

Report Progetto Compilatori ed Interpreti

Marco Benito Tomasone 1038815

Luca Genova 1038843

Simone Boldrini 1038792

2021-2022

Indice

1	Introduzione	2
1.1	Grammatica del Linguaggio	2
2	Analisi Lessicale e Sintattica	4
2.1	Esempi	5
3	Analisi Semantica	8
3.1	Environment	8
3.2	STentry	9
3.3	Check Semantics	9
3.3.1	Variabili o funzioni non dichiarate	9
3.3.2	Dichiarazioni Multiple nello stesso ambiente	10
4	Controllo dei tipi	11
4.1	Regole di inferenza utilizzate	11
4.2	Comandi	14
4.3	Espressioni	15
5	Analisi degli Effetti	18
6	Generazione di codice	21
6.1	Interprete	22
7	Compilatore AssetLan	24

Capitolo 1

Introduzione

Questo documento è il Report per il progetto del corso di Compilatori ed Interpreti del prof. Cosimo Laneve A.S. 2021/2022. Il progetto ha come obiettivo quello di costruire un compilatore per il linguaggio *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.

1.1 Grammatica del Linguaggio

Ogni programma in *AssetLan* è formato da quattro parti principali che sono le dichiarazioni di variabili, le dichiarazioni degli asset, le dichiarazioni delle funzioni e la chiamata a delle funzioni. I tipi permessi per le variabili sono *int* e *bool*, mentre per le funzioni oltre ai due tipi sopracitati è permesso dichiarare funzioni *void*. I parametri asset hanno tipo *asset*.

Nella dichiarazione di funzioni oltre al tipo e al nome della funzione bisogna indicare una lista di parametri tra parentesi tonde e una lista di asset tra parentesi quadre. Nel corpo di una funzione è possibile fare molteplici dichiarazioni di variabili e poi definire degli statement. Uno statement può essere un assegnamento, una funzione move che permette di spostare il contenuto di un asset all'interno di un altro asset (svuotando il primo asset), una print, una transfer, una return, un *if-then-else* oppure una chiamata a funzione. Rispetto alla grammatica che ci è stata fornita inizialmente abbiamo apportato due piccole modifiche:

```
1 function: (type | 'void') ID '('(decp)?')' '['(adec)?']'
           {' (dec';')* statement* '}' ;
2 decp: dec;
```

La prima è per separare la dichiarazione dei parametri formali di una funzione dalle dichiarazioni delle variabili nel corpo della funzione. Per fare ciò è stato

inserito un nuovo simbolo non terminale *dec* che richiama il non terminale *dec*.

```
1  initcall: ID '(' (exp (',' exp)* )? ')' '[' (bexp (','  
    bexp)* )? ']' ;  
2  bexp: exp ;
```

La seconda modifica apportata riguarda l'introduzione di un nuovo simbolo *bexp* che sostituisce le chiamate al simbolo *exp* all'interno dei parametri asset nella chiamata di una funzione.

Capitolo 2

Analisi Lessicale e Sintattica

Le prime due fasi del compilatore per il linguaggio AssetLan hanno previsto la costruzione dell'analizzatore lessicale e sintattico. *L'analizzatore lessicale* prende in input le stringhe di codice in AssetLan e ritorna una lista di token. L'analizzatore lessicale genera degli errori dal momento in cui non riconosce un certo token all'interno della grammatica.

L'analizzatore sintattico (parser) prende in input la lista di token generati dall'analizzatore lessicale e cerca di ricostruire l'Albero di Sintassi Astratta (AST). Se la lista di token non rispetta la grammatica, non sarà possibile costruire l'albero ed il parser genererà un errore.

Per la costruzione del lexer e del parser abbiamo utilizzato ANTLR, un generatore di parser che utilizza il sistema di parsing LL.

Per ritornare la lista degli errori lessicali e sintattici abbiamo creato la classe *SyntaxErrorListener* che estende la classe *BaseErrorListener*. La classe contiene un *ArrayList* che conterrà tutti gli errori. Nella funzione *syntaxError*, di cui mostriamo uno snippet di codice in basso, gli errori vengono classificati in lessicali e sintattici in base al riconoscitore che li ha generati (lexer o parser) e verranno poi stampati in un file tramite un *PrintWriter*.

```
3
4 public void syntaxError(...) {
5     ...
6     while(errors.hasNext()) {
7         SyntaxError i = (SyntaxError)errors.next();
8         if(i.getRecognizer().getClass().getSimpleName().
           equals("AssetLanLexer"))
9             lexicalErrors += "Error: " + i.getMsg() + "
              at line " + i.getLine() + "\n";
10        else
11            syntaxErrors += "Error: " + i.getMsg() + "
```

```

12         }
13     ...
14 }
        at line " + i.getLine() + "\n";

```

Nel main del nostro compilatore, iniziamo leggendo il file in input da compilare nella variabile *charStreams*. Inizializzeremo poi un lexer al quale passeremo il file in input e ci darà in output una lista di token. Questa lista di token verrà data in pasto al parser. Inizializzeremo poi un *errorListener* di tipo *SyntaxErrorListener* precedentemente descritto e lo aggiungeremo come *ErrorListener* sia al lexer che al parser. Non è necessario rimuovere gli altri *errorListener* poichè come indicato dalla documentazione di ANTLR è possibile avere più di un tipo di *ErrorListener* per queste due classi.

```

15     CharStream charStreams = CharStreams.
        fromFileName(fileName);
16     AssetLanLexer lexer = new AssetLanLexer(
        charStreams);
17     CommonTokenStream tokens = new CommonTokenStream
        (lexer);
18     AssetLanParser parser = new AssetLanParser(
        tokens);
19
20     SyntaxErrorListener errorListener = new
        SyntaxErrorListener();
21     lexer.addErrorListener(errorListener);
22     parser.addErrorListener(errorListener);

```

2.1 Esempi

In questo primo esempio abbiamo inserito dei token non ammessi dalla grammatica (\$,@,~ ?). Notiamo come negli errori lessicali siano riportati i primi tre simboli mentre non figura il "?" poichè inserito all'interno di un commento.

```

1     asset x;$
2     void f(int n)[asset $u, asset v]{
3     if (n == 0) {u -o x ;}
4     else { u -o x ; v -o x ;}
5     }
6     void main()[asset a]{
7     f(0)[a,a] ;          // semantica di f()[a,a] ?

```

```

8      transfer @x ;
9      }
10     main(~)[1] ;

1      -----LEXICAL ERRORS-----
2      Error: token recognition error at: '$' at line 1
3      Error: token recognition error at: '$' at line 2
4      Error: token recognition error at: '@' at line 8
5      Error: token recognition error at: '~' at line
        10
6
7      -----SYNTAX ERRORS-----

```

In questo secondo esempio non abbiamo inserito errori lessicali ma abbiamo inserito le dichiarazioni degli asset e delle variabili dopo le dichiarazioni delle funzioni. Notiamo come dia come errore lessicale la presenza della parola *asset* alla riga 10, mentre si aspetta di trovare una dichiarazione di funzione o un id per richiamare una funzione.

```

1      void f()[asset u ,asset v]{
2          u -o y ;
3          v -o x ;
4      }
5      void main()[asset u ,asset v]{
6          u -o x ;
7          u -o y ;
8      f()[x,y] ;
9      }
10     asset x, y ;
11     int a,b = 3;
12     main()[2,3]

1      -----LEXICAL ERRORS-----
2
3      -----SYNTAX ERRORS-----
4      Error: extraneous input 'asset' expecting {'void
        ', 'int', 'bool', ID} at line 10
5      Error: mismatched input ',,' expecting '(' at
        line 10

```

In questo terzo esempio manca il non terminale *initcall* della grammatica, che è obbligatorio. Il parser in questo caso genera un errore dichiarando come riconosce l'<EOF> invece di trovare un'ulteriore dichiarazione a funzione o la chiamata ad una funzione.

```

1      int a ;
2      asset x ;
3      void f(int n)[asset u, asset v, asset w]{
4          u -o x ;
5          f(n - 1)[v,w,u] ;
6      }
7      void main()[asset a, asset b, asset c]{
8          f(1)[a,b,c] ;
9          transfer x ;
10     }

1      -----LEXICAL ERRORS-----
2
3      -----SYNTAX ERRORS-----
4      Error: mismatched input ';' expecting '[' at
        line 5
5      Error: extraneous input '<EOF>' expecting {'void
        ', 'int', 'bool', ID} at line 10

```


Capitolo 3

Analisi Semantica

3.1 Environment

In questo progetto l'ambiente viene definito come un insieme di scope, gestiti dalla classe *Environment*, che contiene la *symbolTable* implementata come una lista di *HashMap* di stringhe e *STentry*. La classe *Environment* inoltre contiene un campo *nestingLevel* ed un campo *offset* che servirà come “contatore” per la gestione dell'offset delle varie variabili. I metodi della classe *Environment* si occupano della gestione della *symbolTable*. Ogni volta che si entra in un nuovo ambiente viene creata una nuova *symbolTable* che viene aggiunta in testa alla lista e viene aumentato il *nestingLevel*. Allo stesso modo quando si esce da un ambiente viene eliminata la hash map che si trova in testa e diminuito il *nestingLevel* e ristabilito l'offset. In questa classe sono anche presenti alcune funzioni per lavorare con gli environment:

- *lookup()*: restituisce la entry di un id all'interno della *symbolTable*
- *addDeclaration()*: aggiunge una dichiarazione per un certo id
- *exitScope()*: performa l'uscita da uno scope
- *newScope()*: performa la creazione di un nuovo scope
- *clone()*: crea una nuova istanza di un ambiente a partire da un altro
- *equals()*: restituisce true o false a seconda che due ambienti siano uguali o meno

3.2 STentry

STentry è un'interfaccia che viene implementata da tre possibili classi che sono:

- *STentryFun*: crea l'istanza dell'STentry di una funzione. Contiene un nodo funzione, il nestingLevel e l'offset.
- *STentryVar*: crea l'istanza dell'STentry di una variabile booleana o intera. Contiene il tipo, il nestingLevel e l'offset.
- *STentryAsset*: è un'estensione di STentryVar, alla quale aggiunge il campo *liquidity*.

3.3 Check Semantics

Le funzioni *checkSemantics* di ogni nodo si occupano di andare a effettuare dei controlli semantici. Molte di loro sono implementate chiamando ricorsivamente i controlli semantici sui nodi dai quali sono composti. In questa fase ci siamo preoccupati principalmente di controllare variabili e funzioni non dichiarate ed eventuali dichiarazioni multiple nello stesso ambiente. Tutte le funzioni *checkSemantics* ritornano una lista di errori semantici.

3.3.1 Variabili o funzioni non dichiarate

Per quanto riguarda il controllo di variabili o funzioni non dichiarate, riportiamo come esempio uno snippet di codice dalla classe *MoveNode*.

```
6
7 public ArrayList<SemanticError> checkSemantics(
8     Environment e) {
9     ArrayList<SemanticError> res = new ArrayList<
10         SemanticError>();
11     if(e.isDeclared(id1.getId()) == EnvError.NO_DECLARE)
12     {
13         res.add(new SemanticError((id1.getId())+": is
14             not declared [Move]"));
15     }
16     if(e.isDeclared(id2.getId())== EnvError.NO_DECLARE){
17         res.add(new SemanticError((id2.getId())+": is
18             not declared [Move]"));
19     }
20     return res;
```

16 } |

In questo caso per entrambi gli operandi della funzione `move` si controlla che all'interno di qualsiasi ambiente della `symbolTable` sia presente una dichiarazione per quella variabile, altrimenti in caso contrario viene aggiunto un errore. I `checkSemantics` per le altre classi sulle quali viene effettuato un controllo su possibili variabili non dichiarate sono praticamente speculari a questo.

3.3.2 Dichiarazioni Multiple nello stesso ambiente

Per quanto riguarda le dichiarazioni multiple si va a controllare che non esista all'interno dello stesso ambiente nel quale si vuole effettuare la dichiarazione, una dichiarazione con lo stesso nome. Riportiamo, a titolo di esempio, uno snippet di codice preso dalla classe *AssetNode*.

```
17 public ArrayList<SemanticError> checkSemantics(  
    Environment e) {  
18     ArrayList<SemanticError> res = new ArrayList<  
        SemanticError>();  
19     if(e.isMultipleDeclared(id.getId()) == EnvError.  
        NO_DECLARE)  
20         Environment.addDeclaration(e,id.getId(),"asset")  
            ;  
21     else  
22         res.add(new SemanticError(id.getId()+" already  
            declared [assetNode]"));  
23     return res;  
24 }
```

Mostriamo ora un esempio di codice in cui sono presenti questi due tipi di errore:

```
1 int f;  
2 asset x ;  
3 void f(int n)[asset u, asset v]{  
4     if (n == 0) {u -o y ;}  
5     else { u -o x ; v -o x ;  
6 }  
7 f(5) [2,3] ;
```

E gli errori che vengono generati:

```
1 f: id already declared [function]  
2 y: is not declared [Move]
```

Capitolo 4

Controllo dei tipi

Il controllo dei tipi viene effettuato tramite la funzione *Type Check*. I tipi presenti nel nostro linguaggio sono:

- *int*: rappresenta un tipo intero;
- *bool*: rappresenta il tipo booleano, può assumere quindi valori *true* o *false*;
- *asset*: rappresenta i parametri asset delle funzioni, che andranno tra parentesi quadre e rappresentano delle risorse generiche.
- *void*: rappresenta il tipo vuoto, utile per le funzioni che non hanno un valore di ritorno.

4.1 Regole di inferenza utilizzate

Il nostro ambiente, Γ , sarà una funzione così fatta:

$$\Gamma : ID \rightarrow T \times \text{off}$$

Associerà quindi un nome ad un tipo e ad un offset. L'insieme T è così definito:

$$T = \{\text{void}, \text{asset}, \text{bool}, \text{int}\}$$

Definiamo inoltre una funzione $proj_i$ che ci permetterà di ritornare l'*i-esimo* elemento di $\Gamma(\text{id})$.

La prima regola che andremo ad analizzare è la regola del program. Questa regola verrà valutata nell'ambiente vuoto (poichè stiamo iniziando a controllare ora il programma) e avrà un offset pari a 0. L'offset è infatti una

variabile globale all'interno della classe Environment che parte da zero e diminuirà ogni volta che viene effettuata una nuova dichiarazione.

$$\frac{\emptyset, 0 \vdash field : \Gamma \quad \Gamma \vdash asset : \Gamma' \quad \Gamma' \vdash function : \Gamma'' \quad \Gamma'' \vdash initcall : T}{\emptyset, 0 \vdash field \ asset \ function \ initcall : T}$$

Successivamente, si va a valutare la regola field, che non è altro che una dichiarazione di variabili con o senza inizializzazione. Abbiamo così definito due regole di inferenza, una con ed una senza l'inizializzazione.

$$\frac{x \notin (top(\Gamma)) \quad off = n - 1}{\Gamma, n \vdash T \ x; : \Gamma[x \rightarrow T \times off], n = n - 1}$$

La regola con l'inizializzazione sarà:

$$\frac{x \notin (top(\Gamma)) \quad \Gamma \vdash e = T' \quad T = T' \quad off = n - 1}{\Gamma, n \vdash T \ x = e; : \Gamma[x \rightarrow T \times off], n = n - 1}$$

Dopo field, ci sarà la possibilità di definire degli asset globali. Gli asset globali non potranno mai essere inizializzati, quindi avremo bisogno di una sola regola di inferenza. Quando dichiareremo un asset il suo valore liquidity sarà inizialmente settato a zero, poichè questi asset saranno vuoti.

$$\frac{x \notin dom(top(\Gamma)) \quad off = n - 1}{\Gamma, n \vdash asset \ x; : \Gamma[x \rightarrow asset \times off], n=n-1}$$

Entrambi i campi field e asset permettono di effettuare più dichiarazioni in sequenza. Bisogna, quindi, definire la regola per la valutazione di una sequenza di dichiarazioni field ed una sequenza di dichiarazioni asset. Mostriamo quindi la regola per la sequenza di dichiarazioni field.

$$\frac{\Gamma, n \vdash f : \Gamma', n' \quad \Gamma', n' \vdash F; : \Gamma'', n''}{\Gamma, n \vdash f; F; : \Gamma'', n''}$$

Una regola speculare viene usata per la sequenza di dichiarazioni asset.

$$\frac{\Gamma, n \vdash a : \Gamma', n' \quad \Gamma', n' \vdash A; : \Gamma'', n''}{\Gamma, n \vdash a; A; : \Gamma'', n''}$$

Sia i parametri formali che le dichiarazioni interne al corpo di una funzione vengono gestiti tramite regola dec (da noi rinominata in decp per i parametri formali). La regola per la dichiarazione sarà quindi:

$$\frac{x \notin (top(\Gamma)) \quad off = n + 1}{\Gamma, n \vdash T \ x; : \Gamma[x \rightarrow T \times off], n = n + 1}$$

Come si può notare in questo caso c'è un'importante differenza, l'offset invece che diminuire di uno aumenterà di uno. Questo è dettato dal fatto che all'interno del record di attivazione il *frame pointer* si troverà in altro (ad indirizzi di memoria inferiori) rispetto ai parametri formali e alle dichiarazioni di variabili. Questo non creerà problemi con gli offset delle altre variabili (come le dichiarazioni globali) poichè le variabili *dec* e *decp* (come anche *adec* che mostreremo successivamente) si troveranno in un nuovo scope e come mostreremo più avanti dalla regola della funzione, nel nuovo scope l'offset verrà riazzerato. Anche in questo caso è necessario definire la regola per la sequenza di dichiarazioni *dec* e *decp*.

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

La regola di inferenza per l'*adec* (quindi i parametri formali *asset* di una funzione) è praticamente speculare a quella della dichiarazione degli *asset* globali con la differenza che anche in questo caso l'offset sarà positivo e non negativo.

$$\frac{x \notin \text{dom}(\text{top}(\Gamma)) \quad \text{off} = n + 1}{\Gamma, n \vdash \text{asset } x; : \Gamma[x \rightarrow \text{asset} \times \text{off}], n = n + 1}$$

Anche in questo caso la regola per la definizione della sequenza di definizioni di parametri *asset* formali sarà speculare a quella precedente.

$$\frac{\Gamma, n \vdash \text{adec} : \Gamma', n' \quad \Gamma', n' \vdash \text{Adec}; : \Gamma'', n''}{\Gamma, n \vdash \text{adec}; \text{Adec}; : \Gamma'', n''}$$

4.2 Comandi

Tutti i comandi ritornano un tipo per mantenere coerenza con il comando `return` che ritornerà un tipo da utilizzare per controllare la coerenza con il tipo di ritorno della funzione. Il tipo di ritorno dei comandi è però inutile e potrebbe essere anche `void`. Passando ora alla valutazione degli statement, la regola di inferenza per l'*assegnamento* è:

$$\frac{x \in \text{dom}(\Gamma) \quad \text{proj}_1(\Gamma(x)) = T \quad \Gamma \vdash e = T' \quad T' \leq T}{\Gamma \vdash x = e : \Gamma}$$

Non c'è bisogno di controllare che T e/o T' siano diversi da `void` poichè è vietato dalla grammatica dichiarare un identificatore che non sia una funzione di tipo `void`. Questo vincola T ad essere di tipo *int* o *bool* e di conseguenza vincola T' ad essere di tipo *bool* se T è anch'esso *bool* o ad essere *int* o *asset* (poichè *asset* è un sottotipo di *int*) se T è di tipo intero.

La regola per la *print* è:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{print } e : \Gamma}$$

La regola per la *move* va a valutare i due parametri, controllando che siano dichiarati (non all'interno del dominio di testa ma all'interno di tutto il programma) e controlla che entrambi i parametri siano di tipo *asset*. La regola quindi è:

$$\frac{a_1 \in \text{dom}(\Gamma) \quad a_2 \in \text{dom}(\Gamma) \quad \text{proj}_1(\Gamma(a_1)) = T \quad \text{proj}_1(\Gamma(a_2)) = T' \quad T = T' = \text{asset}}{\Gamma \vdash a_1 - o \ a_2 : \text{void}}$$

Regola per la *transfer*, va a valutare che il parametro *asset* in input sia definito all'interno del programma e che sia di tipo *asset*:

$$\frac{x \in \text{dom}(\Gamma) \quad \text{proj}_1(\Gamma(x)) = T \quad T = \text{asset}}{\Gamma \vdash \text{transfer } x : \text{void}}$$

Per la *return* avremo bisogno di più regole di inferenza a seconda che ci sia un'espressione di ritorno o la funzione sia di tipo `void` e quindi la *return* non abbia un'espressione. La regola per la *return* senza espressione è un assioma:

$$\overline{\Gamma \vdash \text{return}; : \text{void}}$$

Regola per la *return* con *exp*:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{return } e : T}$$

Serviranno ora due regole per vincolare la valutazione della return, il cui tipo di ritorno verrà utilizzato nella regola della funzione per tiparla correttamente.

$$\frac{\Gamma \vdash S : \text{void} \quad \Gamma \vdash \text{return } e; : T}{\Gamma \vdash S; \text{return } e; : T}$$

$$\frac{\Gamma \vdash S : \text{void} \quad \Gamma \vdash \text{return}; : \text{void}}{\Gamma \vdash S; \text{return}; : \text{void}}$$

La regola per il controllo dei tipi dell'*if-then-else* sarà:

$$\frac{\Gamma \vdash \text{cond} : T \quad T = \text{bool} \quad \Gamma \vdash s_1 : \text{void} \quad \Gamma \vdash s_2 : \text{void}}{\Gamma \vdash \text{if}(\text{cond}) \{s_1\} \text{ else } \{s_2\} : \text{void}}$$

Sia s_1 che s_2 verranno valutati in gamma, e quindi nello stesso ambiente, poichè l'esecuzione dei due rami è alternativa quindi l'esecuzione di uno non può interferire l'altro.

La regola per la definizione di funzione è:

$$\frac{\Gamma \cdot [f \rightarrow (T_1 x_1, \dots, T_n x_n, \text{asset } a_1, \dots, \text{asset } a_m) \rightarrow T] \vdash d : \Gamma' \quad \Gamma' \vdash s : T' \quad T = T'}{\Gamma \vdash Tf(T_1 x_1, \dots, T_n x_n)[\text{asset } a_1, \dots, \text{asset } a_m] = \{d s\} : \Gamma[f \rightarrow (T_1, \dots, T_n, \text{asset}_1, \dots, \text{asset}_m) \rightarrow T]}$$

La regola per l'invocazione di funzione e quindi per il nodo call è:

$$\frac{\Gamma \vdash f : T_1, \dots, T_n, \text{asset}_1, \dots, \text{asset}_m \rightarrow T \quad (\Gamma \vdash x_i : T'_i)^{i \in 1..n} (T'_i = T_i)^{i \in 1..n} \quad (\Gamma \vdash a_i : \text{asset})^{i \in 1..m}}{\Gamma \vdash f(x_1, \dots, x_n)[a_1, \dots, a_m] : T}$$

La regola per l'invocazione della funzione in *initcall* è simile a quella della *call*. La differenza sta nel fatto che i parametri a_1, \dots, a_m devono essere interi, poichè sono gli *asset* che vengono passati al contratto per funzionare.

$$\frac{\Gamma \vdash f : T_1, \dots, T_n, \text{asset}_1, \dots, \text{asset}_m \rightarrow T \quad (\Gamma \vdash x_i : T'_i)^{i \in 1..n} (T'_i = T_i)^{i \in 1..n} \quad (\Gamma \vdash a_i : \text{int})^{i \in 1..m}}{\Gamma \vdash f(x_1, \dots, x_n)[a_1, \dots, a_m] : T}$$

4.3 Espressioni

Tutte le espressioni ritorneranno un tipo. La prima regola è quella per la *baseExp*.

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

La regola di inferenza per la *negExp* è:

$$\frac{\Gamma \vdash \text{exp} : T \quad T = \text{int}}{\Gamma \vdash \neg \text{exp} : \text{int}}$$

La regola di inferenza per la *notExp* è:

$$\frac{\Gamma \vdash exp : T \quad T = bool}{\Gamma \vdash !exp : bool}$$

La regola di inferenza per la *derExp* è:

$$\frac{x \in dom(\Gamma) \quad proj_1(\Gamma(x)) = T}{\Gamma \vdash x : T}$$

La regola di inferenza per la *binExp* cambia a seconda dell'operatore utilizzato. Le regole per la somma, sottrazione, moltiplicazione e divisione sono:

$$\begin{array}{c} \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 + e_2 : int} \\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 - e_2 : int} \\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 * e_2 : int} \\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 / e_2 : int} \end{array}$$

Le regole per il maggiore, minore, maggiore uguale e minore uguale sono, bisogna ricontrollare però nel progetto come abbiamo fatto per gli asset e i sottotipi e aggiungere porco cane:

$$\begin{array}{c} \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 > e_2 : bool} \\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 < e_2 : bool} \\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 >= e_2 : bool} \\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = int}{\Gamma \vdash e_1 <= e_2 : bool} \end{array}$$

Le regole per l'uguaglianza e la disuguaglianza sono:

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T'}{\Gamma \vdash e_1 == e_2 : bool}$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T'}{\Gamma \vdash e_1 ! = e_2 : bool}$$

Le regole per l'and e l'or logico sono invece:

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = bool}{\Gamma \vdash e_1 \ \&\& \ e_2 : bool}$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T' \quad T = T' = bool}{\Gamma \vdash e_1 \ || \ e_2 : bool}$$

Per quanto riguarda la regola per la *CallExp*, rimandiamo alla regola dello statement call, poichè anche nel codice la *TypeCheck* di *CallExp* si limita a richiamare ricorsivamente la *TypeCheck* sul *CallNode*.

Per quanto riguarda invece *boolExp* e *valExp* le due regole di inferenza sono degli assiomi e sono:

$$\overline{\Gamma \vdash boolean : bool}$$

$$\overline{\Gamma \vdash integer : int}$$

Capitolo 5

Analisi degli Effetti

Dopo aver effettuato la check semantics e la type check effettueremo un'altra visita dell'albero per effettuare un'analisi degli effetti. La nostra analisi degli effetti dovrà effettuare il controllo della liquidity di un contratto. Con più precisione dovrà controllare che:

- a fine di una funzione i parametri formali asset devono essere svuotati
- controllare il flusso degli asset (i valori degli asset non devono essere persi)
- a fine del contratto tutti i parametri globali asset devono essere svuotati

Per effettuare questo controllo definiamo un ambiente Σ , che sarà una funzione così definita:

$$\Sigma = ID \rightarrow L$$

Dove l'insieme L è così definito:

$$L = \{\emptyset, \mathbf{1}\}$$

L'ordinamento parziale per i simboli dell'insieme L per il controllo della liquidity è:

$$\emptyset \leq \mathbf{1}$$

Dove \emptyset rappresenta un asset vuoto e $\mathbf{1}$ rappresenta un asset pieno. Definiamo ora una funzione \oplus così fatta:

\oplus	\emptyset	$\mathbf{1}$
\emptyset	\emptyset	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

Definiamo ora le regole di inferenza per il controllo degli effetti. La prima regola è la *dichiarazione dei parametri globali asset*:

$$\overline{\Sigma \vdash \text{asset } x : \Sigma[x \rightarrow \emptyset]}$$

Questa regola sarà un assioma poichè non abbiamo bisogno di controllare che l'identificatore non sia già dichiarato dato che questo tipo di controllo è stato effettuato nella fase di analisi semantica.

Definiamo ora la regola di inferenza per la funzione *move*.

$$\frac{\Sigma(a_1) = \sigma_1 \quad \Sigma(a_2) = \sigma_2}{\Sigma \vdash a_1 - o \ a_2 : \Sigma[a_1 \rightarrow \emptyset, a_2 \rightarrow \sigma_1 \oplus \sigma_2]}$$

Nel progetto, la funzione \oplus sarà sostituita invece dalla funzione `Math.max()` di Java.

La regola di inferenza per la funzione *transfer* sarà invece:

$$\overline{\Sigma \vdash \text{transfer } a : \Sigma[a \rightarrow \emptyset]}$$

La regola per l'*If Then Else* sarà:

$$\frac{\Sigma \vdash \text{Cond} : \Sigma' \quad \Sigma' \vdash S_1 : \Sigma_1 \quad \Sigma' \vdash S_2 : \Sigma_2}{\Sigma \vdash \text{if}(\text{cond})\{S_1\}\text{else}\{S_2\} : \max(\Sigma_1, \Sigma_2)}$$

Si andrà a valutare la condizione che restituirà un nuovo ambiente (nel caso in cui questa sia una chiamata a funzione) e in questo nuovo ambiente ottenuto andremo a valutare i due rami then ed else e restituiremo il massimo tra i due ambienti.

$$\frac{\Sigma \vdash \text{Cond} : \Sigma' \quad \Sigma' \vdash S_1 : \Sigma_1}{\Sigma \vdash \text{if}(\text{cond})\{S_1\} : \max(\Sigma_1, \Sigma')}$$

Sarà necessaria una seconda regola di inferenza per il caso in cui il nostro if non avrà il ramo else. In questo caso particolare restituiremo il massimo tra l'ambiente ottenuto dopo la valutazione del ramo then e quello senza la valutazione del ramo stesso.

Per il nodo call la regola di inferenza sarà:

$$\frac{\Sigma' = \Sigma \cdot [a_i \rightarrow \Sigma(u_i)]^{i \in 1..m} \quad \Sigma'[u_i \rightarrow \emptyset]^{i \in 1..m} \quad \Sigma' \vdash \{D \ S\} : \Sigma''}{\Sigma \vdash f(x_1, \dots, x_n)[u_1, \dots, u_m] : \Sigma''}$$

In questa regola avremo che u_1, \dots, u_m saranno i parametri attuali della chiamata. Si creerà un nuovo scope in cui verranno definiti i parametri a_1, \dots, a_m che saranno i parametri formali della funzione, ai quali verrà assegnato il valore della Liquidity dei parametri attuali u_i . Successivamente i parametri

attuali u_i verranno svuotati, quindi il valore della loro Liquidity sarà pari a \emptyset . In questo nuovo ambiente si andrà a valutare il corpo della funzione tramite una lookup per id nella STentry che conterrà il nodo FuncionNode da cui ricaveremo gli statement del corpo.

Per il nodo initcall la regola di inferenza sarà:

$$\frac{\Sigma \vdash (e_i = \sigma_i)^{i \in 1..m} \quad \Sigma' = \Sigma \cdot [a_i \rightarrow \sigma_i]^{i \in 1..m} \quad \Sigma' \vdash \{D \ S\} : \Sigma''}{\Sigma \vdash f(x_1, \dots, x_n)[e_1, \dots, e_m] : \Sigma''}$$

Come si può notare, questa regola è simile a quella per la call. La differenza principale sta nel fatto che i parametri attuali asset sono delle espressioni. Si andranno, prima di tutto, a valutare queste espressioni e otterremo un valore di Liquidity che assegneremo ai parametri formali. Non sarà poi necessario svuotare i parametri attuali asset in quanto non sono variabili asset ma espressioni.

La regola per la valutazione del corpo della funzione sarà invece:

$$\frac{\begin{array}{l} \Sigma = [(w_j \rightarrow \sigma_j)^{j \in 1..k}] \cdot [(a_i \rightarrow \sigma_{k+i})^{i \in 1..m}] \\ \Sigma \vdash D : \Sigma \quad \Sigma \vdash S : \Sigma' \\ \Sigma' = \Sigma'_0 \cdot \Sigma'_1 \quad (\Sigma'_1(a_i) = \emptyset)^{i \in 1..m} \end{array}}{\Sigma \vdash \{D \ S\} : \Sigma'_0}$$

In questa regola definiamo inanzitutto com'è fatto l'ambiente Σ , che sarà una concatenazione di due ambienti: quello di testa che conterrà i parametri formali a_i asset e quello più interno che conterrà i parametri globali w_j asset. Come si può notare, valuteremo le dichiarazioni nel corpo della funzione, ma dal momento in cui non si tratta di variabili asset, restituiremo l'ambiente così com'è. Questo varrà anche per tutti gli altri nodi dell'albero per i quali non abbiamo definito una regola. Dopo la valutazione degli statement, controlleremo che tutti i parametri formali siano svuotati e torneremo un ambiente aggiornato escludendo l'ambiente di testa in Σ' , che è quello che conteneva i parametri formali asset.

Questa regola non vale però nel caso in cui ci sia una chiamata ricorsiva. In questo caso avremo bisogno di utilizzare il metodo del punto fisso. Per mantenere la monotonia all'interno del metodo del punto fisso faremo partire i parametri formali asset a $\mathbb{1}$ per assicurarsi la discesa:

$$\frac{\begin{array}{l} \Sigma = [(w_j \rightarrow \sigma_j)^{j \in 1..k}] \cdot [(a_i \rightarrow \mathbb{1})^{i \in 1..m}] \\ \Sigma \vdash D : \Sigma \quad \Sigma \vdash S : \Sigma' \\ \Sigma' = \Sigma'_0 \cdot \Sigma'_1 \quad (\Sigma'_1(a_i) = \emptyset)^{i \in 1..m} \end{array}}{\Sigma \vdash \{D \ S\} : \Sigma'_0}$$

Capitolo 6

Generazione di codice

L'ultima fase di questo progetto si occupava di andare a generare il codice bytecode per il codice assetlan preso in input e di costruire l'interprete per l'esecuzione del bytecode generato. Siamo così andati in ogni nodo dell'albero a scrivere la funzione *codGeneration()*. Ogni nodo restituirà una stringa e l'insieme di tutte queste stringhe, sarà il nostro codice bytecode da eseguire tramite l'interprete. Abbiamo definito una classe *Instruction* che conterrà un intero, il codice del token rappresentante l'istruzione da eseguire, e tre stringhe che rappresentano i tre possibili argomenti che quell'istruzione prende in input. Se un argomento non c'è (poichè esistono istruzioni che hanno meno di tre argomenti o addirittura nessuno come la *pop*) questo verrà settato a *null*.

I registri utilizzati sono:

- **Instruction Pointer (\$ip):** indica la prossima istruzione da eseguire
- **Stack pointer (\$sp):** punta alla cima dello stack
- **Frame pointer (\$fp):** punta all'access link corrente relativo al frame attivo
- **Access link (\$al):** registro utilizzato per attraversare la catena statica degli scope
- **Accumulatore (\$a0):** registro utilizzato per salvare il valore calcolato da alcune espressioni
- **Registro generico (\$a1):** utilizzabile liberamente all'interno del programma
- **Return address (\$ra):** registro utilizzato per salvare l'indirizzo di ritorno una volta usciti da un record di attivazione.

- **Balance (\$b)**: registro utilizzato per tenere traccia della somma delle transfer effettuate durante tutto il programma

Per ogni istruzione letta viene creato un oggetto di tipo `Instruction`, definito dalla classe `Instruction.java`. La CPU esegue le istruzioni passate al costruttore dall'interprete. Dopodichè, ad ogni iterazione, si accede all'istruzione indicata dal registro `$ip` che poi viene incrementato di uno per passare ad eseguire l'istruzione successiva. L'operazione da eseguire dipende dal campo `code` dell'oggetto `Instruction`. Per gestire i registri si utilizzano le funzioni `regStore` e `regRead` per aggiornare e leggere il valore contenuto nel registro opportuno. La memoria è stata semplicemente implementata come un array di interi poichè non c'è bisogno di memorizzare altre informazioni.

Durante l'esecuzione del programma a seguito delle chiamate a funzione si vengono a creare dei *Record di attivazione*. La struttura del nostro record di attivazione sarà quella visibile in figura. I record di attivazione cresceranno dal basso verso l'altro, quindi il vecchio frame pointer si troverà ad indirizzi di memoria più alti del return address. Dopo aver caricato il vecchio frame pointer caricheremo prima tutti gli asset in ordine, da 1 a n. Il caricamento degli asset avviene in ordine poichè la loro dichiarazione (per mantenere coerenza con gli offset per l'accesso alle variabili nel record di attivazione) avviene al contrario. In questo modo possiamo caricare gli asset da 1 a n e conseguentemente svuotarli per mantenere la semantica di $f()[a,a]$, in cui il primo parametro attuale sarà passato al primo formale e poi svuotato ed il secondo parametro formale riceverà un asset vuoto. Successivamente carichiamo i parametri della funzione al contrario da m a 1 e le dichiarazioni interne alle variabili sempre in ordine inverso. Infine, ci sarà il vecchio access link ed il vecchio *\$ra*.

Return Address
Old Access Link
Dec 1
...
Dec k
Param 1
...
Param m
Asset n
...
Asset 1
Vecchio FP

6.1 Interprete

Il funzionamento del nostro interprete è governato dalla classe *ExecuteSVM*. Questa classe conterrà la nostra memoria ed il nostro codice, come detto in precedenza rispettivamente un array di interi ed un array di `Instruction` ed

i nostri registri. Lo *stack pointer* ed il *frame pointer* verranno inizializzati alla grandezza della memoria meno uno, il registro *balance* ed il registro *instruction pointer* verranno inizializzati a 0 mentre *return address* e *access link* non sono inizializzati. Ci sarà inoltre un array *a* di dieci elementi che conterrà i registri temporanei e l'accumulatore.

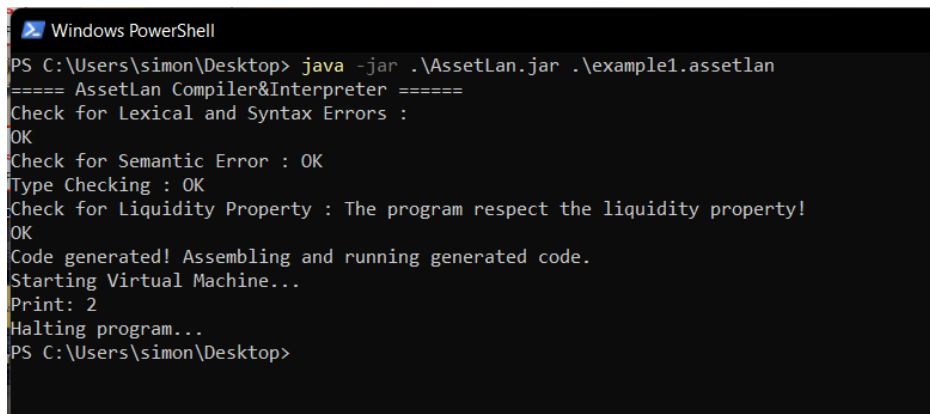
Al momento dell'esecuzione viene chiamato il metodo *cpu()* della classe *ExecuteSVM* che all'interno di un ciclo infinito scorre tutte le istruzioni caricate tramite uno switch. Le condizioni di terminazione sono due:

- *Out of memory*: se il nostro stack pointer supera la memoria a disposizione (quindi se lo sp raggiunge il valore zero).
- *Halt del programma*: se si raggiunge un'istruzione di halt che fa terminare il programma.

Capitolo 7

Compilatore AssetLan

Il compilatore può essere eseguito da terminale tramite il comando `java -jar AssetLanCompiler.jar [*assetlan]`.



```
Windows PowerShell
PS C:\Users\simon\Desktop> java -jar .\AssetLan.jar .\example1.assetlan
===== AssetLan Compiler&Interpreter =====
Check for Lexical and Syntax Errors :
OK
Check for Semantic Error : OK
Type Checking : OK
Check for Liquidity Property : The program respect the liquidity property!
OK
Code generated! Assembling and running generated code.
Starting Virtual Machine...
Print: 2
Halting program...
PS C:\Users\simon\Desktop>
```

Il compilatore si occupa del controllo degli errori lessicali e semantici, controllo di tipo, e se il programma rispetta la proprietà di Liquidity. Una volta eseguiti questi controlli, verrà generato un file `*.asm` contenete le istruzioni che verranno prese in input dall'interprete, il quale eseguirà le istruzioni generate.