Course Unit Compilers

Masters in Informatics and Computing Engineering (MIEIC)

Department of Informatics Engineering

University of Porto/FEUP

2º Semester - 2020/2021

Compiler of the Java-- language to Java Bytecodes

v1.0, February 26, 2021

Table of Contents

Objectives: To learn and to apply the knowledge acquired in the Compilers course unit by building a compiler for programs in a representative language. The compiler shall generate valid JVM (*Java Virtual Machine*) instructions in the *jasmin* format, which the *jasmin* assembler translates to Java bytecodes.

---

1. Introduction and Assessment

The intention of this assignment is to acquire and apply knowledge within the Compilers' course unit . In order to do so, you will learn how to develop a mini-compiler. The project is split into stages that essentially correspond to steps in the flow of a real compiler. It is natural and expected that during this project students face some of the problems that may be present when developing a commercial compiler. The stages of the project allow the students to manage the efforts and dedication to the project according to the level they would like to achieve.

The assignment score is distributed by the following parameters:

i. 10% given to the organization and clarity of the code developed, as well as the documentation;
ii. 20% given to the solutions implemented to solve potential problems;
iii. 20% given to the optimization level performed by the compiler;
iv. 50% according to the results obtained by the functional tests in the two test suites provided:
    a. 30% obtained given the results when using the test suite given to the students to test the compiler during development;
    b. 20% obtained given the results obtained when using the test suite, which is not known *a priori*.

It is expected that each group of students is able to manage and balance the dedication of each member to the project. The work distribution must be reported to the professors and documented in the final report.

Grades of the members within a group are typically equal. Different grades indicate potential issues, such as the distribution of the workload, the group member's commitment, and the capability of explaining the work developed.

2. The jmm Compiler

The compiler, named *jmm*, must translate programs in **Java--**[1]**,** version 0.1 (see Section 3), into java *bytecodes* [1]. Figure 1 shows the compilation flow of the compiler. The compiler must

[1] Based on the MiniJava proposed in Appel and Palsberg's "Modern Compiler Implementation in Java", 2nd Edition, pages 484-486

generate files of the classes with JVM instructions accepted by *jasmin* [2], a tool that translates those classes in Java *bytecodes* (classfiles).

The proposed **Java--** is compatible with **Java**, and can be integrated in a Java application. Classes that have been compiled to bytecode can be used from **Java--** code (the utility of this concept will be detailed later in this document).

The **jmm** compiler is executed using the following notation:

```
java jmm [-r=<num>] [-o] <input_file.jmm>
```

or

```
java –jar jmm.jar [-r=<num>] [-o] <input_file.jmm>
```

Where <*input_file.jmm*> is the **Java--** class we would like to compile.

The "–r" option tells the compiler to use only the first <*num*>[2] local variables of the JVM when assigning the local variables used in each **Java--** function to the local JVM variables. Without the "–r" option (similar to –r=0), the compiler will use the available JVM local variables to store the local variables used in each **Java--** function.

With the "–o" option, the compiler should perform a set of code optimizations.

Section 4 (on page 7) details the "-r" and "-o" options.

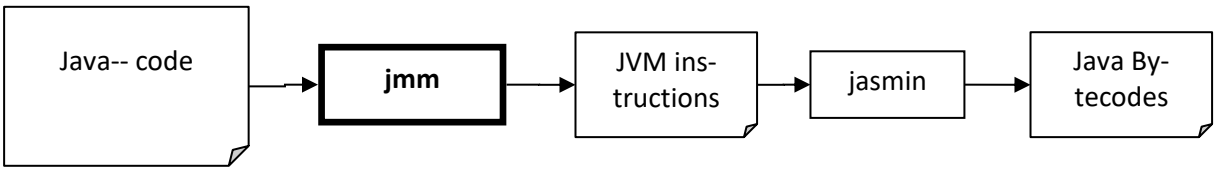

Figure 1. Compilation Flow.

The compiler shall generate a class with the name <*input_file*>.j. The .j classes will then be translated into Java *bytecode* classes (*classfiles)* using the tool *jasmin* [2].

The compiler shall include the following stages/phases: lexical analysis, syntactic analysis, semantic analysis, and code generation. The optimization stage (invoked by the "-o" option) is optional.

[2] <num> is an integer between 0 and 255.

Error Handling

An important part of any compiler is the indication of associated errors in each analysis phase. The error messages must be readable and useful. For example, in the syntactic analysis, the compiler should give more error information than the default ones reported by JavaCC. We recommend reading the document about handling and recovering from errors in JavaCC [3].

In this project, the error recovery shall only consider errors in the *while* expression. The parser shall be able to continue parsing the code if a syntactic error is found in a *while* expression. One possibility is to force the analysis to ignore the terminal symbols (tokens) until "{" is found.

Recovering from errors allows the compiler to present more meaningful errors from a single compilation. Note that in the presence of errors in the input program, the compiler should not abort execution immediately after the first error. Instead, it shall report a list of errors so that the developer proceeds with their correction (e.g., the compiler could report the first 10 errors found before aborting the execution).

Semantic Analysis

In order to reduce the workload of the semantic analysis, the compiler shall only verify the semantic rules in expressions, including the arguments in the invocation of methods that belong to the class being analyzed (i.e., calls to imported methods are assumed to be semantically correct).

The detected semantic errors shall be reported, and the compiler shall abort execution.

The semantic analysis shall have as input JSON files generated from the AST[3] obtained from the JavaCC grammar, representing the source code and the symbol table. We will provide a set of Java interfaces that shall be implemented during the project, and that will allow the generation of the JSON output.

As output, the semantic analysis shall generate the OLLIR described in [7].

Code Generation

Code generation shall take as input the OLLIR described in [7] and generate *jasmin* code.

[3] *Abstract Syntax Tree*.

3. The Java-- Language

The Java-- language is a subset of the Java programming language. Thus, invalid programs in Java must be also invalid in Java--, and Java-- valid programs are valid in Java and must be functionally equivalent.

Figure 2 presents the grammar of Java—in EBNF (*Extended Backus–Naur Form*)[4]. The elements of the grammar ***Identifier*** and ***IntegerLiteral*** shall obey to the lexical rules of the Java programming language. The comments in Java—shall obey to the Java rules regarding comments.

In this version of the compiler, the only accepted calls to methods shall be in statements with direct assignment (e.g., a = f();, a=m1.g();)or as simple call statements, i.e., without assignment (e.g., f2(); , m1.g();).

```
Program          = ImportDeclaration, ClassDeclaration, EOF

ImportDeclaration = {"import", Identifier, { ".", Identifier }, ";"};

ClassDeclaration = "class", Identifier, [ "extends", Identifier ], "{", { VarDeclaration }, { MethodDeclaration } "}";

VarDeclaration   = Type, Identifier, ";";

MethodDeclaration = "public", Type, Identifier, "(", [ Type, Identifier, { ",", Type, Identifier }, ], ")", "{", { VarDeclaration }, { Statement }, "return", Expression, ";", "}"
                  | "public", "static", "void", "main", "(", "String", "[", "]", Identifier, ")", "{", { VarDeclaration }, { Statement }, "}"
                  ;

Type             = "int", "[", "]"
                 | "boolean"
                 | "int"
                 | Identifier
                 ;
```

[4] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

```
Statement    = "{", { Statement }, "}"
             | "if", "(", Expression, ")", Statement, "else", Statement
             | "while", "(", Expression, ")", Statement
             | Expression, ";"
             | Identifier, "=", Expression, ";"
             | Identifier, "[", Expression, "]", "=", Expression, ";"
             ;

Expression   = Expression, ( "&&" | "<" | "+" | "-" | "*" | "/") , Expression
             | Expression, "[", Expression, "]"
             | Expression, ".", "length"
             | Expression, ".", Identifier, "(", [ Expression { ",", Expres-
               sion } ], ")"
             | IntegerLiteral
             | "true"
             | "false"
             | Identifier
             | "this"
             | "new", "int", "[", Expression, "]"
             | "new", Identifier ,"(" ,")"
             | "!", Expression
             | "(", Expression, ")"
             ;
```

Figure 2. Grammar of Java-- in EBNF. Note: ‘,’ delimits tokens in sequence, ‘{…}’ means zero or more occurrences of ‘…’, ‘[…]’ means optional ‘…’.

A program in the **Java--** language contains one or more classes. Figure 3 depicts an example of a simple **Java--** program.

```
import io;

class Fac {
    public int ComputeFac(int num){
        int num_aux ;
        if (num < 1)
            num_aux = 1;
        else
            num_aux = num * (this.ComputeFac(num-1));
        return num_aux;
    }
    public static void main(String[] args){
        io.println(new Fac().ComputeFac(10)); //assuming the existence
                                              // of the classfile io.class
    }
}
```

Figure 3. First program in Java--.

4. Optimizations and Register Allocation in the jmm Compiler

The stages described in this section are necessary to be eligible for final grades between 18 and 20 (out of 20). We expect that by default (i.e., without the option "–o"), the compiler generates JVM [1] code with lower cost instructions in the cases of: *iload*, *istore*, *astore*, *aload*, loading constants to the stack, use of *iinc*, etc.

All the optimizations included in your compiler shall be identified in the **README.txt** file that shall be part of the files of the project to be submitted once completed.

Option –r=<n>

With the option "–r", and for $n \geq 1$, the compiler tries to assign the local variables used in each function of each **Java--** class to the first <*n*> local variables of the JVM (or to the maximum local variables of the JVM when *n* is greater than the maximum):

- It shall report the local variables used in each method of the **Java--** class that are assigned to each local variable of the JVM.
- If the value of *n* is not enough to have the required local variables of the JVM sufficient to store the variables of the Java-- method, the compiler shall abort execution and shall report an error and indicate the minimum number of JVM local variables required.
- This option needs the determination of the lifetime of variables using *dataflow analysis* (regarding lifetime analysis, please consult the slides of the course and/or one of the books in the bibliography). An efficient implementation of dataflow analysis uses, for *def* and *use* sets, objects of the *BitSet* Java class.
- The register allocation, sometimes also referred as "register assignment", (in this particular case, it corresponds to the assignment of the variables of a function to the first *n* local

variables of the JVM) can be performed with the use of the graph coloring algorithm described in the course. However, we accept that your compiler includes a different register allocation algorithm.

Option "–o":

With the option "-o" the compiler shall include the following two optimizations:

- Identify uses of the local function variables that can be substituted by constants. It can diminish the number of local variables of the JVM used as variables with statically known constant values can be promoted to constants. Note that in this optimization known as "constant propagation" your compiler does not need to include the evaluation of expressions with constants (an optimization known as "constant folding").
- Use templates for compiling "while" loops that eliminate the use of unnecessary "goto" instructions just after the conditional branch that controls if the loop shall execute another iteration or shall terminate (the compilers that have included this optimization by default do not need to make modifications and do not need that this optimization be controlled by option "-o")

Table 1 presents the percentages for each optimization.

Table 1. Summary of the optimizations and associated percentage of points (maximum of 2) in the global score of the compiler.

Optimization	% in terms of points assigned to the optimization stage of the compiler
-r	60 %
-o	20 %
others	20 %

Include an additional code optimization selected by your group. Surprise us!!!!

5. Suggested Stages for the Compiler

The suggested development stages for the **jmm** compiler are the following:

1. Develop a *parser* for **Java--** using JavaCC and taking as starting point the Java-- grammar furnished (note that the original grammar may originate conflicts when implemented

with *parsers* of LL(1)[5] type and in that case you need to modify the grammar in order to solve those conflicts);

2. Include error treatment and recovery mechanisms for while conditions;
3. Proceed with the specification of the file **jjt**[6] to generate, using JJTree, a new version of the *parser* including in this case the generation of the syntax tree (the generated tree should be an AST[7]), annotating the nodes and leafs of the tree with the information (including tokens) necessary to perform the subsequent compiler steps; **[*checkpoint*[8] 1]**
4. Implement the interfaces that will allow the generation of the JSON files representing the source code and the necessary symbol tables;
5. Implement the Semantic Analysis[9] and generate the LLIR code, OLLIR (see document [7]), from the AST;
6. Generate from the OLLIR the JVM code accepted by *jasmin* corresponding to the invocation of functions in **Java--**;
7. Generate from the OLLIR JVM code accepted by *jasmin* for arithmetic expressions; **[*checkpoint* 2]**
8. Generate from the OLLIR JVM code accepted by *jasmin* for conditional instructions (*if* and *if-else*);
9. Generate from the OLLIR JVM code accepted by *jasmin* for loops;
10. Generate from the OLLIR JVM code accepted by *jasmin* to deal with arrays.
11. Complete the compiler and test it using a set of Java-- classes; **[*checkpoint* 3]**
12. Proceed with the optimizations related to the code generation, related to the register allocation ("-r" option) and the optimizations related to the "-o" option.
    [this task is necessary for students intending to be eligible for final project grades greater or equal than 18 (out of 20)]

As soon as you have the generation of the AST concluded, it is a good idea to proceed to the tasks needed to generate the OLLIR code and the JVM instructions for simple Java-- examples. Then, you can consider the more generic case related to the invocation of functions.

[5] Although the value of the global *lookahead* should be 1, it is acceptable if you use local lookahead values greater than 1 and syntactic lookahead.
[6] Basically, you can copy the file **.jj* and make the modifications so that JJTree generates the code to generate the syntax trees.
[7] *Abstract Syntax Tree*.
[8] It corresponds to the presentation of the stages already developed.
[9] The compiler shall not verify if invocations to methods with code not included in the Java-- class under compilation are according to the prototypes of that methods.

Note that the compiler shall report the possible errors that may occurred in the syntactic and semantic analysis.

6. JVM Instructions and the generation of Java Bytecodes

As a first basis to learn about the JVM instructions and the Java bytecodes generated we include herein a set of examples. Figure 4 shows a simple Java class to print "Hello World".

```
class Hello {
   public static void main(String args[]) {
        System.out.println("Hello World!");
   }
}
```

Figure 4. Java "Hello" class (file "Hello.java").

After compiling the class above with the *javac* (*javac Hello.java*) we obtain the file *Hello.class* (file with the Java bytecodes for the given input class). To see the JVM instructions and the class attributes (see Figure 5) we can execute[10]:

```
javap –c Hello
```

```
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return
public static void main(java.lang.String[]);
  Code:
   0: getstatic     #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3: ldc           #3; //String Hello World!
   5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return
}
```

Figure 5. JVM instructions in the *bytecodes* obtained after compilation of the "Hello" class with *javac*.

We can program with JVM instructions a class equivalent to the previous class. In this case, we will obtain the *bytecodes* Java from the *assembly* with JVM instructions using *jasmin*. The class equivalent to the *Hello* class (in the file *Hello.java*) is the class described in the file *Hello.j*

[10] To obtain information about the number of local variables and the number of levels of the stack necessary for each method one can use: ***javap –verbose Hello***.

using an *assembly* language with JVM instructions and syntax supported by *jasmin*. Figure 6 presents the code of the *Hello* class in file *Hello.j*.

```
; class with syntax accepted by jasmin 2.3

.class public Hello
.super java/lang/Object

;
; standard initializer
.method public <init>()V
   aload_0

   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2
   ;.limit locals 2 ; this example does not need local variables

   getstatic java/lang/System.out Ljava/io/PrintStream;
   ldc "Hello World!"
   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
   return
.end method
```

Figure 6. Programming of the "Hello" class (file "Hello.j") using JVM instructions in a syntax accepted by *jasmin*.

We can now generate the Java bytecodes for this class using *jasmin*:

```
java –jar jasmin.jar Hello.j
```

(this way we obtain the *Hello.class* classfile which has the same functionality as the class generated previously from the Java code)

7. References

[1] The Java Virtual Machine Specification, http://java.sun.com/docs/books/jvms/
[2] Jasmin Home Page, http://jasmin.sourceforge.net/
[3] JavaCC , Error Handling, https://javacc.github.io/javacc/tutorials/error-handling.html
[4] JavaCC, https://javacc.org/
[5] V. Kodaganallur, "Incorporating language processing into Java applications: a JavaCC tutorial," in IEEE Software, vol. 21, no. 4, pp. 70-77, July-Aug. 2004, doi: 10.1109/MS.2004.16.
[6] Compilers Course, "Compiler Interfaces and Stages", v1.0, MIEIC, FEUP, February 2021

[7] Compilers Course, “OO-based Low-Level Intermediate Representation (OLLIR)”, v1.0, MIEIC, FEUP, February 2021.