*. Per fare questo è stata creata la classe *SymbolTable* dove sono implementati i metodi:

- **addToST()**: aggiunge alla *symbol table* un nuovo scope.
- **lookup()**: controlla se l'ID passato come parametro è già presente nella *symbol table*.
- **exitScope()**: cancella l'ultimo ambiente creato nella *symbol table* e decrementa il livello di profondità.

Per poter catturare gli errori semantici all'interno del codice è necessario visitare l'albero di sintassi astratta, che viene generato estendendo la classe di *ANTLR SimpLanPlusBaseVisitor*, partendo dal non terminale iniziale *block*.
Per ogni possibile produzione è stata creata una classe che implementa l'interfaccia *Node*, contenente il metodo *checkSemantics()* che ritorna una lista di errori semantici sul nodo attuale. Così facendo:

- quando viene usata una variabile/funzione controlliamo se questa è presente nella *symbol table*, restituendo un errore in caso negativo;
- quando viene dichiarata una variabile/funzione controlliamo nella *symbol table* se questa è già presente al livello di profondità attuale, restituendo un errore in caso affermativo.

2.1 Esempi di funzionamento

```
1    { int x = 4;
2    y = 2; }
```

Listing 1: Variabile non dichiarata

Errore: *Variable id y never declared*

```
1    { int p(){int x = 4;}
2    q(); }
```

Listing 2: Funzione non dichiarata

Errore: *Function id q never declared*

```
1    { int x;
2    int x = 4; }
```

Listing 3: Variabile dichiarata più volte nello stesso ambiente

Errore: *Variable id x already declared*

```
1    { int p(){int x = 5;}
2    int p(){int y = 4;} }
```

Listing 4: Funzione dichiarata più volte nello stesso ambiente

Errore: *Function id p already declared*

2.2 Funzioni aggiuntive

Stampa su File: Tramite il metodo descritto nella sezione precedente viene creato il file *semantic_errors.txt* contenente gli errori semantici restituiti.

***toPrint()*:** tramite questo metodo, presente nell'interfaccia *Node*, viene stampato l'albero di sintassi astratta nella console.

3 Modifiche rispetto alla prima consegna

Rispetto alla prima consegna sono stati differenziati gli errori (Sezione 1) dai *warning*, sono state modificati alcuni aspetti della *checkSemantics()* (Sezione 2) ed è stato aggiunto l'*offset* nella *symbol table* per la gestione delle variabili nella generazione di codice (Sezione 5).

3.1 Warning ed errori

Rispetto alla Sezione 1 sono stati differenziati gli errori dai *warning*. Con *warning* intendiamo quegli errori che non causano un problema per il compilatore ma che avvertono sul fatto che una variabile o una funzione, anche se dichiarate, non sono mai utilizzate, quindi sarebbe meglio toglierle. Questi vengono stampati in *console* con il colore giallo.
Gli errori, invece, rappresentano un problema per il programma e quindi è necessario interrompere l'esecuzione. Questi vengono stampati in *console* con il colore rosso.

3.2 Offset

L'offset viene utilizzato nella generazione di codice per prendere la posizione corretta delle variabili rispetto al *frame pointer*. L'offset, definito nella *symbol table*, è inizializzato a 0 ogni qual volta si apre un nuovo ambiente e quando si esce si ripristina il precedente.
Ad ogni variabile inizializzata viene assegnato l'offset corrente e successivamente questo viene incrementato per poter essere assegnato ad una nuova variabile.

4 Esercizio 3

Traccia: *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;*
- *Dichiarazione di variabili non utilizzate.*

Il report deve contenere TUTTE le regole semantiche utilizzate e relativa discussione. Si faccia attenzione all'aliasing.

Dopo aver controllato che non ci siano errori sintattici e semantici, si è passati all'analisi dei tipi. Quest'ultima viene effettuata tramite il metodo *typeCheck()* presente all'interno dell'interfaccia *Node*.

4.1 Analisi dei Tipi

Nella grammatica SimpLanPlus sono presenti i tipi:

- *int* e *bool*: per le variabili e per la definizione di funzioni.
- *void*: solamente per la definizione di funzioni.

Nella nostra implementazione sono stati utilizzati due tipi aggiuntivi:

- *ArrowTypeNode*: per il typeCheck() di *DecFunNode* e *CallNode*.
- *RefTypeNode*: utilizzato per le variabili passate per riferimento.

4.1.1 Type Check dei Nodi

Di seguito vengono riportate le regole semantiche relative ai tipi e la relativa discussione.

- **BlockNode**:

$$\frac{\Gamma \vdash \text{declaration}^* : \Gamma' \quad \Gamma' \vdash \text{statement}^* : \Gamma''}{\Gamma \vdash \ \{\text{declaration}^* \ \text{statement}^*\} : \text{VoidTypeNode}}$$

  Partendo dall'ambiente Γ viene effettuato il *typeCheck()* di ogni dichiarazione generando un ambiente Γ'. Questo viene usato per il *typeCheck()* di ciascun *statement* che genera un ambiente Γ''. Il *typeCheck()* di ogni BlockNode ritorna un *VoidTypeNode*.

- **VarNode**:

$$\frac{\Gamma(\text{id}) \ = \ \text{T}}{\Gamma \vdash \ \text{id} : \text{T}}$$

  Dall'ambiente Γ si prende il tipo di *id* e si ritorna.

- **Operatori +, -, *, /**:

$$(+, \ -, \ *, \ /) : \ \text{NumTypeNode} \ \text{x} \ \text{NumTypeNode} \ \rightarrow \ \text{NumTypeNode}$$

$$\frac{\Gamma \vdash \text{leftExp} : \text{NumTypeNode} \quad \Gamma \vdash \text{rightExp} : \text{NumTypeNode}}{\Gamma \vdash \text{leftExp} \ (+, \ -, \ *, \ /) \ \text{rightExp} : \text{NumTypeNode}}$$

  Si controlla che il tipo dei due operandi sia un *NumTypeNode* e si ritorna un *NumTypeNode*.

- **Operatori** $\leq, \ \geq, \ >, \ <$:

$$(\leq, \geq, >, <) : \ \text{NumTypeNode} \ \text{x} \ \text{NumTypeNode} \ \rightarrow \ \text{BoolTypeNode}$$

$$\frac{\Gamma \vdash \text{leftExp} : \text{NumTypeNode} \quad \Gamma \vdash \text{rightExp} : \text{NumTypeNode}}{\Gamma \vdash \text{leftExp} \ (\leq, \ \geq, \ >, \ <) \ \text{rightExp} : \text{BoolTypeNode}}$$

  Si controlla che il tipo dei due operandi sia un *NumTypeNode* e si ritorna un *BoolTypeNode*.

- **Operatori Booleani (&&, ||)**:

$$(\&\&,\ \|) :\ \text{BoolTypeNode}\ \ \text{x}\ \ \text{BoolTypeNode}\ \rightarrow\ \text{BoolTypeNode}$$

$$\frac{\Gamma \vdash \text{leftExp} : \text{BoolTypeNode} \quad \Gamma \vdash \text{rightExp} : \text{BoolTypeNode}}{\Gamma \vdash \text{leftExp}\ (\&\&,\ \|)\ \text{rightExp} : \text{BoolTypeNode}}$$

Si controlla che il tipo dei due operandi sia un *BoolTypeNode* e si ritorna un *BoolTypeNode*.

- **Operatori == e !=**:

$$\frac{\Gamma \vdash \text{leftExp} : T_1 \quad \Gamma \vdash \text{rightExp} : T_2 \quad T_1 = T_2}{\Gamma \vdash \text{leftExp}\ (==,\ !=)\ \text{rightExp} : \text{BoolTypeNode}}$$

Si controlla che il tipo dei due operandi sia lo stesso e si ritorna un *BoolTypeNode*.

- **NegExpNode**:

$$\frac{\Gamma \vdash \text{leftExp} : T_1 \quad T_1 = \text{NumTypeNode}}{\Gamma \vdash\ \text{-}\ \text{exp} : \text{NumTypeNode}}$$

Si controlla che il tipo dell'espressione sia un *NumTypeNode* e si ritorna un *NumTypeNode*.

- **NotExpNode**:

$$\frac{\Gamma \vdash \text{leftExp} : T_1 \quad T_1 = \text{BoolTypeNode}}{\Gamma \vdash\ !\ \text{exp} : \text{BoolTypeNode}}$$

Si controlla che il tipo dell'espressione sia un *BoolTypeNode* e si ritorna un *BoolTypeNode*.

- **NumExpNode**:

$$\frac{}{\Gamma \vdash \text{value} : \text{NumTypeNode}}$$

Si ritorna il tipo *NumTypeNode*.

- **BoolExpNode**:

$$\frac{}{\Gamma \vdash \text{value} : \text{BoolTypeNode}}$$

Si ritorna il tipo *BoolTypeNode*.

- **DecVarNode**:

$$\frac{\text{id} \notin \text{dom}\{\text{top}\{\Gamma\}\} \quad \Gamma \vdash \text{exp} : \text{T} \quad \text{T} = \text{type}}{\Gamma \vdash \text{type id} = \text{exp}; : \text{VoidTypeNode}}$$

Partendo dall'ambiente Γ si controlla che l'*id* non sia presente nell'ambiente di testa della *symbol table*. Si fa il *typeCheck()* dell'espressione, si controlla che il tipo di *id* e dell'espressione sia uguale e si ritorna VoidTypeNode.

- **AssignmentNode**:

$$\frac{\dfrac{\text{id} \in \text{dom}\{\Gamma\}}{\Gamma(\text{id}) = \text{T}_1} \quad \Gamma \vdash \text{exp} : \text{T}_2 \quad \text{T}_1 = \text{T}_2}{\Gamma \vdash \text{id} = \text{exp}; : \text{VoidTypeNode}}$$

Partendo dall'ambiente Γ si controlla che l'*id* sia presente nella *symbol table*, si verifica che il tipo di *id* e dell'espressione sia uguale e si ritorna VoidTypeNode.

- **IteNode**:

$$\frac{\Gamma \vdash \text{cond} : \text{T} \quad \Gamma \vdash \text{tBranch} : \text{T}_1 \quad \text{T} = \text{BoolTypeNode}}{\Gamma \vdash \text{if(cond) tBranch} : \text{VoidTypeNode}}$$

Quando *elseBranch* non è presente, si controlla che il tipo della condizione sia BoolTypeNode, si fa il *typeCheck()* di *thenBranch* e si ritorna VoidTypeNode.

$$\frac{\Gamma \vdash \text{cond} : \text{T} \quad \Gamma \vdash \text{tBranch} : \text{T}_1 \quad \Gamma \vdash \text{eBranch} : \text{T}_2 \quad \text{T} = \text{BoolTypeNode} \quad \text{T}_1 = \text{T}_2}{\Gamma \vdash \ \text{if(cond)} \ \ \text{tBranch} \ \ \text{else} \ \ \text{eBranch} : \text{VoidTypeNode}}$$

Quando *elseBranch* è presente, si controlla che il tipo della condizione sia BoolTypeNode, si controlla che il tipo di *thenBranch* e di *elseBranch* sia uguale e si ritorna VoidTypeNode.

- **PrintNode**:

$$\frac{\Gamma \vdash \text{exp} : \text{T} \quad \text{T} \neq \text{VoidTypeNode}}{\Gamma \vdash \ \text{print(exp);} : \text{VoidTypeNode}}$$

Partendo dall'ambiente Γ, si controlla che il tipo dell'espressione non sia un VoidTypeNode e si ritorna VoidTypeNode.

- **RetNode**:

$$\frac{}{\Gamma \vdash \ \text{return;} : \text{VoidTypeNode}}$$

Quando l'espressione non è presente si ritorna semplicemente VoidTypeNode.

$$\frac{\Gamma \vdash \text{exp} : \text{T}}{\Gamma \vdash \ \text{return(exp);} : \text{T}}$$

Quando l'espressione è presente si esegue il *typeCheck()* dell'espressione e si ritorna il suo tipo.

- **DecFunNode**:

$$\frac{\text{fun} \notin \text{dom}\{\text{top}\{\Gamma\}\} \quad \Gamma[\text{fun} \ \mapsto \ \text{T}] \vdash \text{funBody} : \text{T}^1 \quad \text{T}^1 = \text{T}}{\Gamma \vdash \ \text{T fun() \{ funBody \}} : \text{VoidTypeNode}}$$

Quando non sono presenti argomenti, partendo da Γ si controlla che la funzione non sia già stata definita nell'ambiente di testa della *symbol table*.

Poi si aggiunge all'ambiente Γ la definizione della funzione e si fa il *typeCheck()* del body. Infine, si controlla che il tipo del body sia equivalente al tipo della funzione e si ritorna VoidTypeNode.

$$\frac{\text{fun} \notin \text{dom}\{\text{top}\{\Gamma\}\} \quad \Gamma[\text{fun} \ \mapsto \ \text{T}_1 \ \text{x}_1, ..., \text{T}_\text{n} \ \text{x}_\text{n} \ \rightarrow \ \text{T}] \vdash \text{funBody} : \text{T}^1 \quad \text{T}^1 = \text{T}}{\Gamma \vdash \ \text{T fun}(\text{T}_1 \ \text{x}_1, ..., \text{T}_\text{n} \ \text{x}_\text{n}) \ \{ \ \text{funBody} \ \} : \text{VoidTypeNode}}$$

Se, invece, sono presenti argomenti la regola è equivalente a quella precedente con l'aggiunta di questi all'ambiente che serve per fare il *typeCheck()* del *body*.

- **CallNode**:

$$\frac{\frac{\text{fun} \ \in \ \text{dom}\{\Gamma\}}{\Gamma \ \vdash \ \text{fun} : \ \text{T}}}{\Gamma \vdash \ \text{fun}() : \text{T}}$$

Se non sono presenti argomenti, bisogna controllare che la funzione sia definita nell'ambiente. Poi si fa il *typeCheck()* della funzione e si ritorna il suo tipo.

$$\frac{\frac{\text{fun} \ \in \ \text{dom}\{\Gamma\}}{\Gamma \ \vdash \ \text{fun} : \ \text{T}^1 \ \text{x} \ ... \ \text{x} \ \text{T}^\text{n} \ \rightarrow \ \text{T}} \quad (\Gamma \vdash \text{e}_\text{i} : \text{T}^1_\text{i})^{\text{i} \in 1...\text{n}} \quad (\text{T}_\text{i} = \text{T}^1_\text{i})^{\text{i} \in 1...\text{n}}}{\Gamma \vdash \ \text{fun}(\text{e}_1, ..., \text{e}_\text{n}) : \text{T}}$$

Se, invece, sono presenti argomenti bisogna, in aggiunta, controllare che il tipo di ogni parametro attuale sia uguale al corrispondente parametro formale.

4.2 Analisi degli Effetti

Alla creazione, ciascuna variabile viene impostata allo stato *bottom* (bot) che la identifica come definita ma non ancora inizializzata. In seguito, quando viene

inizializzata, passa dallo stato *bottom* allo stato *read-write* (rw).
La dichiarazione di questi stati è definita all'interno della classe *Effect*, situata nel package *mainPackage/SimpLanPlus/utils/symbol_table*. Oltre agli stati, all'interno della classe sono definiti anche i metodi *max*, *seq* ($\triangleright$) e *par* ($\otimes$) che verranno utilizzati per il cambiamento di stato, come si può vedere dalle regole nella Sezione 4.2.1.

4.2.1 Effect Check

Di seguito sono riportate le regole semantiche relative agli effetti e la relativa discussione.

- **BlockNode**:

$$\frac{\Sigma \bullet [] \vdash \text{declaration}^* : \Sigma' \qquad \Sigma' \vdash \text{statement}^* : \Sigma''}{\Sigma \vdash \{\text{declaration}^* \ \text{statement}^*\} : \Sigma''}$$

  Partendo dall'ambiente Σ viene effettuato il *checkEffect()* di ogni dichiarazione generando un ambiente Σ'. Questo viene usato per il *checkEffect()* di ciascun *statement* che genera un ambiente Σ'' che risulterà essere l'ambiente di ritorno.

- **VarNode**:

$$\frac{}{\Sigma \vdash \ \text{id} : \Sigma \triangleright [\text{id} \mapsto \text{rw}]}$$

  Nell'ambiente Σ si imposta l'effetto *rw* (*read-write*) a *id*.

- **ExpNodes**

$$\frac{\text{ids(e)} = \{\text{x}_1, ..., \text{x}_\text{n}\}}{\Sigma \vdash \ \text{e} : \Sigma \triangleright [\text{x}_1 \mapsto \text{rw}, ..., \text{x}_\text{n} \mapsto \text{rw}]}$$

  Si crea un ambiente vuoto e si imposta l'effetto di ogni *id* nell'espressione a *rw*. Poi si esegue la funzione *seq* fra Σ e l'ultimo ambiente creato.

- **DecVarNode**:

$$\frac{}{\Sigma \vdash \ \text{type id} : \Sigma[\text{id} \mapsto \text{bot}]}$$

Se non è presente l'espressione si imposta semplicemente la variabile *id* a *bot* (*bottom*) nell'ambiente Σ.

$$\frac{\Sigma \vdash \text{exp} : \Sigma'}{\Sigma \vdash \ \text{type id} \ = \ \text{exp}; : \Sigma' \triangleright [\text{id} \mapsto \text{rw}]}$$

Se è presente l'espressione si crea un ambiente vuoto e si imposta *id* a *rw*. Poi, partendo da Σ, si fa il *checkEffect()* dell'espressione generando così Σ'. In ultimo si esegue la funzione *seq* fra Σ' e il nuovo ambiente creato.

- **AssignmentNode**:

$$\frac{\Sigma \vdash \text{exp} : \Sigma'}{\Sigma \vdash \ \text{id} \ = \ \text{exp}; : \Sigma' \triangleright [\text{id} \mapsto \text{rw}]}$$

Si crea un ambiente vuoto e si imposta *id* a *rw*. Poi, partendo da Σ, si fa il *checkEffect()* dell'espressione generando così Σ'. In ultimo si esegue la funzione *seq* fra Σ' e il nuovo ambiente creato.

- **IteNode**:

$$\frac{\Sigma \vdash \text{cond} : \Sigma' \quad \Sigma' \vdash \text{tBranch} : \Sigma''}{\Sigma \vdash \ \text{if(cond)} \ \ \text{tBranch} : \Sigma''}$$

Se non è presente il ramo *else*, partendo dall'ambiente Σ viene effettuato il *checkEffect()* della condizione generando un ambiente Σ'. Questo viene usato per il *checkEffect()* del branch *then* che genera un ambiente Σ'' che sarà poi ritornato.

$$\frac{\Sigma \vdash \text{cond} : \Sigma' \quad \Sigma' \vdash \text{tBranch} : \Sigma_1 \quad \Sigma' \vdash \text{eBranch} : \Sigma_2}{\Sigma \vdash \ \text{if(cond)} \ \ \text{tBranch else} \ \ \text{eBranch} : \max(\Sigma_1, \Sigma_2)}$$

Se invece il ramo *else* è presente, partendo dall'ambiente Σ viene effettuato il *checkEffect()* della condizione generando un ambiente Σ'. Da quest'ultimo si generano due ambienti: Σ_1 per il ramo *then* e Σ_2 per il ramo *else*. In ultimo si esegue la funzione *max* fra questi due ambienti.

- **DecFunNode**:

$$\frac{\Sigma_0 = [\mathrm{x_1} \mapsto \mathrm{bot}, ..., \mathrm{x_m} \mapsto \mathrm{bot}, \mathrm{y_1} \mapsto \mathrm{bot}, ..., \mathrm{y_n} \mapsto \mathrm{bot}] \quad \Sigma|_{\mathrm{FUN}} \bullet \Sigma_0[\mathrm{f} \mapsto \Sigma_0 \to \Sigma_1] \vdash \ \mathrm{body} : \Sigma|_{\mathrm{FUN}} \bullet \Sigma_1[\mathrm{f} \mapsto \Sigma_0 \to \Sigma_1]}{\Sigma \vdash \ \mathrm{f}(\mathrm{var\ T_1\ x_1}, ..., \mathrm{var\ T_m\ x_m}, \mathrm{T'_1\ y_1}, ..., \mathrm{T'_n\ y_n})\ \mathrm{body} : \Sigma[\mathrm{f} \mapsto \Sigma_0 \to \Sigma_1]}$$

Partendo dall'ambiente Σ si crea un nuovo ambiente con l'inizializzazione di tutti i parametri a *bot*. Poi si esegue il *checkEffect()* del body.

- **CallNode**:

$$\frac{\Gamma \vdash \mathrm{f} : \&\mathrm{T_1\ x\ ...\ x\ \&T_m\ x\ T'_1\ x\ ...\ x\ T'_n} \to \mathrm{T} \quad \Sigma(\mathrm{f}) = \Sigma_0 \to \Sigma_1 \quad (\Sigma_1(\mathrm{y_i}) \leq \mathrm{del})^{1 \leq \mathrm{i} \leq \mathrm{n}} \quad \Sigma' = \Sigma[(\mathrm{z_i} \mapsto \Sigma(\mathrm{z_i}) \triangleright \mathrm{rw})^{\mathrm{z_i} \in \mathrm{var}(\mathrm{e_1}, ..., \mathrm{e_n})}] \quad \Sigma'' = \otimes_{\mathrm{i} \in 1...\mathrm{m}}[\mathrm{u_i} \mapsto \Sigma(\mathrm{u_i}) \triangleright \Sigma_1(\mathrm{x_i})]}{\Sigma \vdash \ \mathrm{f}(\mathrm{u_1}, ..., \mathrm{u_m}, \mathrm{e_1}, ..., \mathrm{e_n}) : \mathrm{update}(\Sigma', \Sigma'')}$$

Partendo dall'ambiente Γ si prende la definizione della funzione con i parametri formali e il tipo di ritorno. Successivamente, utilizzando l'ambiente Σ di *f*, si controlla che l'effetto di tutti i parametri formali passati per valore sia $\leq$ dell'effetto *del*. Verificato questo, si creano due nuovi ambienti:

  - Σ': uguale all'ambiente Σ eseguendo il comando *seq* fra ogni parametro attuali passati per valore e *rw*;
  - Σ'': si esegue il comando *par* per ogni parametro formale passato per riferimento.

Infine, tra questi due ambienti si esegue la funzione *update*.

4.3 Codici da verificare/discutere

Esempio 1

```
{ int a; int b; int c = 1 ;
  if (c > 1) { b = c ; } else { a = b ; } }
```

In questo codice il compilatore stampa il warning "Id *a* initialized but not used" perché nel ramo *else* la variabile *a* viene inizializzata ma mai usata. Successivamente, viene restituito l'errore *You cannot load null values* in quanto nonostante *b* nell'ambiente del ramo *then* viene inizializzata, quando viene eseguito il *checkEffect()* del ramo *else* esso non tiene in considerazione l'ambiente del ramo *then*.

Esempio 2

```
{ int a; int b; int c ;
  void f(int n, var int x){
    x = n ;
  }
  f(1,a) ; f(2,b) ; f(3,c) ;
}
```

Vengono restituiti 3 errori:

- Variable *a* used but not initialized
- Variable *b* used but not initialized
- Variable *b* used but not initialized

Ciascuna variabile viene impostata a Read-Write quando vengono passate come parametri della funzione *f*, ma non essendo prima inizializzate causano errore.

Esempio 3

```
{ int a; int b; int c = 1 ;
  void h(int n, var int x, var int y, var int z){
    if (n==0) return ; else { x = y ; h(n-1,y,z,x) ;}
  }
  h(5,a,b,c);
}
```

Come avviene nell'esempio precedente, le variabile a e b vengono usate nella chiamata della funzione h ma non sono state inizializzate prima. Gli errori sono:

- Variable a used but not initialized
- Variable b used but not initialized

5 Esercizio 4

Traccia: *Definire un linguaggio bytecode per eseguire programmi in SimpLanPlus e implementare l'interprete. In particolare:*

- *Il bytecode deve avere istruzioni per una macchina a pila che memorizza in un apposito registro il valore dell'ultima istruzione calcolata;*
- *Implementare l'interprete per il bytecode;*
- *Compilare ed eseguire i programmi del linguaggio ad alto livello.*

Per prima cosa è stato implementato il metodo *codeGeneration()* dell'interfaccia *Node*, all'interno del package *mainPackage/SimpLanPlus/ast/nodes*, per ciascun nodo definito. Questo metodo genera il codice nel linguaggio *SVM*. Dopo aver generato il codice dell'input, questo verrà scritto all'interno del file *generatedCode.txt*. Tale codice sarà l'input dell'interprete che lo eseguirà.

5.1 Generazione del codice dei nodi

Di seguito viene riportata la spiegazione della *codeGeneration()* di ciascun nodo.

5.1.1 BlockNode

Ogni qual volta viene aperto un blocco, all'interno della pila viene generato un record di attivazione come quello mostrato in Figura 3. Per prima cosa bisogna inserire il vecchio *frame pointer* sullo *stack* e aumentare lo *stack pointer* di una unità per ogni dichiarazione di variabili.
Si è dovuto distinguere tra il blocco principale e i vari blocchi aperti successivamente, in quanto il metodo *codeGeneration()* è lo stesso per ciascun blocco. Infatti, per il Block iniziale a causa della mancanza dell'*Access Link*, che verrà inizializzato nei successivi record di attivazione, viene al suo posto inserito lo *Stack Pointer*.

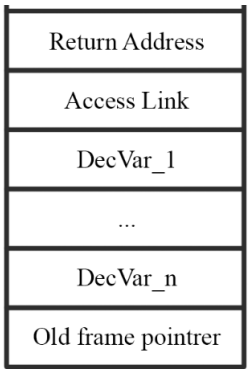

Figura 3: Record di attivazione di BlockNode

Dopo aver costruito questa prima parte di record, viene invocata la *codeGeneration()* per ogni dichiarazione e per ogni statement.
Dopodiché, nel caso del *Block* iniziale viene inserita l'istruzione *"halt"*, mentre nei successivi:

- viene fatto il *pop* delle varie dichiarazioni, dell'*access link* e del *return address*;
- viene ripristinato il vecchio *frame pointer* e tolto dalla pila;

Arrivati a questo punto viene invocato il metodo *codeGeneration()* della dichiarazione di ciascuna funzione.

5.1.2 VarNode

Questa sezione fa riferimento alla generazione di codice degli identificatori (*ID*).
In questo file, oltre all'implementazione del metodo *codeGeneration()* è stato creato il metodo *codeGenerationForRefType(boolean isFirstGeneration)*.
Nel primo metodo si risale la catena tramite l'*access link*, si prende l'*offset* dell'identificatore dalla *symbol table* e tramite il comando *lw $a0 offset($al)* viene caricato in *$a0* il valore corretto.
Il secondo metodo, invece, viene usato per la generazione del codice di variabili passate per riferimento, dove non va effettuato il *push* del valore della variabile

ma del suo indirizzo. Dopo aver risalito la catena e preso il giusto *offset*, si controlla, tramite il parametro *isFirstGeneration*, se siamo nella prima chiamata o in una ricorsiva. Nel primo caso viene inserito in *$a0* la somma tra *$al* e *offset*, che indica l'indirizzo della variabile. Nel secondo caso, invece, viene eseguita l'istruzione *lw $a0 offset($al)*. Quest'ultimo metodo verrà richiamato solo nella *codeGeneration()* di CallNode.

5.1.3 ExpNodes

Di seguito viene spiegata la *codeGeneration()* per le espressioni.
Innanzitutto, viene generato il codice degli operandi, successivamente la *codeGeneration()* si distingue in base dell'operazione da eseguire:

- **Operatori Matematici di base (+, -, *, /)**: a ciascuno dei quattro operatori è stato assegnato un comando specifico (*add, sub, mult, div*). Ognuno di questi prende in input due registri con i valori delle espressioni calcolate precedentemente e carica nel registro *$a0* il risultato finale.

- **Operatori Booleani (&&, ||)**: a ciascuno operatore è stato assegnato un comando specifico (*and, or*). Ognuno di questi prende in input due registri con i valori delle espressioni calcolate precedentemente e carica nel registro *$a0* il risultato finale.

- **Operatori == e !=**: questi due operatori vengono gestiti entrambi dal comando *beq*. Tramite quest'ultimo è possibile saltare in un punto specifico del programma, indicato da una *label*, se i due registri passati in input hanno lo stesso valore.

- **Operatori $\leq, \geq, >, <$**: è stato assegnato il comando *bleq* all'operatore minore-uguale e tramite l'utilizzo combinato di quest'ultimo con *beq* è possibile derivare gli altri operatori, come da esempio in Listing 5.

- **BoolExp**: si carica in *$a0* il valore 1 se il booleano è *true*, altrimenti viene caricato 0.

- **NegExp**: dopo aver generato il codice dell'espressione, si moltiplica il valore per -1 e il risultato viene caricato in *$a0*.
- **NotExp**: dopo aver generato il codice dell'espressione, viene utilizzato il comando *not* e il risultato viene caricato in *$a0*.
- **NumExp**: carica in *$a0* il valore numerico.

```
1        mv $al $fp
2        lw $a0 1($al)
3        push $a0
4        li $a0 2
5        lw $t1 0($sp)
6        pop
7        beq $t1 $a0 equalTrueBranch
8        bleq $t1 $a0 lesseqTrueBranch
9        li $a0 1
10       b endlesseqTrueBranch
11   lesseqTrueBranch:
12       li $a0 0
13   endlesseqTrueBranch:
14       b endequalTrueBranch
15   equalTrueBranch:
16       li $a0 1
17   endequalTrueBranch:
```

Listing 5: Esempio dell'esecuzione di x $\geq$ 2

5.1.4 DecVarNode

Se, oltre alla dichiarazione, la variabile viene anche inizializzata bisogna generare il codice dell'espressione, risalire la catena e memorizzare il valore, tramite il comando *sw*, nell'*offset* corrispondente alla variabile già presente nella *symbol table*.

5.1.5 AssignmentNode

Come accade per DecVarNode, si effettua la generazione di codice dell'espressione, si risale la catena e si memorizza nell'offset corretto il valore dell'espressione. Se, invece, l'assegnamento è nel *body* di una funzione, si controlla che l'identificatore sia un riferimento e in questo caso carichiamo il valore dell'espressione nell'indirizzo passato come riferimento.

5.1.6 IteNode

Questo file fa riferimento allo *statement if-then-else*. Inizialmente viene generato il codice della condizione e del confronto con il valore 1 (*true*). Viene generata la *label* del ramo *then* alla quale il programma salterà in caso positivo.
Viene generato il codice del ramo *else*, se esiste. Successivamente, si aggiunge la label del ramo *then* con il rispettivo codice generato.

5.1.7 PrintNode

Nella *codeGeneration()* di questo file, innanzitutto si genera il codice dell'espressione e successivamente viene aggiunto il comando *print $a0*.

5.1.8 RetNode

Anche per il return, avviene la generazione di codice dell'espressione (se esiste). Dopodiché si aggiunge il comando del salto incondizionato (*b label*) seguito dalla label di fine blocco.

5.1.9 CallNode

Nella generazione del codice della chiamata di funzione, per prima cosa viene inserito nello *stack* il vecchio *frame pointer*.
In seguito, viene inserito nello *stack* il valore di ogni parametro attuale partendo dall'ultimo. Se un parametro formale è passato per riferimento viene richiamata la funzione *codeGenerationForRefType()* descritta nella sezione 5.1.2.
Fatto ciò, dopo aver risalito la catena, viene effettuato il *push* dell'*access link* e,

infine, si utilizza il comando *jal label*, che permette di saltare alla dichiarazione di funzione.
La Figura 4 mostra il record di attivazione per le chiamate di funzione.

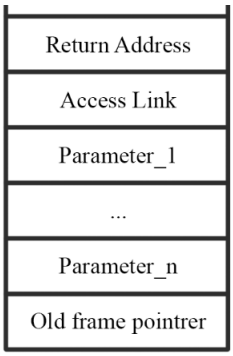

Figura 4: Record di attivazione della chiamata di funzione

5.1.10 DecFunNode

Per quanto riguarda la generazione di codice della dichiarazione di funzione, viene inserita la *label*, richiamata dal comando *jal* nella chiamata di funzione, ed effettuato il *push* del *return address*.
Dopodiché, si effettua la generazione di codice del *body*, si riprende il *return address* dal *top* dello *stack* e viene effettuato il *pop* di tutti gli argomenti, dell'*access link* e del *return address*. Infine, si ripristina il *frame pointer* e viene eseguito il comando *jr $ra* che salta all'indirizzo presente nel registro *$ra*.

5.2 Interprete

È stata definita la grammatica *SVM*, contenente le possibili istruzioni da generare:

```
push $r1 : decrementa $sp e inserisce $r1 nel top dello stack
pop : incrementa il valore di $sp
sw $r1 off($r2) : memorizza $r1 nell'indirizzo $r2+offset
lw $r1 off($r2) : carica in $r1 il valore nell'indirizzo $r2+offset
li $r1 NUMBER : carica NUMBER nel registro $r1
mv $r1 $r2 : memorizza il valore di $r2 in $r1
and $r1 $r2 $r3 : $r2 && $r3 viene memorizzato in $r1
or $r1 $r2 $r3 : $r2 || $r3 viene memorizzato in $r1
not $r1 $r2 : carica in $r1 il valore di !$r2
add $r1 $r2 $r3 : $r2 + $r3 viene memorizzato in $r1
sub $r1 $r2 $r3 : $r2 - $r3 viene memorizzato in $r1
mult $r1 $r2 $r3 : $r2 * $r3 viene memorizzato in $r1
div $r1 $r2 $r3 : $r2 / $r3 viene memorizzato in $r1
addi $r1 $r2 NUMBER : $r2 + NUMBER viene memorizzato in $r1
subi $r1 $r2 NUMBER : $r2 - NUMBER viene memorizzato in $r1
multi $r1 $r2 NUMBER : $r2 * NUMBER viene memorizzato in $r1
divi $r1 $r2 NUMBER : $r2 / NUMBER viene memorizzato in $r1
b LABEL : salto incondizionato all'istruzione puntata da LABEL
beq $r1 $r2 LABEL : salto alla LABEL se $r1 == $r2
bleq $r1 $r2 LABEL : salto alla LABEL se $r1 <= $r2
jal LABEL : salto alla LABEL nella definizione di funzione
jr $ra : salto al return address
print $r1 : stampa del valore in $r1
halt : termina l'esecuzione
```

Listing 6: Istruzioni SVM

e i possibili registri da utilizzare in queste ultime:

- **$a0**: è un registro speciale, detto **accumulatore**, usato per memorizzare il valore di ciascuna computazione;
- **$t1**: è il registro usato per memorizzare valori temporanei;
- **$sp**: è lo *Stack Pointer* e punta sempre al top della pila;

- **$fp**: è il *Frame Pointer* e punta all'ambiente statico dell'attuale record di attivazione;
- **$al**: è l'*Access Link* e viene usato per risalire la catena dinamica;
- **$ra**: è il *Return Address* e memorizza l'indirizzo di ritorno.

Dopo aver generato il codice, quest'ultimo viene processato dal lexer e dal parser della grammatica *SVM* generati automaticamente da ANTLRv4. Successivamente, il *visitor* crea l'albero di sintassi astratta e tramite il metodo *visit()* si genera una lista di Instruction (Sezione 5.2.1).
Dopo aver controllato che non ci siano errori sintattici, è stata istanziata la *VM* (Sezione 5.2.2) che prende in input la lista di Instruction precedentemente generata e, tramite il metodo *execute()* (Sezione 5.2.4) esegue il programma passato in input.

5.2.1 Classe Instruction

La classe Instruction è una struttura che viene utilizzata all'interno del *visitor* per contenere:

- **String name**: nome dell'istruzione;
- **String param1**: contiene il primo parametro dell'istruzione (se richiesto dall'istruzione);
- **String param2**: contiene il secondo parametro dell'istruzione (se richiesto dall'istruzione);
- **String param3**: contiene il terzo parametro dell'istruzione (se richiesto dall'istruzione);
- **Integer offset**: contiene l'offset (se richiesto dall'istruzione).

5.2.2 Virtual Machine

La classe VM contiene, oltre alla definizione della memoria e alla definizione dei registri, il metodo *execute()* (Sezione 5.2.4) necessario per l'esecuzione del

programma in input.
All'interno del costruttore:

- viene inizializzata la memoria come un *array* di grandezza *MEMSIZE* (di default uguale a 1000) contenente oggetti MemoryItem (Sezione 5.2.3);
- sono inizializzati *stack pointer* e *frame pointer* a *MEMSIZE*;
- viene inizializzato il *program counter* a 0.

Inoltre, sono stati definiti tre metodi che verranno utilizzati all'interno del metodo *execute()* (Sezione 5.2.4):

- **readRegister(String registerName)**: ritorna il valore del registro passato come parametro:

```
private Integer readRegister(String registerName) {
    return switch (registerName) {
        case "$sp" -> stackPointer;
        case "$hp" -> heapPointer;
        case "$fp" -> framePointer;
        case "$ra" -> returnAddress;
        case "$a0" -> a0;
        case "$t1" -> t1;
        case "$al" -> accessLink;
        default -> null;
    };
}
```

- **writeRegister(String registerName, Integer val)**: imposta il valore *val* all'interno del registro passato come primo parametro:

```
private void writeRegister(String registerName, Integer val) {
    switch (registerName) {
        case "$sp" -> stackPointer = val;
        case "$hp" -> heapPointer = val;
        case "$fp" -> framePointer = val;
        case "$ra" -> returnAddress = val;
        case "$a0" -> a0 = val;
```

```
8           case "$t1" -> t1 = val;
9           case "$al" -> accessLink = val;
10      }
11  }
```

- **getMemoryStatus()**: ritorna *MEMSIZE* se è stata esaurita la memoria, altrimenti ritorna la prima cella libera.

5.2.3 Memoria

La memoria della VM è definita come un *array* di MemoryItem. Ogni MemoryItem è una struttura contenente due campi:

- ***Integer data***: inizializzato a *null* e conterrà il valore del registro passato come parametro al metodo *setData(Integer data)* (quest'ultima imposta anche il booleano *isUsed* a *true*).
- ***boolean isUsed***: inizializzato a false e usato sia nell'istruzione *lw* per controllare che non si stia caricando nel registro un valore *null* sia nel metodo *getMemoryStatus()*.

5.2.4 Esecuzione

In questa sezione viene spiegato il metodo *execute()*, situato nella classe VM (Sezione 5.2.2), che esegue il programma passato in input.
Si verifica che ci sia ancora memoria disponibile controllando che il valore di ritorno del metodo *getMemoryStatus()* non sia maggiore dello *stack pointer*, in caso negativo si restituisce l'errore *"Out of memory"* e l'esecuzione termina. Successivamente, per ciascuna istruzione nella lista di *Instruction* (Sezione 5.2.1), vengono presi i tre parametri e l'offset che saranno utilizzati nelle varie istruzioni spiegate nella Sezione 5.2.

5.3 Codici da verificare/discutere

Esempio 1

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

La funzione definita in questo esempio calcola il fattoriale del numero passato come parametro. Quindi la chiamata *f(10)* stampa come output *3628800*, che risulta essere proprio il fattoriale di 10.
Il il codice intermedio generato per la variabile x, viene seguito da quello generato dalla chiamata di funzione *f(10)* che salterà al codice della definizione di funzione che, nella nostra implementazione, viene posto nell'ultima parte della generazione.

Esempio 2

```
{ int u = 1 ;
  void f(var int x, int n){
    if (n == 0) { print(x) ; }
    else { int y = x * n ; f(y,n-1) ; }
  }
  f(u,6) ;
}
```

Anche quest'ultimo codice restituisce come risultato il fattoriale di un numero, differenziandosi da quello precedente per l'uso di due parametri invece del singolo numero. Infatti, oltre al numero di cui si vuole calcolare il fattoriale, viene passata anche una variabile per riferimento che ad ogni iterazione, ad eccezione della prima, contiene il prodotto di due numeri successivi (ad esempio nella seconda iterazione *int y = 6 * 5*). Quindi, il codice stampa il fattoriale di 6 che risulta essere 720.

La particolarità in questo codice è il passaggio per riferimento della prima variabile. Nella nostro compilatore il passaggio per riferimento è implementato passando l'indirizzo di memoria della variabile passata. In questo modo quando nel *body* della funzione ci sono operazioni di lettura o scrittura della variabile in questione, queste vengono direttamente effettuate sull'indirizzo memorizzato.

Esempio 3

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

In quest'ultimo esempio, con la chiamata *f(5, 4)* verrà valutato subito il ramo *then* della funzione stampando come risultato 9.
Invece, chiamando *f(4, 5)* viene valutato il ramo else che, per prima cosa, restituirà il warning *"Id x initialized but not used"*, successivamente entra nella chiamata ricorsiva. Quando viene eseguita quest'ultima, i valori di m e n vengono incrementati contemporaneamente e questo porta alla non terminazione del programma, restituendo così l'errore *"Out of memory"*.