Report Progetto Compilatori ed Interpreti

Marco Benito Tomasone 1038815
Luca Genova
Simone Pio 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 *Asset Lan*. 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 4 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.

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*.

```
1
2  public void syntaxError(...){
3  ...
4      while(errors.hasNext()) {
5          SyntaxError i = (SyntaxError)errors.next();
6          if(i.getRecognizer().getClass().getSimpleName().
               equals("AssetLanLexer"))
7              lexicalErrors += "Error: " + i.getMsg() + "
                   at line " + i.getLine() + "\n";
8          else
9              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] ?
transfer @x ;
```

```
}
main(~)[1] ;
```

```
--------LEXICAL
   ERRORS----------
Error: token
   recognition
   error at: '$' at
    line 1
Error: token
   recognition
   error at: '$' at
    line 2
Error: token
   recognition
   error at: '@' at
    line 8
Error: token
   recognition
   error at: '~' at
    line 10

--------SYNTAX
   ERRORS----------
```

In questo secondo esempio non abbiamo inserit errori lessicali ma abbiamo inserito le dichiarazioni degli asset e le dichiarazioni di 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.

```
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] ;
}
asset x, y ;
int a,b = 3;
main()[2,3]
```

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

--------SYNTAX
   ERRORS----------
Error: extraneous
   input 'asset'
   expecting {'void
   ', 'int', 'bool
   ', ID} at line
   10
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.

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

2.2 AST

Per la costruzione dell'Albero di Sintassi astratta sono state implementate tutte le funzioni necessarie all'attraversamento dell'albero nel file *AssetLanVisitorImpl*.

Capitolo 3

Analisi Semantica

3.1 Tabella dei simboli

In questa fase della costruzione del nostro compilatore ci siamo concentrati sulla costruzione della tabella dei simboli. Per generare la tabella dei simboli è necessario visitare l'AST, per farlo abbiamo definito tutte le funzioni necessarie all'attraversamento dell'albero nel file *AssetLanVisitorImpl*.
La tabella dei simboli è stata implementata tramite una Lista di HashMap e i metodi per interagire con essa sono definiti nella classe *environment*.