Report Progetto Compilatori ed Interpreti

Marco Benito Tomasone 1038815
Luca Genova 1038843
Simone Boldrini 1038792

2021-2022

Indice

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 *decp* 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() + "
```

```
                at line " + i.getLine() + "\n";
    }
...
}
```

Nel main del nostro compilatore, inziamo 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.

```
        CharStream charStreams = CharStreams.
           fromFileName(fileName);
        AssetLanLexer lexer = new AssetLanLexer(
           charStreams);
        CommonTokenStream tokens = new CommonTokenStream
           (lexer);
        AssetLanParser parser = new AssetLanParser(
           tokens);

        SyntaxErrorListener errorListener = new
           SyntaxErrorListener();
        lexer.addErrorListener(errorListener);
        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.
.

```
        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]{
        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 dichiando come riconosce l'<EOF> invece di trovare un'ulteriore dichiarazione a funzione o la chiamata ad una funzione.

```
int a ;
asset x ;
void f(int n)[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 ;
}
```

```
--------LEXICAL ERRORS----------

--------SYNTAX ERRORS----------
Error: mismatched input ';' expecting '[' at
   line 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*. 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 symbolTable che si trova in testa e diminuito il nestingLevel. In questa classe sono anche presenti alcune funzioni per effettuare una lookup all'interno della symbolTable e per aggiungere delle dichiarazioni all'interno della tabella dei simboli.

3.2 STentry

Questa classe viene utilizzata dall'Environment per costruire una entry di una variabile all'occorrenza ed aggiungerla al proprio scope. Contiene il tipo ed il numero di argomenti.

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(
       Environment e) {
8      ArrayList<SemanticError> res = new ArrayList<
           SemanticError>();
9      if(e.isDeclared(id1.getId()) == EnvError.NO_DECLARE)
           {
10         res.add(new SemanticError((id1.getId())+": is
               not declared [Move]"));
11     }
12     if(e.isDeclared(id2.getId())== EnvError.NO_DECLARE){
13         res.add(new SemanticError((id2.getId())+": is
               not declared [Move]"));
14     }
15     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:

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

Associerà quindi un nome ad un tipo, ad un valore di una liquidity e ad un offset. Gli insiemi T ed L sono così definiti:

$$T = \{\text{void, asset, bool, int}\}$$
$$L = \{\bot, \emptyset, \mathbb{1}\}$$

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

$$\bot \leq \emptyset \leq \mathbb{1}$$

Dove $\perp$ rappresenta le variabili non asset, $\emptyset$ rappresenta un asset vuoto e $\mathbb{1}$ rappresenta un asset pieno. Definiamo inoltre una funzione $proj_i$ che ci permetterà di ritornare l' *i-esimo* elemento di Γ(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.

$$[\text{prog}]\ \frac{\emptyset, 0 \vdash field : \Gamma\ \Gamma \vdash asset : \Gamma'\ \Gamma' \vdash function : \Gamma''\ \Gamma'' \vdash initcall : T}{\emptyset, 0 \vdash \text{field asset function initcall} : max(proj_2(\Gamma''))}$$

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

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

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

Capitolo 5

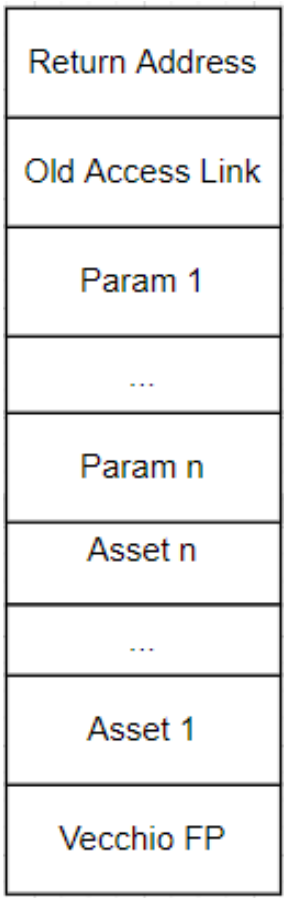
Generazione di codice

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 frame

- **Balance ($b):** registro utilizzato per tenere traccia della s
  La 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.



Per le chiamate di funzione si verranno a creare dei record di attivazione. I nostri record di attivazione avranno una crescita dal basso verso l'alto, quindi guardando l'immagine possiamo dire che il vecchio frame pointer sarà ad un indirizzo di memoria maggiore rispetto al Return Address. Gli asset vengono caricati in ordine da 1 a n poichè durante il loro caricamento vengono svuotati. Per garantire una compatibilità con l'accesso all'offset durante la dichiarazione questi vengono dichiarati a partire dalla fine, quindi l'asset n avrà un offset più piccolo rispetto all'asset 1. I parametri invece vengono dichiarati in ordine e di conseguenza verranno caricati al contrario. Salveremo poi nel RdA il vecchio access link ed il return address.

5.1 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 pointed* 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