Compiler to JVM

Jiawei Chen, Zhetuo Qi
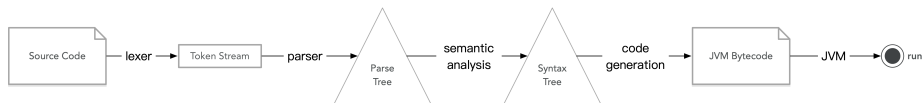
Zhejiang University

May 29, 2019

Language Specification

Example

```
1  namespace Test {
2      struct MyStruct {
3          int a = f();
4      }
5
6      double d = 4;
7
8      def void main(String[] args) {
9          print("Hello world!\n");
10     }
11
12     def int f() {
13         return 3;
14     }
15 }
```

Compiler Architecture

Source Code → **lexer** → Token Stream → **parser** → Parse Tree → semantic analysis → Syntax Tree → code generation → JVM Bytecode → **JVM** → ● run

Platform & Dependency

- Java 8
- Gradle
- COMMONS-CLI: parse the command line
- ANTLR 4: lexical/syntactic analysis
- ASM: help to generate Java bytecode

Lexical Analysis

With the help of ANTLR, lexical analysis can be easy:

Lexical Rules Example

```
1 WHITE_SPACE: [ \t\r\n]+ -> skip;
2 LINE_COMMENT: '//' r\n]* -> skip;
3 CHAR_LITERAL: '\'' [a-zA-Z\\] '\'';
```

Syntactic Analysis

By ANTLR, we can write the following parsing rules. All the rules are placed in a .g4 file.
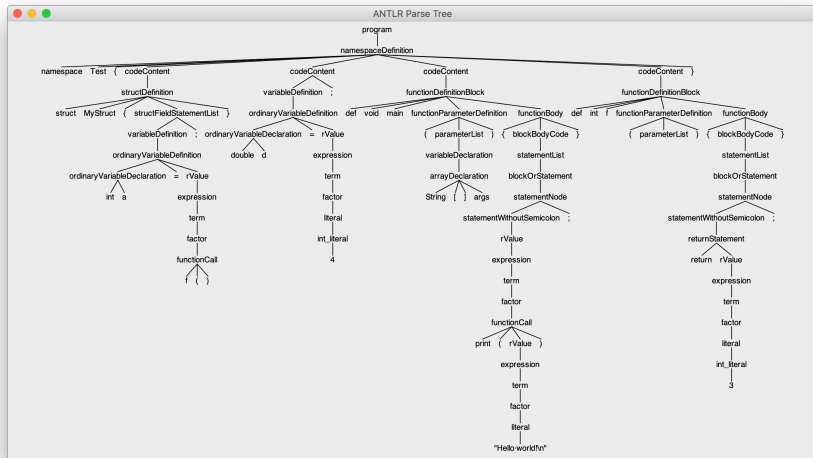
Parsing Rules Example

```
1 namespaceDefinition:
2     NAMESPACE_SYMBOL IDENTIFIER LEFT_CURLY_BRACE codeContent+
          RIGHT_CURLY_BRACE;
```

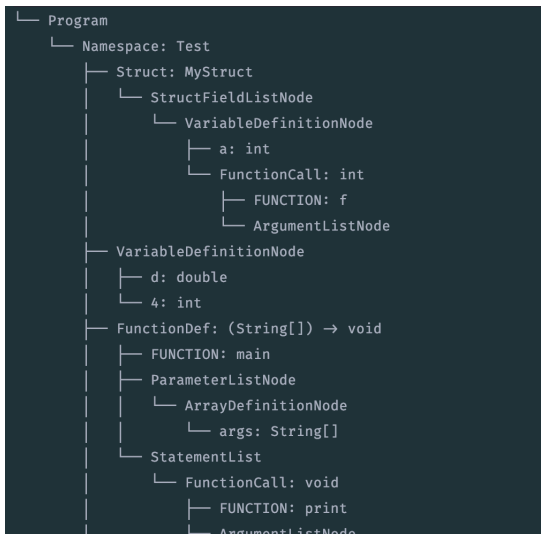**Note** : ANTLR doesn't support left-recursive, we have to solve it manually.

Syntactic Analysis

Result of Syntactic Analysis:

Semantic Analysis

Syntax Tree:

```
└── Program
    └── Namespace: Test
        ├── Struct: MyStruct
        │   └── StructFieldListNode
        │       └── VariableDefinitionNode
        │           ├── a: int
        │           └── FunctionCall: int
        │               ├── FUNCTION: f
        │               └── ArgumentListNode
        ├── VariableDefinitionNode
        │   ├── d: double
        │   └── 4: int
        ├── FunctionDef: (String[]) → void
        │   ├── FUNCTION: main
        │   ├── ParameterListNode
        │   │   └── ArrayDefinitionNode
        │   │       └── args: String[]
        │   └── StatementList
        │       └── FunctionCall: void
        │           ├── FUNCTION: print
        │           └── ArgumentListNode
```

Code Generation

■ Map from language to JVM inner structure:

Structure in this language	Corresponding structure in JVM
Namespace	Class
Struct	Inner class
Fields in Struct	Fields in inner class
Variable declared in namespace	Static fields in class
function	Static method in class
other	Same as JVM

Code Generation

■ Equivalent code between this language and Java:

```
namespace Test {
    struct MyStruct {
        int a = f();
    }

    double d = 4;

    def void main(String[] args) {
        print("Hello world!\n");
    }

    def int f() {
        return 3;
    }
}
```
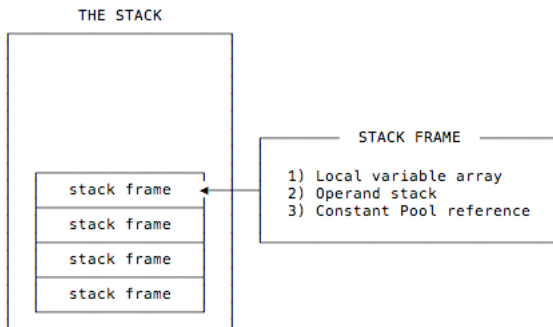
→

```
public class Test {
    public static class MyStruct {
        public int a = f();
    }

    public static double d = 4;

    public static void main(String[] args) {
        System.out.println("Hello world!\n");
    }

    public static int f() {
        return 3;
    }
}
```

Code Generation

■ Format of JVM .class file

Magic	Version
Constant Pool	
Access Flags	
this Class	
super Class	
Interfaces	
Fields	
Methods	
Attributes	

Code Generation

- JVM is a stack machine
- JVM assembly language is a kind of P-Code

THE STACK

STACK FRAME

1) Local variable array
2) Operand stack
3) Constant Pool reference

stack frame

stack frame

stack frame

stack frame

Code Generation

Pseudocode of for-loop

```
1 for (initStatements; condition; stepStatements) { }
```

JVM assembly code of for-loop

```
1     //do initStatements
2 loop_label:
3     //push condition
4     ifeq end_label
5     //BlockCode
6 continue_label:
7     //stepStatements
8     goto loop_label
9 end_label:
```