

# The APLA Language

---

ANUJ THULA, PARAM MEHTA, LUIS QUINTANILLA, SAI ADITYA BODAVALA

TEAM 31,

SER 502,

ARIZONA STATE UNIVERSITY



# Features

---

Here are some of the Features we implemented in our language:

Datatypes: Integer

Decisions: If- then - else.

Operators:

- Assignment: =
- Arithmetic: +, -, \*, /
- Logical: and, or

Console write: “println”.

# Interpreter Used

---

We have used the “ANTLR” interpreter in our language.

In computer-based language recognition, ANTLR (pronounced Antler), or Another Tool For Language Recognition, is a parser generator that uses LL(\*) for parsing. ANTLR is a robust framework that has been developed for over 25 years and is used in various proven technologies such as Hadoop, Hive etc.(<https://en.wikipedia.org/wiki/ANTLR>)

ANTLR is the successor to the Purdue Compiler Construction Tool Set (PCCTS), first developed in 1989, and is under active development. Its maintainer is Professor Terence Parr of the University of San Francisco.

We will go into detail on the installation and use of this interpreter further. [1]

## **Terminal code to compile your grammar using antlr:**

```
Java -jar ../lib/antlr.jar -package de.letsbuildacompiler.parser -o ../src/de/letsbuildacompiler/parser -no-listener -visitor Demo.g4
```

# Design

---

- Datatypes
  - Support for integers
  - Keywords support for int, If - then - else, println.
- Decision
  - If - then - else support.
  - Also support for if.
- Exception Handling
  - Integer exception handling.
  - Variable, String exception handling.
- Method
  - User can also use methods in the program.
  - File Input and Assignment:
    - The user input can also be accepted using a .apla file.

# Run-Time

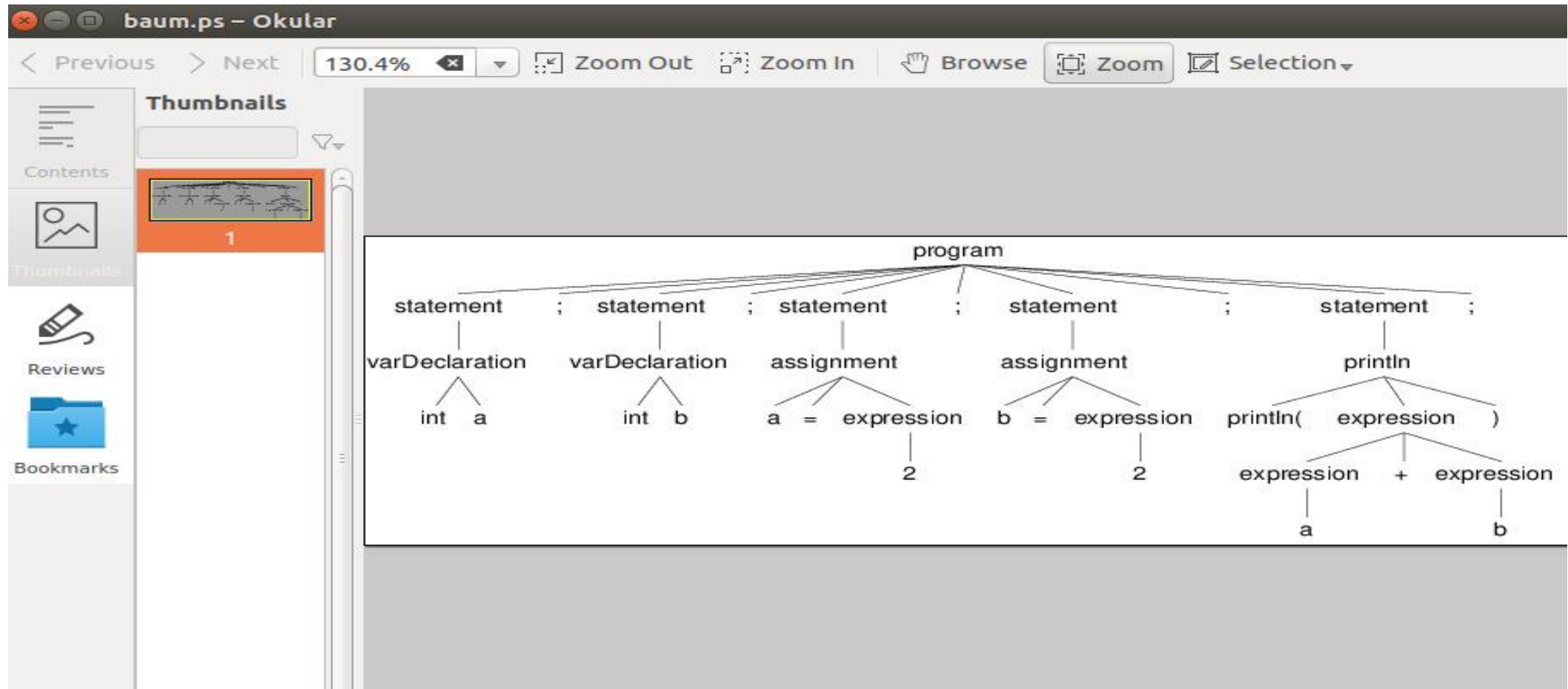
---

- So for our language we have implemented a bottom-up parsing technique design which is shown by the diagram below:
- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.
- The input would be a parse tree.
- **Terminal code to generate Parse Tree:**

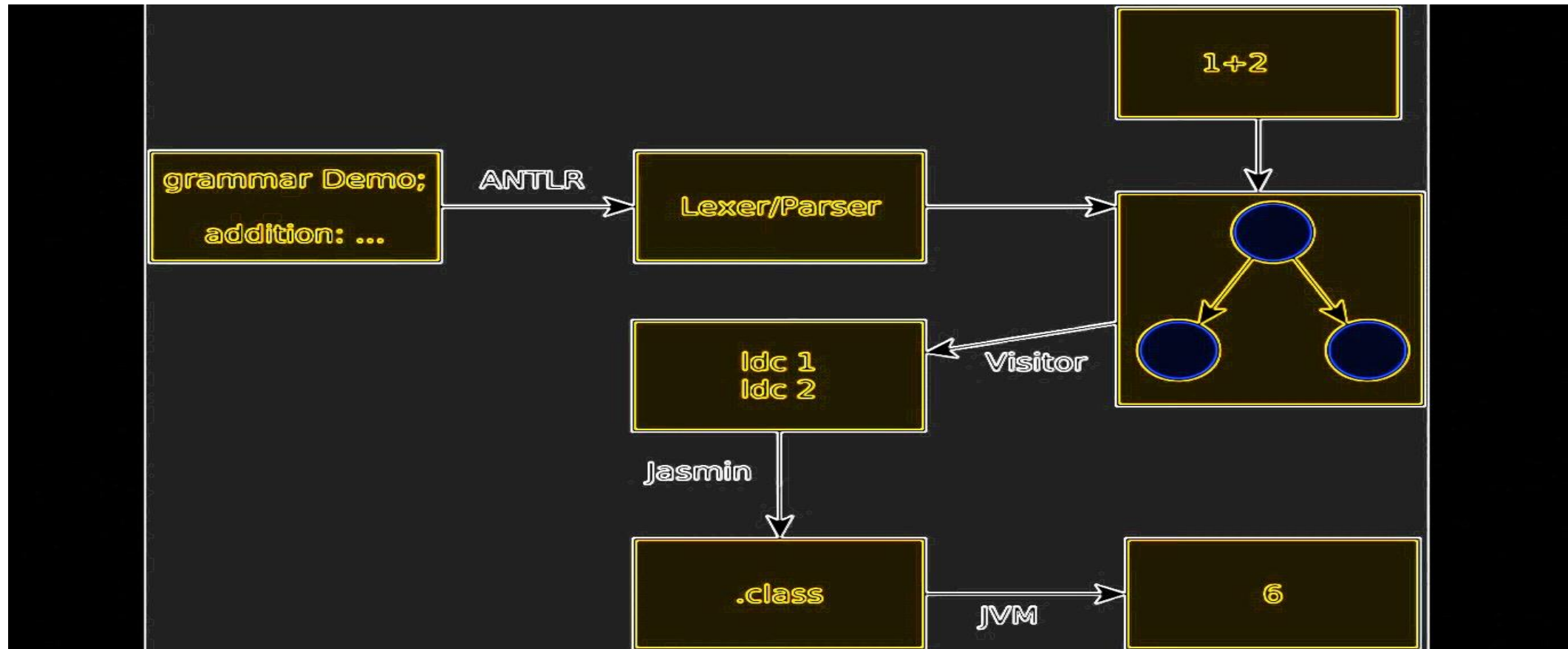
```
java -cp bin:lib/antlr.jar org.antlr.v4.runtime.misc.TestRig de.letsbuildacompiler.parser.Demo program -ps /tmp/baum.ps ../Compiler/code.demo
```

```
okular /tmp/baum.ps (make sure you install okular)
```

# ANTLR Generated Parse Tree



# Design Diagram



# Grammar

The Grammar we used is in the **EBNF**(Extended Backus–Naur form) so it can be implemented with **Antlr** to generate our parser

---

program

: programPiece+ ;

programPiece

: statement #StatementPiece

| method #MethodPiece

;

statement

: println ';' ;

| varAssignment ';' ;

| assignment ';' ;

| branch

;

branch

: 'if' '(' condition=expression ')' True=section 'else' False=section ;

section

: '{' statement\* '}' ;

expression

: left=expression operator=('\*' | '/') right=expression #MULTDIV

| left=expression operator=('+' | '-') right=expression #PLUSMINUS

| num=NUM #Number

| varName=NAME #Variable

| methodCall #MethodExp

;

assignment: varName=NAME '=' expr=expression ;



# Grammar

---

varAssignment

: 'int' varName=NAME;

println

: 'println(' argument=expression ')';

while\_statement

: 'WHILE' expression\_condition '{' block '}'

method

: 'int' methName=NAME '(' ')' '{' statements=statementList 'return' returnVal=expression  
';' '}' ;

statementList: statement\* ;

methodCall

: methName=NAME '(' ')';

NAME

: [a-zA-Z][a-zA-Z0-9]\*;

NUM

: [0-9]+;

WHITESPACE

: [ \t\n\r]+ -> skip;

# Lexical Analyzer

---

The Lexical analyzer we used is called ANTLR.

. ANTLR (code):

ANTLR reads grammar file to parse input

Generates a list of .java files that enable access to the elements of input (similar to a parse tree)

.Tokens

Every unique character tokenized and added to the “stack”

.Lexer

Uses the tokens with the generated set of rules that were defined

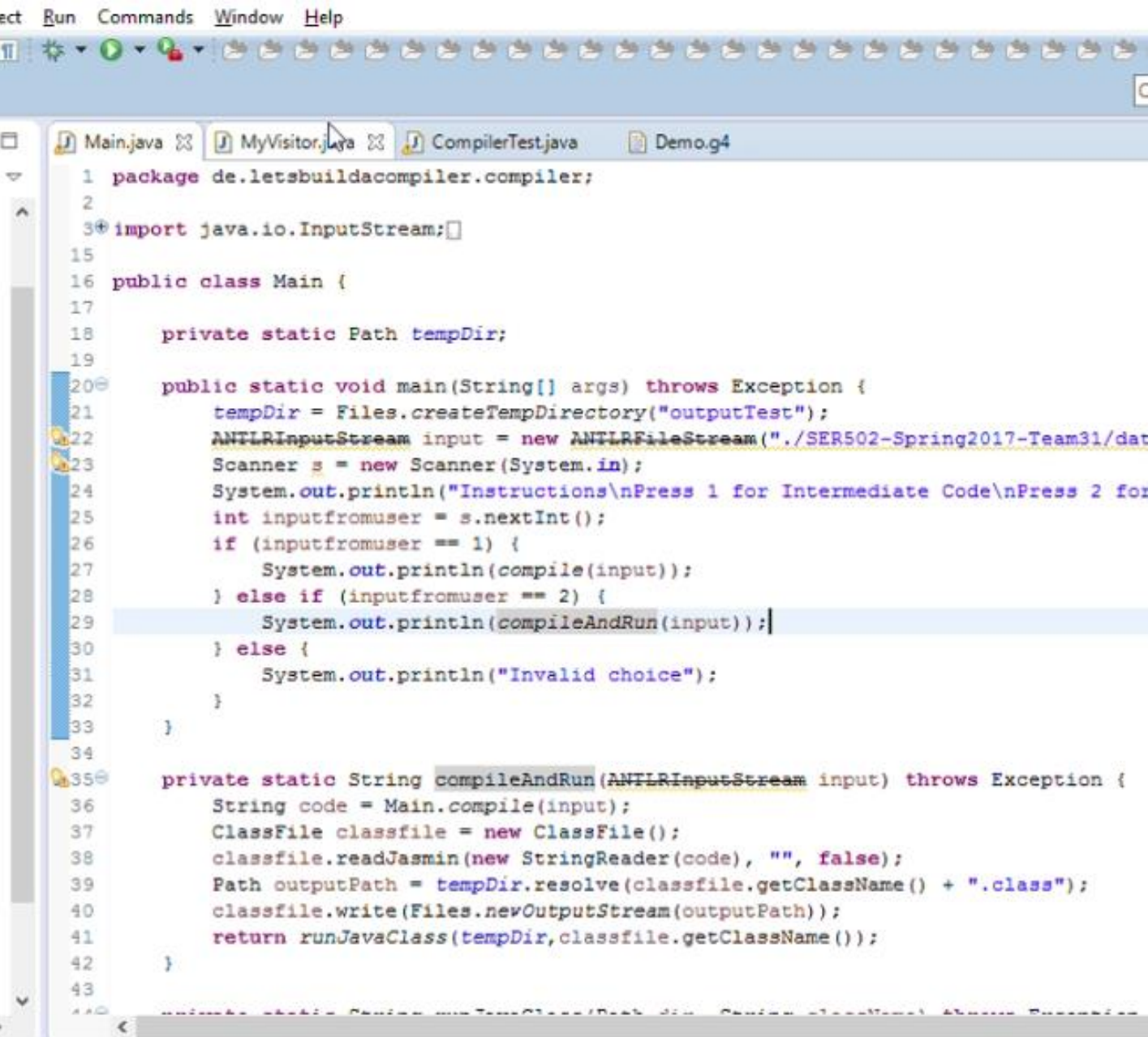
.Parser

Parses the code using the lexer and tokens as rules and generates methods based on the tokens and variables

.Visitor (interface)

Overridable functions are made here. These can be later overridden in MyVisitor.Java

.BaseVisitor



```
1 package de.letsbuildacompiler.compiler;
2
3 import java.io.InputStream;
4
5
6
7
8
9
10
11
12
13
14
15
16 public class Main {
17
18     private static Path tempDir;
19
20     public static void main(String[] args) throws Exception {
21         tempDir = Files.createTempDirectory("outputTest");
22         ANTLRInputStream input = new ANTLRFileStream("../SER502-Spring2017-Team31/dat
23         Scanner s = new Scanner(System.in);
24         System.out.println("Instructions\nPress 1 for Intermediate Code\nPress 2 for
25         int inputfromuser = s.nextInt();
26         if (inputfromuser == 1) {
27             System.out.println(compile(input));
28         } else if (inputfromuser == 2) {
29             System.out.println(compileAndRun(input));
30         } else {
31             System.out.println("Invalid choice");
32         }
33     }
34
35     private static String compileAndRun(ANTLRInputStream input) throws Exception {
36         String code = Main.compile(input);
37         ClassFile classfile = new ClassFile();
38         classfile.readJasmin(new StringReader(code), "", false);
39         Path outputPath = tempDir.resolve(classfile.getClassName() + ".class");
40         classfile.write(Files.newOutputStream(outputPath));
41         return runJavaClass(tempDir, classfile.getClassName());
42     }
43
44     private static String runJavaClass(Path dir, String className) throws Exception {
45         Process process = Runtime.getRuntime().exec("java " + className);
46         process.waitFor();
47         return process.getInputStream().readAllBytes().toString();
48     }
49 }
```

# Intermediate Code:

- We used Java to get our intermediate code:
- In Main.java,
  - ‘Compile’ is used to get the Intermediate code.
  - ‘Compile and Run’ is used to get output code.

# Intermediate Code Example:

```
class public HelloWorld
```

```
.super java/lang/Object
```

```
.method public static main([Ljava/lang/String;)V
```

```
.limit stack 100
```

```
.limit locals 100
```

```
ldc 4
```

```
istore 0
```

```
ldc 16
```

```
istore 1
```

```
iload 0
```

```
iload 1
```

```
iload 1
```

```
imul
```

```
iload 0
```

```
idiv
```

```
iadd
```

```
iload 1
```

```
isub
```

```
istore 2
```

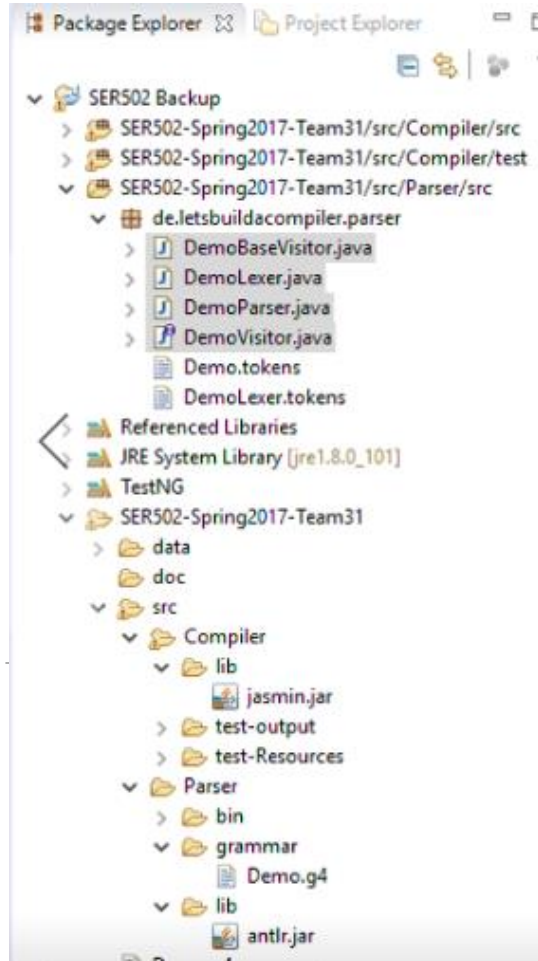
```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

```
iload 2
```

```
invokevirtual java/io/PrintStream/println(I)V
```

```
return
```

```
.end method
```



- We used `jasmin` to make class files for our intermediate code.
- `Jasmin` is an assembler for the java virtual machine. It takes ASCII descriptions of java classes, written in a simple assembler-like syntax using the java virtual machine instruction set. It converts them into binary java class files, suitable for loading by a java runtime system.
- It is contained in `src/compiler/lib/jasmin.Jar` [2]

# References:

---

[1] ANTLR: <http://www.antlr.org/>

[2] Jasmin: <http://jasmin.sourceforge.net/>

Both tools were taken from these sources and installed to Eclipse accordingly.

Eclipse: <https://eclipse.org/>