Report Compilatori e Interpreti
A.A. 2021/2022

Filippo Bartolucci
filippo.bartolucci2@studio.unibo.it

Alfonso Esposito
alfonso.esposito5@studio.unibo.it

Isabella Marasco
isabella.marasco3@studio.unibo.it

Indice

Capitolo 1

Introduzione

Questo progetto è parte della prova d'esame per il corso "Compilatori e Interpreti A.A. 2021/2022" il cui scopo è implementare un compilatore e interprete per un linguaggio di programmazione inventato, AssetLan.
AssetLan è un semplice linguaggio imperativo con asset, in cui i parametri possono essere sia standard che asset, con ricorsione e senza mutua ricorsione.

Il compilatore di AssetLan svolge le seguenti operazioni:

- *Analisi lessicale* con generazione dei token
- *Analisi sintattica* con generazione dell' Abstract Syntax Tree
- *Analisi semantica* sull'utilizzo delle variabili nel codice
- *Type checking* per espressioni e operazioni
- Controllo di *effetti* e *liquidity* del codice.

Successivamente si passa alla generazione del codice intermedio in un linguaggio bytecode che viene poi eseguito da un interprete.

Il progetto è stato sviluppato in *Java* utilizzando le librerie di *ANTLR*.

Capitolo 2

Sintassi

La grammatica di AssetLan ha come simbolo iniziale **init** che produce un blocco **program**, il quale è composto da una sequenza di dichiarazioni di variabili, di asset e di funzioni. L'entry point del programma è rappresentata dal blocco **initcall**.

```
init       : program ;
program    : field* asset* function* initcall ;
```

Nel caso delle dichiarazioni le possibili istruzioni sono tre:

- Variabili di tipo *intero* o *booleano*
- Asset
- Funzioni di tipo *void*, *intero* o *booleano*

```
field      : type ID ('=' exp)? ';' ;

asset      : 'asset' ID ';' ;

function   : type ID
           '(' (param (',' param)* )? ')'
           '[' (aparam (',' aparam)* )? ']'
           '{' field* statement* '}' ;
```

Gli statement possibili in AssetLan sono:

- assignment → operazione di assegnamento per le variabili

- move → sposta un asset da una parte ad un'altra
- print → stampa a schermo il valore della exp associata
- transfer → trasferisce l'asset all' utente ovvero chi esegue il codice
- ret → restituisce al chiamante il valore di exp
- ite → istruzione condizionale
- call → chiamata di funzione

```
statement  : assignment ';'
           | move ';'
           | print ';'
           | transfer ';'
           | ret ';'
           | ite
           | call ';' ;
```

Le espressioni in questo linguaggio possono essere scritte nel seguente modo:

```
exp        : '(' exp ')'                                    #baseExp
           | '-' exp                                        #negExp
           | '!' exp                                        #notExp
           | ID                                             #derExp
           | left=exp op=('*' | '/')          right=exp     #binExp
           | left=exp op=('+' | '-')          right=exp     #binExp
           | left=exp op=('<' | '<=' | '>' | '>=') right=exp #binExp
           | left=exp op=('=='| '!=')         right=exp     #binExp
           | left=exp op='&&'                 right=exp     #binExp
           | left=exp op='||'                 right=exp     #binExp
           | call                                           #callExp
           | BOOL                                           #boolExp
           | NUMBER                                         #valExp;
```

La libreria di *ANTLR* si è occupata della generaziona del lexer e del parser del linguaggio a partire dal file della grammatica.

Capitolo 3

Analisi Semantica

L'analisi semantica è suddivisa in tre fasi:

- Analisi degli scope e identificatori
- Type checking
- Analisi di effetti e liquidity

3.1 Environment e STEntry

La Symbol Table è fondamentale per l'analisi di scope e identificatori, in quanto ci permette di tenere traccia delle variabili e dei loro livelli di annidamento. Si tratta di una funzione Γ che prende in input un ID e restituisce la sua STentry, composta da Tipo, NestingLevel, Offset.

$$\Gamma : (Id) \rightarrow (Type, NestingLevel, Offset) \tag{3.1}$$

A livello di implementazione la SymbolTable è stata realizzata come una lista di hashmap, in cui l'indice della lista corrisponde al livello di annidamento e la hashmap allo scope al livello dell'indice.

La Symbol Table è rappresentata dalla classe java GammaEnv che fornisce i seguenti metodi:

newEmptyScope( ) : $\Gamma \rightarrow \Gamma \bullet [\ ]$

addDecl(x) : $\Gamma \rightarrow \Gamma_{[x \rightarrow Stentry(Type, NestingLevel, Offset)]}$

lookup(x) : $\Gamma(x) \rightarrow Stentry(Type, NestingLevel, Offset)$

exitScope( ) : $\Gamma \bullet \Gamma' \rightarrow \Gamma$

3.2 Analisi di scope e identificatori

In questa fase viene visitato per la prima volta l'Abstract Syntax Tree e ogni volta che si attraversa un nodo si fanno le seguenti operazioni:

- Per i nodi function vengono eseguite le operazioni di newEmptyScope() e exitScope().
- Per ogni dichiarazione viene aggiunta all'environment una nuova entry, con chiave l'id della variabile, e come valore una STentry inizializzata con i valori corrispondenti.
- per ogni statement o espressione viene controllato se le variabili utilizzate sono state già dichiarate in un punto precedente del codice

Di seguito vengono riportate le regole di inferenza utilizzate in questa prima visita:

$$\frac{X \notin dom(Top(\Gamma))}{\Gamma \ \vdash \ T \ X \ : \ \Gamma_{[x \to T]}} \text{ [VarDec]}$$

$$\frac{\begin{array}{c} T^a = Asset \qquad\qquad f \notin dom(Top(\Gamma)) \\ \Gamma_{[f \to (T_1,...,T_n,Asset_1,...,Asset_m) \to T, x_1 \to T_1,...,x_n \to T_n, a_1 \to T_a,...,a_m \to T_a]} \vdash e : T \end{array}}{\Gamma \ \vdash \ T \ f(T_1 x_1, ..., T_n x_n)[T^a a_1, ..., T^a a_m] \ = \ e; \ : \ \Gamma_{[f \to (T_1,...,T_n,Asset_1,...,Asset_m) \to T]}} \text{ [FunDec]}$$

$$\frac{\Gamma \vdash d : \Gamma' \quad \Gamma' \vdash D : \Gamma''}{\Gamma \ \vdash \ d \ D \ : \ \Gamma''} \text{ [DecSeq]}$$

3.3 Type Checking

AssetLan ammette 4 tipi: intero, booleano, asset e void (quest'ultimo utilizzabile solo dalle funzioni). Di seguito verranno discusse le regole formali per il tipaggio di tutti i costrutti del linaguaggio e le loro implementazioni.

- *Program* è il nodo iniziale del programma, include una serie di dichiarazioni ordinate di variabili, asset e funzioni seguite dalla entry point. Il tipo dell'entry point corrisponde al tipo del programma.

$$\frac{[\ ] \vdash D : \Gamma \quad \Gamma \vdash Initcall : T}{[\ ] \vdash D\ \ InitCall : T}\ \text{[Program]}$$

$$\frac{\Gamma(X) : T \quad \Gamma \vdash Exp : T}{\Gamma\ \vdash\ T\ X\ = Exp\ :\ T}\ \text{[Field]}$$

$$\frac{\Gamma(f) = (T_1, ...T_n, Asset_1, ..., Asset_m) \to T \quad (\Gamma \vdash x_i : T_i)_{\forall i\ in\ 1..n} \quad (\Gamma \vdash a_i : Int)_{\forall i\ in\ 1..m}}{\Gamma\ \vdash\ f(x_1, ...x_n)[a_1, ..., a_m]\ :\ T}\ \text{[InitCall]}$$

- In AssetLan gli statement possibili sono:

$$\frac{\Gamma(X) = T \quad \Gamma \vdash E : T' \quad T = T'}{\Gamma\ \vdash\ X\ =\ e : Void}\ \text{[Asg]}$$

$$\frac{\Gamma(f) = (T_1, ...T_n, Asset_1, ..., Asset_m) \to T \quad (\Gamma \vdash x_i : T_i)_{\forall i\ in\ 1..n} \quad (\Gamma \vdash a_i : Asset)_{\forall i\ in\ 1..m}}{\Gamma\ \vdash\ f(x_1, ...x_n)[a_1, ..., a_m]\ :\ T}\ \text{[Call]}$$

$$\frac{\Gamma \vdash exp : Bool \quad \Gamma \vdash Stm_1 : T_1 \quad \Gamma \vdash Stm_2 : T_2 \quad T_1 = T_2}{\Gamma\ \vdash\ if\ (exp)\ \{Stm_1\}\ else\ \{Stm_2\} :\ T_1}\ \text{[Ite]}$$

L'if restituisce un tipo se è presente un return non void nei due rami.

$$\frac{\Gamma(a_1) = Asset \quad \Gamma(a_2) = Asset}{\Gamma \ \vdash \ a_1 \ -o \ a_2 \ : \ Void} \text{ [Move]}$$

$$\frac{\Gamma \vdash Exp : T}{\Gamma \ \vdash \ print \ \ Exp : \ Void} \text{ [Print]}$$

$$\frac{\Gamma \vdash Exp : T}{\Gamma \ \vdash \ return \ \ Exp : \ T} \text{ [Ret]}$$

$$\frac{\Gamma(a) = Asset}{\Gamma \ \vdash transfer \ a \ : \ Void} \text{ [Transfer]}$$

- Di seguito vengono riportate le regole di inferenze per tipaggio delle espressioni:

$$\frac{}{\Gamma \vdash N \ \ : Int} \text{ [Int]}$$

$$\frac{}{\Gamma \vdash B \ \ : Bool} \text{ [Bool]}$$

$$\frac{\Gamma(ID) : T}{\Gamma \vdash ID \ \ : T} \text{ [ID]}$$

$$\frac{\Gamma \vdash Exp : T \quad T! = Bool}{\Gamma \vdash -Exp \ \ : T} \text{ [Neg]}$$

$$\frac{\Gamma \vdash Exp : Bool}{\Gamma \vdash !Exp \ \ : T} \text{ [Not]}$$

- *BinExpNode* può esprimere diverse funzioni binarie con parametri e tipi diversi.

$$\frac{\Gamma \vdash op : (T1xT2) \to T \quad \Gamma \vdash Exp1 : T1 \quad \Gamma \vdash Exp2 : T2}{\Gamma \ \vdash \ Exp\ op\ Exp \ : \ T} \text{ [BinExp]}$$

I tipi che può assumere **op** sono i seguenti:

+ : (int x int) -> int

\- : (int x int) -> int

* : (int x int) -> int

/ : (int x int) -> int

< : (int x int) -> bool

<= : (int x int) -> bool

> : (int x int) -> bool

>= : (int x int) -> bool

== : (T x T) -> bool

!= : (T x T) -> bool

&& : (bool x bool) -> bool

|| : (bool x bool) -> bool

In *BinExpNode* è ammesso l'uso del tipo asset insieme a int, ovvero è possibile fare operazioni aritmetiche tra asset e interi

3.4 Analisi di effetti e liquidity

AssetLan è un linguaggio semplice che non permette la cancellazione di variabili. Le variabili quando dichiarate possono essere non inizializzate, quindi è necessario fare attenzione al loro utilizzo negli statement.

Oltre ai tipi Int e Bool, AssetLan introduce gli Asset, risorse con valore intero e maggiore o uguale a zero. Il loro valore può essere spostato con un'operazione move o passato al chiamante con una transfer.

Un programma in AssetLan risulta corretto quando rispetta la Liquidity, ovvero:

- per ogni funzione, i parametri formali asset devono essere 0 alla fine della sua esecuzione
- alla fine del programma i campi asset sono 0.

Per verificare la liquidity è necessario operare un'analisi degli effetti del codice di un programma.

Per la gestione degli effetti è stata creata una nuova SymbolTable SigmaEnv (da qui in poi ci riferiremo a questo ambiente come Σ). Una lookup di un ID in Σ restituisce un valore booleano che rappresenta:

- per le variabili se è dichiarata (false $\perp$) o inizializzata (true RW)
- per gli asset se è vuoto (false) o pieno (true)

Di seguito vengono riportate le alcune delle regole di inferenza del linguaggio:

$$\frac{IDs(exp) = x_1, ..., x_n, a_1, ..., a_m \quad (\Sigma(x) = RW)_{\forall x \ in \ IDs(Exp)}}{\Sigma \vdash \ \ Exp : \ \ \Sigma'_{[x_1 \to RW, ..., x_n \to RW]}} \text{ [Exp]}$$

$$\frac{\Sigma \vdash e : \Sigma' \quad \Sigma(X) \leq RW}{\Sigma \ \vdash \ X \ = \ e : \ \Sigma'_{[X \to RW]}} \text{ [Asg]}$$

$$\frac{\sigma = \Sigma(a1)}{\Sigma \ \vdash \ a_1 \ - o \ a_2 \ : \ \Sigma_{[a1 \to 0, a2 \to \sigma]}} \text{ [Move]}$$

$$\frac{}{\Sigma \ \vdash \ transfer \ a \ : \ \Sigma_{[a \to 0]}} \text{ [Transfer]}$$

$$\frac{\Sigma \vdash exp : \Sigma' \quad \Sigma' \vdash Stm_1 : \Sigma_1 \quad \Sigma' \vdash Stm_2 : \Sigma_2}{\Sigma \ \vdash \ if \ (exp) \ \{Stm_1\} \ else \ \{Stm_2\} : \ Max(\Sigma_1, \Sigma_2)} \text{ [Ite]}$$

L'operazione Max(Env1, Env2) restituisce un nuovo ambiente Σ in cui gli effetti risultanti sono il massimo per ogni ID $\in \Sigma$

3.4.1 Punto fisso per l'analisi degli effetti di funzioni

L'analisi degli effetti delle chiamate richiede più attenzione rispetto agli altri statement. Il corpo di una funzione viene analizzato al momento della chiamata ed è necessario seguire altre possibili chiamate presenti nel corpo, facendo attenzione a quelle ricorsive.

Le funzioni permettono passaggio di variabili solo per valore, quindi vengono valutati gli effetti per ogni espressione usata come parametro nella chiamata.

$$f(exp_1, ..., exp_n)[a_1, ..., a_m] : (\Sigma \vdash exp_i)_{\forall i \in 1..n}$$

Il passaggio di parametri asset funziona diversamente da quello di variabili. L'asset parametro attuale si svuota e il suo valore viene passato all'asset parametro formale. Alla fine della chiamata di funzione viene controllato l'effetto per ogni asset locale e se uno di questi risulta pieno allora la liquidity non è rispettata.

Gli asset vengono valutati da destra verso sinistra per rispettare la codeGeneneration dei parametri in una chiamata di funzione.

Quando viene trovata una chiamata di funzione in uno statement si controllano gli effetti per ogni espressione e si salvano gli effetti attuali degli asset che verranno svuotati. A questo punto si assegnano agli asset formali gli effetti salvati in chiamata e vengono controllati gli effetti degli statement nel corpo della funzione. Al termine del corpo, se gli asset locali non sono stati svuotati con move e transfer la funzione non è liquida.

Codice per gestione del punto fisso nella chiamata.

```
if (env.isRecursive(this.id)){ // Checking recursive call
    Boolean fixedPoint = env.fixedPoint(actualEffects,ids);
    if (!fixedPoint){
        // Fixed point not reached...
        env = called_function.checkFunctionEffects(env,actualEffects);
        // After fixed point, updating effects after function call...
        actualEffects = env.getFixedPointResult();
        for(int i = 0; i < assets.size(); i++) {
            Node a = assets.get(i).getEntry();
            if(actualEffects.get(i)) {
                env.lookup(a).setTrue();
            }else{
                env.lookup(a).setFalse();
            }
        }
    }else{
        // Fixed Point!
        env.addFixedPointResult(env.getEffects(ids)); // Updating effects...
    }
}else{
    env.addFunctionCall(this.id);
    env = called_function.checkFunctionEffects(env,actualEffects);
}
```

In caso di chiamate ricorsive viene utilizzato il metodo del punto fisso per evitare di eseguire un loop infinito di analisi degli effetti delle chiamate.

Prima di una chiamata ricorsiva vengono confrontati gli effetti attuali degli asset e il loro valore ad inizio funzione. Se almeno uno degli effetti è diverso significa che il punto fisso non è stato ancora raggiunto ed è quindi necessario continuare a seguire la chiamata di funzione.
Invece, se gli effetti combaciano, il punto fisso è stato raggiunto e quindi non è necessario seguire la chiamata perché non ci saranno più variazioni.
Raggiunto il punto fisso si aggiornano gli effetti con i valori trovati che vengono poi passati alle chiamate precedenti.

Codice usato per controllare se il punto fisso è stato raggiunto

```
Boolean fixedPoint = true;
for(int i = 0; i < actualEffects.size() && fixedPoint; i++) {
    String id = ids.get(i);
    boolean status = this.lookup(id).getStatus();
```

```
    if (!status == actualEffects.get(i)){
        fixedPoint = false; // fixed point not reached
    }
}
return fixedPoint;
```

Capitolo 4

Generazione codice intermedio per AVM

Dopo aver effettuato l'analisi semantica e l'analisi degli effetti, il passo successivo è stato quello di generare il bytecode ovvero, un set di istruzioni progettato per essere eseguito da un interprete.
L'interprete di base è una classe Java che cicla su tutte le istruzioni prodotte dal compilatore e le esegue. Questa lettura avviene in un ciclo che si interrompe nel momento in cui viene letta l'istruzione halt o nel momento in cui viene esaurita la memoria della pila disponibile.
AssetLan Virtual Machine è una macchina a pila e a registri astratta che si occuperà di eseguire il bytecode generato dal compilatore.
Le istruzioni dell'interprete sono le seguenti:

```
instruction:
(
    PUSH n=NUMBER
      | PUSH r1=REGISTER
      | POP
      | ADD r1=REGISTER r2=REGISTER r3=REGISTER
      | ADDI r1=REGISTER r2=REGISTER n=NUMBER
      | SUB r1=REGISTER r2=REGISTER r3=REGISTER
      | SUBI r1=REGISTER r2=REGISTER n=NUMBER
      | MULT r1=REGISTER r2=REGISTER r3=REGISTER
      | MULTI r1=REGISTER r2=REGISTER n=NUMBER
      | DIV r1=REGISTER r2=REGISTER r3=REGISTER
      | DIVI r1=REGISTER r2=REGISTER n=NUMBER
      | OR r1=REGISTER r2=REGISTER r3=REGISTER
```

```
      | AND r1=REGISTER r2=REGISTER r3=REGISTER
      | NOT r1=REGISTER r2=REGISTER
      | STOREW r1=REGISTER o=NUMBER LPAR r2=REGISTER RPAR
      | LOADW r1=REGISTER o=NUMBER LPAR r2=REGISTER RPAR
      | LOAD r1=REGISTER n=NUMBER
      | MOVE r1=REGISTER r2=REGISTER
      | BRANCH l=LABEL
      | BCOND r1=REGISTER l=LABEL
      | EQ r1=REGISTER r2=REGISTER r3=REGISTER
      | LE r1=REGISTER r2=REGISTER r3=REGISTER
      | LT r1=REGISTER r2=REGISTER r3=REGISTER
      | GT r1=REGISTER r2=REGISTER r3=REGISTER
      | GE r1=REGISTER r2=REGISTER r3=REGISTER
      | JAL l=LABEL
      | JR r1=REGISTER
      | PRINT r1=REGISTER
      | TRANSFER r1=REGISTER
      | HALT
);
```

Di seguito una breve descrizione delle diverse funzionalità delle istruzioni:

- **Push**: memorizza il valore sulla pila e decrementa lo stack pointer.
- **Pop**: rimuove l'ultimo elemento dalla pila e incrementa lo stack pointer.
- **Add**: effettua la somma di due registri e inserisce il risultato nel primo registro.
- **Addi**: effettua la somma di un registro con una costante intera e inserisce il risultato nel primo registro.
- **Sub**: effettua la differenza di due registri e inserisce il risultato nel primo registro.
- **Subi**: effettua la differenza di un registro con una costante intera e inserisce il risultato nel primo registro.
- **Mult**: effettua il prodotto di due registri e inserisce il risultato nel primo registro.
- **Multi**: effettua il prodotto di un registro con una costante intera e inserisce il risultato nel primo registro.
- **Div**: effettua la divisione di due registri e inserisce il risultato nel primo registro.

- **Divi**: effettua la divisione di un registro con una costante intera e inserisce il risultato nel primo registro.
- **Or**: restituisce true se uno dei due registri valgono true, false nel caso in cui entrambi i registri valgono false.
- **And**: restituisce true se entrambi i registri valgono true, false altrimenti.
- **Not**: restituisce true se il registro vale false, altrimenti restituisce false.
- **StoreW**: salva il valore di r1 all'indirizzo di memoria calcolato sommando l'offset con r2.
- **LoadW**: carica il valore salvato all'indirizzo di memoria calcolato sommando l'offeset con r2 all'interno del registro r1.
- **Load**: carica un intero all'interno del registro r1.
- **Move**: salva il valore di r1 in r2.
- **Branch**: effettua un salto incondizionato alla label specificata.
- **BCond**: effettua un salto condizionato alla label specificata. Il salto avviene solo se il registro r1 è uguale a false.
- **Eq**: carica true in r1 se r2 e r3 sono uguali, altrimenti viene caricato false.
- **Le**: carica true in r1 se r2 è minore o uguale di r3, altrimenti viene caricato false.
- **Lt**: carica true in r1 se r2 è minore di r3, altrimenti viene caricato false.
- **Gt**: carica true in r1 se r2 è maggiore di r3, altrimenti viene caricato false.
- **Ge**: carica true in r1 se r2 è maggiore o uguale di r3, altrimenti viene caricato false.
- **Jal**: salta alla label specificata e salva l'istruzione successiva nel return address.
- **Jr**: salta all'istruzione specificata nel return address.
- **Print**: stampa il valore salvato nel registro r1.
- **Transfer**: carica il valore di r1 nel campo wallet della Asset Virtual Machine.
- **Halt**: ferma l'esecuzione del programma e stampa il valore del wallet.

4.1 Generazione del codice intermedio

Per generare il codice è stata utilizzata la funzione codeGeneration implementata per ogni costrutto del linguaggio, mostrate di seguito.

4.1.1 Program

AssetLan non prevede blocchi se non quello creato dal nodo iniziale program in cui sono contenuti una serie di dichiarazioni globali.

```
StringBuilder out = new StringBuilder();
out.append("//BEGIN PROGRAM\n\n");
out.append("//BLOCK \n");

out.append("push 0\n".repeat(fields.size() + assets.size()));

out.append("mv $sp $fp //Load $fp for initial block\n");
out.append("// GLOBAL FIELDS ASG\n");
for (Node f : fields){
    out.append(f.codeGeneration());
}
out.append("\n//INITCALL\n");
out.append(initcallnode.codeGeneration());
out.append("\nhalt //exit program...\n\n");
out.append("//FUNCTIONS\n\n");
for (Node f : functions) out.append(f.codeGeneration());
return out.toString();
```

Grammatica:

```
    program: field* asset* function* initcall ;
```

La grammatica di *program* richiama i nodi field, asset, function e initcall. Viene allocato spazio sulla pila per variabili e asset globali, una volta impostato il frame pointer del blocco iniziale viene chiamata la codeGeneration di *field* per assegnarli i valori. La codeGeneration viene chiamata sul nodo *Initcall* e sui nodi *Function*.

4.1.2 Field

```
if (exp == null) {
    return "";
}
StringBuilder out = new StringBuilder();
out.append("\n// Field ").append(id).append("\n");
out.append(exp.codeGeneration());
out.append("sw $a0 ").append(entry.getOffset()).append("($fp)").append("\n");
return out.toString();
```

Grammatica:

```
    field : type ID ('=' exp)? ';' ;
```

Nella codegeneration di *Field* viene chiamata la codegen su exp e il suo valore viene caricato all'indirizzo della variabile. Se non c'è inizializzazione viene restituita la stringa vuota.

4.1.3 Asset

```
return "push 0\n";
```

Grammatica:

```
    asset : 'asset' ID ';' ;
```

Nel nodo *Asset* viene allocato spazio sulla pila per le variabili.

4.1.4 Function

```
StringBuilder out = new StringBuilder();
out.append(f_label).append(": //").append(this.id).append("\n");
out.append("mv $sp $fp\n");
out.append("push $ra\n");
for (Node n : body_params) out.append(n.codeGeneration());
for (Node s:statements) out.append(s.codeGeneration());
out.append("\n").append(end_label).append(":\n");
out.append("subi $sp $fp 1 \n");
out.append("lw $fp 0($fp) \n");
out.append("lw $ra 0($sp)\n");
out.append("pop\n");
int parameter_size = params.size() + assets.size() + body_params.size();
out.append("addi $sp $sp ").append(parameter_size).append("\n");
out.append("pop\n");
out.append("lw $fp 0($sp)\n");
out.append("pop\n");
out.append("jr $ra\n");
return out.toString();
```

Grammatica:

```
function : type ID
           '(' (param (',' param)* )? ')'
           '[' (aparam (',' aparam)* )? ']'
           '{' field* statement* '}' ;
```

Nel nodo *Function* la codeGeneration inizia con una label che rappresenta la entry point della funzione, in seguito viene fatta la move per caricare il frame pointer per il record di attivazione per poi farne il push. Dopo aver impostato il frame pointer, per ogni variabile locale della funzione ne viene assegnato il valore attraverso la codeGeneration di *field*.
Per ogni nodo di *Statement* viene chiamata la codeGeneration. Nel caso in cui ci sia un return per poter saltare alla fine della funzione è stata creata la label *endLabel*.
Alla fine del corpo della funzione vengono effettuate le operazioni per il ripristino del frame pointer e vengono poi eseguite le pop per tutti i parametri di cui è stato fatto push durante la chiamata di funzione.

4.1.5 Initcall

```
StringBuilder out = new StringBuilder();
out.append("push $fp\n");
ArrayList<Node> bodyParams = this.getBodyParams();
out.append("push 0 \n".repeat(bodyParams.size()));

for (int i = aexp.size()-1; i>=0; i--){
    out.append(aexp.get(i).codeGeneration());
    out.append("push $a0 \n");
}

for (int i = exp.size()-1; i>=0; i--){
    out.append(exp.get(i).codeGeneration());
    out.append("push $a0 \n");
}
out.append("mv $fp $al //put in $al actual fp\n");
out.append("push $al\n");
out.append("jal ").append(this.getLabel()).append(" //Initcall: jump to start
    of ").append(id).append("\n");
return out.toString();
```

Grammatica:

```
initcall: ID '(' (exp (',' exp)* )? ')' '[' (aexp (',' aexp)* )? ']' ;
```

Il nodo *Initcall* è un caso particolare del nodo *call* essendo l'entry point del programma. Nella sua codeGeneration vengono valutati i parametri nel seguente ordine: bodyParam, aexp ed exp. Questo con lo scopo di rispettare l'ordine degli offset nelle dichiarazioni.

4.1.6 Statement

Grammatica:

```
statement  : assignment ';'
           | move ';'
           | print ';'
           | transfer ';'
           | ret ';'
           | ite
           | call ';' ;
```

Statement è un nodo di passaggio e viene chiamata la codeGeneration sul suo figlio.

Assignment

```
StringBuilder out = new StringBuilder();
out.append("\n// Asg ").append(id).append("\n");
out.append(exp.codeGeneration());
out.append("mv $fp $al //put in $al actual fp\n");
out.append("lw $al 0($al) //go up to chain\n".repeat(Math.max(0, this.currentNL
    - entry.getNestinglevel())));
out.append("sw $a0 ").append(entry.getOffset()).append("($al) //put $a0 in Id
    ").append(id).append("\n");
return out.toString();
```

Grammatica:

```
assignment : ID '=' exp ;
```

La codeGeneration di *Assignment* inizia con la valutazione dell'espressione di destra andando a richiamare la codeGeneration sul nodo *exp*. Il risultato della chiamata viene memorizzato all'interno della memoria all'indirizzo determinato dalla somma dell'offset della variabile con il valore dell'access link.

Move

```
StringBuilder out = new StringBuilder();
out.append("mv $fp $al");
out.append("lw $al 0($al)\n".repeat(Math.max(0, this.currentNL) -
    this.entry1.getNestinglevel()));
```

```
int offsetWithAL = entry1.getOffset();
out.append("lw $a0 ").append(offsetWithAL).append("($al)").append("\n");
out.append("push $a0");

out.append("li $a0 0 //emptying the asset\n");
out.append("sw $a0 ").append(offsetWithAL).append("($al)").append("\n");

out.append("mv $fp $al //moving the asset\n");
out.append("lw $al 0($al)\n".repeat(Math.max(0, this.currentNL) -
    this.entry2.getNestinglevel()));
offsetWithAL = entry2.getOffset();
out.append("lw $a0 ").append(offsetWithAL).append("($al)").append("\n");
out.append("lw $a2 0($sp)");
out.append("add $a0 $a2 $a0 \n");
out.append("sw $a0 ").append(offsetWithAL).append("($al)").append("\n");

return out.toString();
```

Grammatica:

```
move: ID '-o' ID ;
```

La codeGeneration dell'istruzione *Move* può essere divisa concettualmente in due passaggi principali:

1. Viene caricato all'interno del registro $a0 il valore del primo asset, che successivamente viene memorizzato sulla pila attraverso l'istruzione push. In seguito, viene svuotato l'asset salvando 0 all'interno dell'area di memoria riservata ad esso.
2. Successivamente viene caricato il valore del secondo asset nel registro $a0, al quale viene sommato il top della pila che contiene il valore del primo asset. Infine, viene caricato il nuovo valore nell'area di memoria riservata al secondo asset.

Print

```
return this.exp.codeGeneration() + "print $a0\n";
```

Grammatica:

```
print: 'print' exp ;
```

L'unica istruzione eseguita nella codeGeneration del nodo *Print* è la stampa del valore del registro e nel caso in cui sia presente un'espressione si valuta prima quest'ultima.

Transfer

```
StringBuilder out = new StringBuilder();
out.append("mv $fp $al //put in $al actual fp\n");
out.append("lw $al 0($al)\n".repeat(Math.max(0, this.currentNL) -
    this.entry.getNestinglevel()));
int offsetWithAL = entry.getOffset();
out.append("lw $a0 ").append(offsetWithAL).append("($al) //loads in $a0 the
    value in ").append(id).append("\n");

out.append("transfer $a0\n");

out.append("li $a0 0 // Emptying the asset...\n");
out.append("sw $a0 ").append(offsetWithAL).append("($al)").append("\n");
return out.toString();
```

Grammatica:

```
transfer: 'transfer' ID ;
```

L'esecuzione della codeGeneration per *Transfer* si occupa di caricare all'interno del registro $a0 il valore dell'asset per poi, andarlo a trasferire nel wallet dell'Asset Virtual Machine ed infine svuotarlo.

Return

```
StringBuilder out = new StringBuilder();
if( exp != null){
    out.append(exp.codeGeneration()).append("\n");
}
out.append("b ").append(parent_f.getEndLabel()).append("\n");
return out.toString();
```

Grammatica:

```
ret : 'return' (exp)? ;
```

Il nodo *Return* nella codeGeneration salta alla fine della funzione e in caso sia presente un'espressione si chiama la sua codeGeneration.

Ite

```
StringBuilder out = new StringBuilder();
String lFalse = LabelManager.getFreshLabel("false");
String lEnd = LabelManager.getFreshLabel("end");
out.append("//ite\n");
out.append(exp.codeGeneration());
out.append("bc $a0 ").append(lFalse).append("\n");
        // True
for (Node s : thenb) {
    out.append(s.codeGeneration());
}
out.append("b").append(" ").append(lEnd).append("\n");
out.append(lFalse).append(":\n");
// Else Branch
for (Node s : elseb) {
    out.append(s.codeGeneration());
}
// End
out.append(lEnd).append(":\n");
return out.toString();
```

Grammatica:

```
ite : 'if' '(' exp ')' '{' thenb = statementseq '}' ('else' '{' elseb =
    statementseq '}')?;
```

La codeGeneration di *Ite* comincia con l'esecuzione della codeGeneration dell'espressione la quale può restituire due valori:

0 : si salta alla label lFalse per poi eseguire la codeGneration del nodo else

1 : si chiama la codeGeneration del ramo then

Call

```
StringBuilder out = new StringBuilder();
out.append("//Call ").append(id).append(" call\n");
out.append("push $fp\n");
ArrayList<Node> bodyParams = this.getBodyParams();

out.append("push 0 \n".repeat(bodyParams.size()));

for (int i = assets.size()-1; i>=0; i--){
    STentry entry = assets.get(i);
    out.append("mv $fp $al\n");
    out.append("lw $al 0($al)\n".repeat(Math.max(0, this.currentNL) -
        entry.getNestinglevel()));
    int offsetWithAL = entry.getOffset();
    out.append("lw $a0 ").append(offsetWithAL).append("($al) //loads in $a0 the
        value in ").append(ids.get(i)).append("\n");
    out.append("push $a0 \n");

    out.append("li $a0 0\n");
    out.append("sw $a0 ").append(offsetWithAL).append("($al)").append("\n");
}

for (int i = expressions.size()-1; i>=0; i--){
    out.append(expressions.get(i).codeGeneration());
    out.append("push $a0 \n");
}

out.append("mv $fp $al // calling function...\n");
out.append("lw $al 0($al) //go up to chain\n".repeat(Math.max(0, currentNL -
    entry.getNestinglevel())));
out.append("push $al\n");
out.append("jal ").append(this.getLabel()).append("//jump to start of function
    and put in $ra next instruction\n");

return out.toString();
```

Grammatica:

```
call : ID '(' (exp (',' exp)* )? ')' '[' (ID (',' ID)* )? ']' ;
```

Con il nodo *Call* viene gestita la chiamata di una funzione. In prima istanza viene riservato lo spazio in memoria per tutte i parametri dichiarati nel body, per tutti gli asset e i parametri attuali. Infine viene gestita la chiamata della funzione caricando in $al l'access link corretto, che viene successivamente salvato sulla pila, e poi viene effettuata l' effettiva chiamata utilizzando l'istruzione jal, che effettua un salto incondizionato alla funzione identificata dalla label, e che salva in $ra l'indirizzo di ritorno dell'istruzione successiva.

4.1.7 Exp

Grammatica

```
exp        : '(' exp ')'                                          #baseExp
           | '-' exp                                              #negExp
           | '!' exp                                              #notExp
           | ID                                                   #derExp
           | left=exp op=('*' | '/')          right=exp          #binExp
           | left=exp op=('+' | '-')          right=exp          #binExp
           | left=exp op=('<' | '<=' | '>' | '>=') right=exp     #binExp
           | left=exp op=('=='| '!=')         right=exp          #binExp
           | left=exp op='&&'                 right=exp          #binExp
           | left=exp op='||'                 right=exp          #binExp
           | call                                                 #callExp
           | BOOL                                                 #boolExp
           | NUMBER                                               #valExp;
```

Attraverso il nodo *Exp* viene gestita la generazione del codice intermedio di tutte le espressioni del linguaggio. In particolare, meritano una menzione i nodi:

- DerExp
- BinExp
- NegExp
- NotExp

Il nodo *CallExpNode* richiama ricorsivamente la funzione codeGeneration sul nodo Call, che abbiamo già precedentemente discusso, mentre per il nodo BaseExpNode

analogamente chiamiamo ricorsivamente codeGeneration, ma in questo caso sul nodo Exp.

BinExp

```
StringBuilder out = new StringBuilder();
String left_generated = left.codeGeneration();
out.append(left_generated);
out.append("push $a0\n");
String right_generated = right.codeGeneration();
out.append(right_generated);
out.append("lw $a2 0($sp)\n");
out.append("pop\n");
switch (op) {
    case "+":{
    out.append("add $a0 $a2 $a0 // a0 = t1+a0\n");
    break;
}
case "-": {
    out.append("sub $a0 $a2 $a0 // a0 = t1-a0\n");
    break;
}
case "*": {
    out.append("mult $a0 $a2 $a0 // a0 = t1+a0\n");
    break;
}
case "/": {
    out.append("div $a0 $a2 $a0 // a0 = t1/a0\n");
    break;
}
case "<=":{
    out.append("le $a0 $a2 $a0 // $a0 = $a2 <= $a0\n");
    break;
}
case "<":{
    out.append("lt $a0 $a2 $a0 // $a0 = $a2 < $a0\n");
    break;
}
case ">":{
    out.append("gt $a0 $a2 $a0 // $a0 = $a2 > $a0\n");
    break;
}
case ">=":{
```

```
    out.append("ge $a0 $a2 $a0 // $a0 = $a2 >= $a0\n");
    break;
}
case "==":{
    out.append("eq $a0 $a2 $a0 // $a0 = $a2 == $a0\n");
    break;
}
case "!=":{
    out.append("eq $a0 $a2 $a0 // $a0 = $a2 == $a0\n");
    out.append("not $a0 $a0 // $a0 = !$a0\n");
    break;
}
case "&&":{
    out.append("and $a0 $a2 $a0 // $a0 = $a2 && $a0\n");
    break;
}
case "||":{
    out.append("or $a0 $a2 $a0 // $a0 = $a2 || $a0\n");
    break;
}
```

Il nodo *BinExp* rappresenta tutte le operazioni che sono possibili tra due registri. Possiamo suddividere la funzione che genera codice intermedio in due parti:

1. La prima parte si occupa richiamare ricorsivamente la codeGeneration sull'espressione di sinistra e sull'espressione di destra.
   Dopo aver valutato l'espressione di sinistra, il risultato viene caricato nel registro $a0 e successivamente memorizzato sullo stack attraverso l'istruzione push. In seguito viene valutata quindi l'espressione di destra, caricato il risultato sul registro $a0. Viene poi caricato nel registro $a2 il valore dell'espressione sinistra che è caricata sul top della pila, che viene poi eliminato con l'istruzione pop.
2. Da questo punto in poi la parte successiva della generazione di codice dipende esclusivamente dal tipo di operazione.

DerExp

```
StringBuilder out = new StringBuilder();
out.append("mv $fp $al // DerExpNode ").append(id).append("\n");
out.append("lw $al 0($al)\n".repeat(Math.max(0, this.currentNL) -
    this.entry.getNestinglevel()));
int offsetWithAL = entry.getOffset();
out.append("lw $a0 ").append(offsetWithAL).append("($al) //loads in $a0 the
    value in ").append(id).append("\n");
return out.toString();
```

Viene caricato il valore della variabile all'interno del registro $a0. Il valore è memorizzato all'indirizzo di memoria determinato dalla somma dell'offset della variabile con il valore dell'access link. Quest'ultimo in particolare viene determinato dalla posizione della variabile nello scope, e risalendo la catena statica fino al raggiungimento del nesting level corretto.

NegExp

```
return exp.codeGeneration() + "multi $a0 $a0 -1 //negate\n";
```

Il nodo *NegExp* chiama prima la codeGenaration di exp ed infine, moltiplica il valore ottenuto e caricato all'interno del registro $a0 con -1 e inserisce il risultato nel registro $a0.

NotExp

```
return exp.codeGeneration() + "not $a0 $a0 //not\n";
```

Il nodo *NotExp* chiama la codeGenaration di exp e restituisce il valore opposto a quello contenuto nel registro $a0.

ValExp

```
return "li $a0 " + this.val + "\n";
```

La codeGenaration di *ValExp* carica nel registro $a0 la costante intera this.val

BoolExp

```
return "li $a0 "+(this.bool?1:0)+"\n";
```

La codeGeneration di *BoolExpNode* carica nel registro $a0 0 se this.bool vale 0 (ossia false), altrimenti carica 1.

Capitolo 5

Test

In questa sezione viene discussa la parte relativa ai test creati per la valutazione del progetto AssetLan. In particolare gli esercizi da 1 a 4 sono stati utilizzati per valutare il corretto funzionamento dell'analisi semantica e dell'analisi degli effetti. L'esercizio 5,la modifica dell'esercizio 4 (che chiameremo d'ora in poi esercizio 4.1), e gli esercizi da 1 a 4 sono stati utilizzati per la valutazione della generazione di codice intermedio. Di seguito saranno presentate nelle sotto sezioni successive le discussioni relative ai test effettuati.

5.1 Test esempio 1

```
asset x;
asset y;
void f()[asset u,asset v]{
    u -o y ;
    v -o x ;
}
void main()[asset u, asset v]{
    u -o x ;
    u -o y ;
    f()[x,y] ;
}
main()[2,3] // scambia i valori di x e y; il contratto non e‘ liquido
```

Esecuzione Esempio 1

```
AssetLan Compiler
File: "./Test/test1" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!

Program type is: void

Checking effects...
Liquidity not respected -> x is not empty
Liquidity not respected -> y is not empty

2 Effect errors found -> Compilation failed.
```

Analisi semantica e analisi dei tipi

In figura possiamo osservare il risultato derivato dalla compilazione dell'Esempio 1. Per quanto riguarda l'analisi semantica e dei tipi, come evidenziato, non risulta nessun errore e il compilatore procede correttamente poi all'analisi degli effetti.

Analisi degli effetti

Per quanto riguarda l'analisi degli effetti viene controllata la liquidity e l'utilizzo di variabili non utilizzate. Ricordiamo che per rispettare la liquidity bisogna rispettare due condizioni:

1. per ogni funzione, i parametri formali asset devono essere 0 alla fine della sua esecuzione (i valori sono stati spostati nei campi asset oppure trasferiti con una transfer).

2. alla fine del programma i campi asset sono 0.

In questo esempio la liquidity non è rispettata perché gli asset x e y non sono stati svuotati prima della fine del programma.

5.2 Test esempio 2

```
int a;
int b = 0 ;
asset z ;
void g()[]{
    transfer z ;
}
void f(int x)[asset y]{
    a = y ;
    b = b+x ;
    y -o z ;
    g()[] ;
}
f(1)[2]       // il contratto e‘ liquido
```

Esecuzione Esempio 2

```
AssetLan Compiler
File: "./Test/test2" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!

Program type is: void

Checking effects...
Effects analysis successful! -> Liquidity is respected.

Code generation...
Code generation successful!

Launching interpreter...

Wallet: +2
Halting program...
```

Analisi semantica e analisi dei tipi

Per quanto riguarda l'analisi semantica e dei tipi non risulta nessun errore e il compilatore procede correttamente all'analisi degli effetti.

Analisi degli effetti

Non ci sono utilizzi di variabili non inizializzate e il programma è liquido perché per ogni funzione gli asset vengono svuotati.

Esecuzione programma

Il programma viene eseguito correttamente e restituisce al wallet dell'utente il valore 2.

5.3 Test esempio 3

```
int a ;
asset x ;
void f()[asset u, asset v, asset w]{
    u -o x ;
    f()[v,w,u];
}
void main()[asset a, asset b, asset c]{
    f()[a,b,c] ;
    transfer x ;
}
main()[1,2,3] ; // il contratto e‘ liquido
```

Esecuzione Esempio 3

```
AssetLan Compiler
File: "./Test/test3" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!
```

```
Program type is: void

Checking effects...
Effects analysis successful! -> Liquidity is respected.

Code generation...
Code generation successful!

Launching interpreter...

AVM Error -> Stack out of memory
Halting...
```

Analisi semantica e analisi dei tipi

Per quanto riguarda l'analisi semantica e dei tipi non risulta nessun errore e il compilatore procede correttamente all'analisi degli effetti.

Analisi degli effetti

Non ci sono utilizzi di variabili non inizializzate e il programma viene valutato come liquido attraverso il metodo del punto fisso.

Esecuzione programma

Il programma viene eseguito correttamente, ma non arriva mai al termine ed esaurisce così la memoria dell'interprete.

5.4 Test esempio 4

```
asset x ;
void f(int n)[asset v, asset u]{
   if (n == 0){
        u -o x ;
   }else{
        u -o x ;
        v -o x ;
   }
}
void main()[asset a]{
```

```
    f(0)[a,a] ; // semantica di f()[a,a] -> la a di destra viene
    transfer x; // svuotata prima di quella di sinistra
                // gli asset vengono valutati da destra per gli effetti
}
main()[1] ; // il contratto e' liquido
```

Esecuzione Esempio 4

```
AssetLan Compiler
File: "./Test/test4" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!

Program type is: void

Checking effects...
Effects analysis successful! -> Liquidity is respected.

Code generation...
Code generation successful!

Launching interpreter...

Wallet: +1
Halting program...
```

Analisi semantica e analisi dei tipi

Per quanto riguarda l'analisi semantica e dei tipi non risulta nessun errore e il compilatore procede correttamente all'analisi degli effetti.

Analisi degli effetti

Non ci sono utilizzi di variabili non inizializzate e il programma è liquido perché per ogni funzione gli asset vengono svuotati.

Esecuzione programma

Il programma viene eseguito correttamente e restituisce al wallet dell'utente il valore 1.

5.5 Test esempio 5

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

Esecuzione Esempio 5

```
AssetLan Compiler
File: "./Test/test5" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!

Program type is: void


Checking effects...
Effects analysis successful! -> Liquidity is respected.

Code generation...
Code generation successful!

Launching interpreter...
```

```
Print: 3628800
Wallet: +0
Halting program...
```

Analisi semantica e analisi dei tipi

Per quanto riguarda l'analisi semantica e dei tipi non risulta nessun errore e il compilatore procede correttamente all'analisi degli effetti.

Analisi degli effetti

Non ci sono utilizzi di variabili non inizializzate e non ci sono asset da svuotare

Esecuzione programma

Il programma viene eseguito correttamente e stampa il fattoriale di 10.

5.6 Test esempio 4.1

Il seguente codice è quello dell'esecizio 4 con il main modificato.

```
asset x ;
void f(int n)[asset u, asset v]{
   if (n == 0){
        u -o x;
   }else{
        u -o x;
        v -o x;
   }
}
void main()[asset a, asset b]{
    f(0)[a,b];
    print x;
    transfer x;
}
main()[1,2];
```

Esecuzione Esempio 4.1

```
AssetLan Compiler
File: "./Test/test1" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!

Program type is: void

Checking effects...
Liquidity in f not respected -> v is not empty

1 Effect errors found -> Compilation failed.
```

Analisi semantica e analisi dei tipi

Per quanto riguarda l'analisi semantica e dei tipi non risulta nessun errore e il compilatore procede correttamente all'analisi degli effetti.

Analisi degli effetti

Il contratto non è liquido perche in f l'asset v viene caricato pieno e non viene mai svuotato.

5.7 Killer Application

Questa breve sezione ha lo scopo di mostrare l'esecuzione del codice fornito in sede di ricevimento come test finale per la valutazione del compilatore

5.8 Test killer application

```
int a ;
asset x ;
void f1()[asset a, asset b, asset c, asset d, asset e, asset f, asset g, asset
    h, asset i, asset j]{
     a -o x ;
     f1()[b,c,d,e,f,g,h,i,j,a];
}
void main()[asset a, asset b, asset c, asset d, asset e, asset f, asset g,
    asset h, asset i, asset j]{
     f1()[a,b,c,d,e,f,g,h,i,j];
     transfer x ;
}
main()[1,2,3,4,5,6,7,8,9,10]
```

Esecuzione killer application

```
AssetLan Compiler
File: "./Test/test3" found.

Parsing...
Parsing successful!

Semantic analysis...
Semantic analysis successful!

Type checking...
Type checking successful!

Program type is: void

Checking effects...
Effects analysis successful! -> Liquidity is respected.

Code generation...
Code generation successful!
```

```
Launching interpreter...

AVM Error -> Stack out of memory
Halting...
```

Analisi semantica e analisi dei tipi

Per quanto riguarda l'analisi semantica e dei tipi non risulta nessun errore e il compilatore procede correttamente all'analisi degli effetti.

Analisi degli effetti

Non ci sono utilizzi di variabili non inizializzate e il programma viene valutato come liquido attraverso il metodo del punto fisso.

Esecuzione programma

Il programma viene eseguito correttamente, ma non arriva mai al termine ed esaurisce così la memoria dell'interprete.