Compilers (L.EIC026)

2º Semester – 2021/2022

Design and Implementation of the Java-- language Compiler: From Java Source to Bytecodes

Version 1.6

BSc. in Informatics and Computing Engineering
Department of Informatics Engineering
Faculty of Engineering of the University of Porto (FEUP)

March 22, 2022

Table of Contents

Objectives: This programming project aims at exposing the students to the various aspects of programming language design and implementation by building a working compiler for a simple, but realistic high-level programming language. In this process, the students are expected to apply the knowledge acquired during the lectures and understand the various underlying algorithms and implementation trade-offs. The envisioned compiler will be able to handle a subset of the popular Java programming language and generate valid JVM (*Java Virtual Machine*) instructions in the *jasmin* format, which are then translated into Java bytecodes by the *jasmin* assembler.

---

1 Introduction

The goal of this programming project assignment is to allow students to acquire programming skills in the development and implementation of language translation tools by applying concepts introduced in the Compilers' course unit. In this process, students are asked to develop a compiler for a subset of the Java programming language, referred to as "jmm", which stands for "Java-minus-minus".

We begin by description the syntax of the Java—language and present a sample first code example and then describe the compilation structure you are expected to follow in developing your Java—compiler.

2 The Java-- Language

The Java-- language is a subset of the Java programming language. Thus, invalid programs in Java must also be invalid in Java--, and Java-- valid programs are valid in Java and must be functionally equivalent. The proposed language, **Java--** is fully compatible with **Java**, and can be integrated into a Java application. Classes that have been compiled into bytecodes can be used in **Java--** code.

Figure 1 depicts the grammar of Java-- in EBNF (*Extended Backus–Naur Form*)[1]. The elements of the grammar ***Identifier*** and ***IntegerLiteral*** follow the lexical rules of the Java programming language. The comments in Java-- also follow the Java rules regarding comments.

In the compiler you are expected to develop, method invocation from imported classes can only be used in statements with direct assignment (*e.g.,* a = M.f();, a=m1.g();) or as simple call statements, *i.e.,* without assignment (*e.g.*, M.f2(); , m1.g();). Calls to methods declared inside the class can only appear in compounded operation (*e.g*, a = b * this.m(10,20), where "m" is declared inside the class)[2].

```
Program          = ImportDeclaration, ClassDeclaration, EOF

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

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

VarDeclaration   = Type, Identifier, ";";
```

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

[2] This restriction is due to information regarding types not being available for imported methods.

```
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
                  ;

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 { ",", Expression } ], ")"
                  | IntegerLiteral
                  | "true"
                  | "false"
                  | Identifier
                  | "this"
                  | "new", "int", "[", Expression, "]"
                  | "new", Identifier ,"(" ,")"
                  | "!", Expression
                  | "(", Expression, ")"
                  ;
```

Figure 1. EBNF Java-- Grammar. (',' delimits tokens in sequence, '{…}' means zero or more occurrences of '…', '[…]' means optional '…').

A program in the **Java--** language contains one or more classes. Figure 2 depicts an example of a simple **Java--** program. Note that the invocation to method "println" from the class "io" is done as a simple call statement (`io.println(…);`). The method argument is a compounded expression that includes a call expression, but it is allowed because the "`ComputeFac`" method pertains to the declared class. If that method was not from this class, then its invocation would have to be done outside and before the `io.println` call statement. This is an intended limitation that simplifies the semantic analysis that will be performed later on by your compiler.

```
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 2. A sample Java-- program.

3 The jmm Compiler and Compilation Flow

The *jmm* compiler is expected to translate programs written in **Java--**[3] (see Section 2), into java *bytecodes* [1]. Figure 3 depicts the compilation flow of the compiler. The compiler generates class files with JVM instructions accepted by *jasmin* [2], a tool that translates those classes in Java *bytecodes* (classfiles).

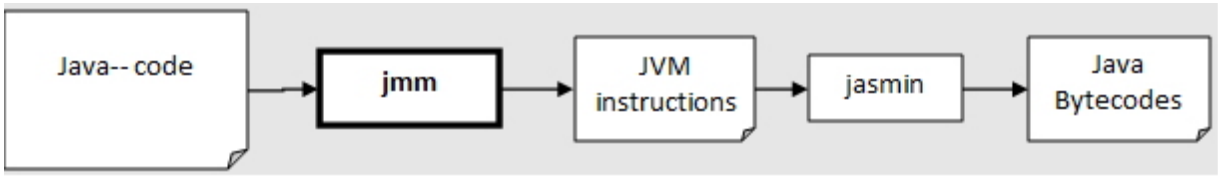

Figure 3. The jmm Compiler Compilation Flow.

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

The *jmm* compiler expects the following command-line syntax (flags between square brackets are optional) which is valid in both Windows and Unix execution environments:

```
comp2022-00 [-r=<num>] [-o] [-d] -i=<input_file.jmm>
```

where *<input_file.jmm>* is the **Java--** class file to be compiled.

This command a the predefined "`comp2022-00`" script which internally calls the binary files produced by the *gradle* build system.

Various aspects of the operation of the *jmm* compiler are controlled by a series of command-line options as described below.

The "–r" option directs the compiler to use only the first *<num>*[4] local variables of the JVM when assigning the local variables used in each **Java--** function to the local JVM variables. This is designed to allow Java—code to be compiled to target architecture with very limited number of physical registers. Without the "–r" option (equivalent to –r=-1), the compiler will use as many JVM local variables as local variables present in the original **Java--** function. When *<num>* is zero the compiler should try to use the minimum number of local variables of the JVM it can.

The "–o" option directs the compiler to perform code optimizations. Section X12 below contains more details about the "-r" and the "-o" options.

The "-d" option enables additional output related to the compilation process, to help programmers understand the compiler's operation. This includes display of the internally generated AST, the symbol table, applied optimizations - such as the number of required registers per method – and any other information deemed to be important to programmers. Without the "-d" option, the compiler should print a minimal amount of information, which should only include the indication of each compilation stage executed, warnings and errors.

bytecode classes (*classfiles)* using *jasmin* [2]. Note that the ".j" file should take the same name as the class declared inside the <input_file>.jmm, not the name as the input file. The JmmParser interface includes the parse() method with the following interface:

```
JmmParserResult parse(String jmmCode, Map<String, String> config);
```

The values in the Map object are drawn from the command line and you are responsible for parsing them into it.

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

4 Compiler Structure

Compilation Phases

Your Jmm compiler project should be structured as a sequence of phases, namely:

1) **Parsing**: the input file is parsed, and an AST is generated.
2) **Semantic analysis**: AST is analyzed, a symbol table is generated.
3) **High-level and low-level optimization**: high-level optimizations are applied to the AST, OLLIR is generated, and low-level optimizations are applied in OLLIR.
4) **Code generation**: Jasmin code is generated from the input OLLIR.

Each of these stages has a corresponding Java interface that needs to be implemented, in order to build the compilation flow. This means that you need to structure you code as sequence of calls tho these methods. This is important so tha later on we can use the same method names and use them to test your code.

These interfaces interconnect with each other in a pipeline fashion. This means that an "output" interface of a given stage is the input interface of the next as depicted in Figure 4.

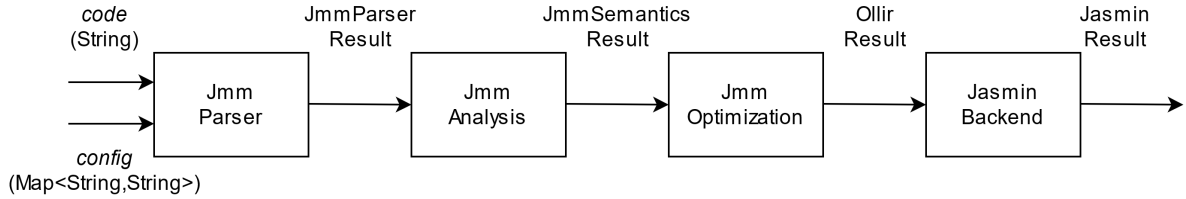

Figure 4. The jmm Compiler pipeline stage organization.

Different compiler stages need to be operational at different points in time throughout your project development as they will be graded for performance and functionality. In Section 5 we present a list of what is expected at each checkpoint.

At the start of the execution, the main Java program invokes the `parse` method as depicted in Figure 5 below defined in the *JmmParser* interface. This method receives a configuration object of the type *Map<String, String>* which needs to be filled with the options you process from the command line.

```java
public static void main(String[] args) {
    SpecsSystem.programStandardInit();

    SpecsLogs.info("Executing with args: " + Arrays.toString(args));

    // read the input code
    if (args.length != 1) {
        throw new RuntimeException("Expected a single argument, a path to an existing input file.");
    }
    File inputFile = new File(args[0]);
    if (!inputFile.isFile()) {
        throw new RuntimeException("Expected a path to an existing input file, got '" + args[0] + "'.");
    }
    String input = SpecsIo.read(inputFile);

    // Create config
    Map<String, String> config = new HashMap<>();
    config.put("inputFile", args[0]);
    config.put("optimize", "false");
    config.put("registerAllocation", "-1");
    config.put("debug", "false");

    // Instantiate JmmParser
    SimpleParser parser = new SimpleParser();

    // Parse stage
    JmmParserResult parserResult = parser.parse(input, config);

    // Check if there are parsing errors
    TestUtils.noErrors(parserResult.getReports());

    // ... add remaining stages
}
```

Figure 5. Example main() function that starts the compilation pipeline.

The configuration map, `config.properties`, is passed as an argument to the first stage of the compilation flow, the parsing stage, and then propagated to the remaining stages. The table below contains the mappings that are expected in the configuration map, as well as default values, expected type, and corresponding command line flag. You are free to develop your own compiler options (for instance to control passes of your own) by adding additional key-string pairs. Note that every value is a **string**.

Key Name	Default Value	Type	Flag
"optimize"	"false"	Boolean (as string)	-o
"registerAllocation"	"-1"	Integer (as string)	-r
"debug"	"false"	Boolean (as string)	-d
"inputFile"	""	String	-i

Your compiler should output the *Jasmin* code for the compiled class with the name *<class_name>*.j. The .j classes will then be translated into Java *bytecode* classes (*classfiles)* using *jasmin* [2]. Note that the ".j" file should take the same name as the class declared inside the <input_file>.jmm, not the name as the input file.

Lexical and Syntactic Analysis

These are the first stages of the compiler which will be based on the JavaCC21 tool [3]. You need to adapt and extend the grammar described in Section 2 (using the JavaCC21 format) as a ".jj" file. This grammar has many issues that need to be solved, including conflict of choice and operator priority as described next.

Choice Conflict: This conflict occurs when the parser has two (or more) possible choices regarding the grammar rule to chose to expand (in the LL parsing algorithmic approach). The behavior of JavaCC21 compiler is to take the first rule (in lexicographicaly order). This means that, when processing a grammar that contains two possible choices for the same (current) token, the JavaCC21 compiler will generate a parser without emitting an error or warning.

Consider the following grammar as an example:

```
Start ::= ( Assign | Add) +
Assign ::= id "=" Factor ";"
Add ::= Factor (+ Factor)? ";"
Factor ::= id | integer
```

Using this grammar, JavaCC21 will generate a parser that will produce the output below for the input: "`a = 10;`".

```
Start
  Assign
    a
    Factor
      10
```

However, when you try to parse "`a+2;`", the parser will return an error as shown below:

```
Encountered an error at (or somewhere around) input:1:2
Was expecting one of the following:
EQUALS
Found string "+" of type PLUS
```

This is due to a grammar conflict between the Assign and Add rules, as both rules can start with the token "`id`". Since the parser is only looking at the current token, it will always follow the first choice accepting that token.

To fix this issue you have two options. The first option requires you to restructure the grammar, for instance by factoring the common tokens in the rules, like the mathematical distributive law (e.g., a (b + c)). This might be a very cumbersome process and might lead to unreadable ASTs that are not easy to process and require a significant amount of post processing work.

The second option is to use "SCAN", a feature of JavaCC21 that allows to look N tokens ahead instead of just one, commonly known as *lookahead*. This is a cleaner approach that maintains the tree as it is already defined. The previous grammar is fixed by simply adding "SCAN 2" in the first choice of the Start rule:

```
Start ::= (SCAN 2 Assign | Add) +
```

This is directing the compiler: "Look at the two next tokens, if they are the first two that the Assign rule accepts (id "="), then continue this rule, otherwise move to the next option."

Therefore, if the parser input is "`a + 1;`" the SCAN will observe an `id` followed by a "+", which is not accepted by the *Assign* rule, and will continue to the *Add* rule.

Using SCAN (lookahead) is a cleaner approach, however it introduces overhead. In highly recursive grammars, SCAN might create a high overhead, especially when using high values of lookahead. As a practical rule, you should combine these two options while trying to keep your grammar definition as clean as possible, and the generated tree clean and organized. **SCAN**s in the project should not have a value higher than 2.

To promote a clean grammar development, you are strongly encouraged to not use lookahead values greater than 2. Failure to do so, will likely result in a lower project grade.

Operator Priority: The provided grammar has all the arithmetic operations in the Expression rule at the same level. You will have to reorganize these operations so that they have the same priority that is expected in Java [4], so that the generated AST already contains the arithmetic operations with a correct structure. It whould be obvious that operations with higher priority should be closer to the leaves in the AST, while lower priority should be closer to the "parent node" of the expression.

Error Handling

An important part of any compiler is the feedback of associated errors at each analysis phase. The error messages must be readable and useful. For example, during syntactic analysis, the compiler can provide more error information than the default ones reported by JavaCC21.

In this project, the error recovery will simply focus on a "global" analysis that just avoids the compiler immediately crashing after an error occurs. Instead, it is expected for the compiler to process and report that error. The provided project skeleton already includes a global try-catch for any type of exception that might occur. It is expected that you extend this "catch" statement with another one that specializes to the *ParseException* class, which represents a JavaCC21 error, and returns the error as a "Report" instance of type "Error". The report should

also determine if the exception is a lexical or a syntactic problem. Note that if a token is created with the type “INVALID”, then it means the compiler was not able to determine the type of the token.

AST Clean Up and Node Annotation

In this step you will proceed with the "clean up” of the AST by removing and changing unnecessary or redundant AST nodes. For this, you will rename the nodes that are generated using the “#<name>” format, or simply avoid the generation of the node for that rule by using “#void”. You must disable smart node creation in order to have full control of the generated AST. To do this, the top of your .jj file should have the option " `SMART_NODE_CREATION = false;`".

For instance, in the following example, the left side shows a “polluted” tree, while the right side shows the same, cleaner, tree for the statement: “`a = b + c;`”

```
Start
  Assignment
     LHS
       Id
     RHS
       Expression
          AdditiveExpression
             MultiplicativeExpression
                UnaryExpression
                  Factor
                     Id
            MultiplicativeExpression
                UnaryExpression
                  Factor
                     Id
```

```
Start
   Assignment
      Id
      BinaryExpression
         Id
         Id
```

You should also annotate the AST, which in the example above is for instance to provide the actual value of the token “id” and the operation type inside the respective nodes, represented in square brackets below:

```
Start
   Assignment
      Id [a]
      BinaryExpression [+]
         Id [b]
         Id [c]
```

Lexical and Syntatic Analysis Interfaces

We have defined three interfaces for the first compiler stage, namely *JmmParser JmmNode*, and *JmmParserResult*. To implement this stage your compiler needs to extend the *JmmParser* interface. The base projet already contains an example implementation, `SimpleParser`. This interface contains a single method that receives a *String* with the jmm code to parser and a *Map<String, String>* with the configuration, and returns is a *JmmParserResult* instance.

The *JmmParserResult* expects three input arguments: the root node of the AST, a list of reports, and the *Map<String, String>* with the configuration. The root node of the AST must be a *JmmNode* instance, an interface also provided in the base project.

The *JmmNode* interface represents a node of the AST and will be used for navigating the AST and accessing information about each node.

The list of reports collects all the logging, debugging warning, and error information that might occur during compilation. For instance, if there is an error during compilation, you should return a *JmmParserResult* instance with at least a report of type Error, and a *null JmmNode* if one could not be created.

To help you with the task of developing your compiler, in addition to these interfaces, we will be providing you with other interfaces and libraries. Among these, we can highlight the JmmVisitor interface and its several default implementations, which allow you to quickly add functionality ot the tree nodes without changing them directly. Two other important libraries are *SpecsCollections* and *SpecsIo* (see [6]). The former provides you with methods to manipulate and filter java collections, while the latter provides you with methods to interact with files and folders in an OS-agnostic way.

Semantic Analysis

This compiler stage is responsible for validating the contents of the AST. This includes verifying the type of operations, if invoked functions or methods exist, their expected arguments, etc.

For this project the compiler will only need to 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).

If the analysis detects semantic errors, they must be reported, and the compiler will abort execution after completing the semantic analysis.

The semantic analysis returns a *JmmSemanticsResult* instance that can be built based on the previous *JmmParserResult*, and additionally contains a *SymbolTable* instance (representing the symbol table) and possibly additional reports

5 Code Optimization and OLLIR

After AST validation by the semantic analysis phase, the compiler should perform an optimization phase. To this effect, you are expected to make use of the *JmmOptimization interface*.

The *JmmOptimization* interface defines three methods, each corresponding to a different optimization stage, respectively: AST-based optimization, conversion to *OO-based Low-Level Intermediate Representation* (OLLIR [5]), and OLLIR-based optimization. Some optimizations will be applied by modifications of the AST (e.g., constant propagation), and other optimizations and register allocation by modifications in the OLLIR code.

The output of this stage is an instance of the *OllirResult* class which can be built on the *JmmSemanticsResult*, adding a *String* with the generated OLLIR code and possibly additional reports.

6 Jasmin Code Generation

In this project, you are asked to output *jasmin* code by implementing the interface *JasminBackend*. You can accomplish this by using the optimized code in the OLLIR format translating it to the jasmin assembly format the Java Virtual Machine (JVM) which ultimately provides a more convenient way to generate Java bytecodes. Please see [2] further details on the *jasmin* format and assembler operation.

7 Optimizations and Register Allocation in the jmm Compiler

The optimizations described in this section are required for you to earn a final project grade between 18 and 20 (out of 20). It is expected that even without optimizations enabled (i.e., without the option "–o"), the compiler generates JVM [1] code with lower cost instructions (e.g., uses `iload_0` instead of `iload 0`). The expected instructions include: *iload*, *istore*, *astore*, *aload*, loading constants to the stack, use of *iinc*, etc.

You must identify all the optimizations included in your compiler in the **README.md** file that will 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 -1):

- It will report the local variables used in each method of the **Java--** class that are assigned to each local variable of the JVM.

- If the required value of *n* is not enough to store the variables of the Java-- method, the compiler will abort, 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). Register allocation, sometimes also referred as "register assignment", (in this 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 will include the following two optimizations:

- Constant Propagation: identify uses of the local variables that can be substituted by constants. This can reduce 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 your compiler does not need to include the evaluation of expressions with constants (an optimization known as "constant folding").
- Elimination of unnecessary GOTOs: use templates for compiling "while" loops that eliminate the use of unnecessary "*goto*" instructions just after the conditional branch that controls if the loop will execute another iteration or 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"). In other words, the optimization should limit the number of *gotos* as much as possible and still maintain the original code functionality.

8 Suggested Development Workplan and Checkpoint Schedule

The suggested development stages for the **jmm** compiler are as follows:

1. Develop a *parser* for **Java--** using JavaCC21 and taking as starting point the provided Java-- grammar (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 to solve those conflicts);
2. Include error handling at a global level to prevent the compiler from crashing;

[5] It is acceptable if you use local lookahead (SCAN) with a maximum of 2 and syntactic lookahead.

3. Proceed with the simplification of the generated syntax tree and the annotation of the nodes and leaves of the tree with the information necessary to perform the subsequent compiler steps;
4. Make sure the AST nodes generated by your grammar implement the interface *Jmm-Node*; **[*checkpoint* 1]**
5. Implement the Semantic Analysis[6] and generate OLLIR code (see document [5]), from the AST;
6. Generate, from the OLLIR code, the JVM code accepted by *jasmin* corresponding to the invocation of functions in **Java--**;
7. Generate, from the OLLIR code, the JVM code accepted by *jasmin* for arithmetic expressions;
8. Generate, from the OLLIR code, the JVM code accepted by *jasmin* for conditional instructions (*if* and *if-else*); **[*checkpoint 2*]**
9. Generate, from the OLLIR code, the JVM code accepted by *jasmin* for loops;
10. Generate, from the OLLIR code, the 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 code generation, register allocation ("-r" option), and the optimizations related to the "-o" option.

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.

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

9 JVM Instructions and the generation of Java Bytecodes

As an introduction to JVM instructions and the Java bytecodes, consider the following example. Figure 6 shows a simple Java class to print "Hello World".

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

Figure 6. Java "Hello" class (file "Hello.java").

[6] The compiler does not need to verify if invocations to imported methods (i.e., code not included in the Java-- class under compilation) are according to the prototypes of that methods.

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 disassemble the class file, obtain the JVM instructions and the class attributes for the code in Figure 7 you can use[7] the "`javap -c Hello`" command:

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

Programming Java bytecodes by hand can be cumbersome. For instance, constants must be added to a constant pool and referred by number. Instead, we will use *jasmin* to write the Java bytecodes in a more human-readable way and compile the *jasmin* code directly to a .class file. The class equivalent to the *Hello* class (in the file *Hello.java*) is the class described in the file *Hello.j* using an *assembly* language with JVM instructions and syntax supported by *jasmin*. Figure 8 presents the code of the *Hello* class in file *Hello.j*.

[7] 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***.

```
; 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 8. 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.

Repository File Organization and Turn-in Instructions

Your repository will start with a base project similar to the one you used during the JavaCC tutorial. In the base repository there is a folder *src-lib*, which contains code that you will need during the project, such as the interfaces for the stages. *Do not change any file in this folder*. We might update the files in this folder at any time, and if there are changes this could break your code.

On the date agreed for each delivery (i.e., checkpoint 2 and final delivery), we will make a copy of the contents of the repositories, which we will be used for evaluation.

Regression Testing

To test your code, you can use the class *TestUtils*, together with unit tests. The class contains a set of overloaded methods for each stage. For the first part of the work, you will use the method *parse()* which has two variants, one that receives a *String* with the jmm code, and another that receives the code *String* and a map with the configuration.

For the test class to find your implementations for the stages, it uses the file *config.properties* that is provided to you and located at root of your repository. It has four properties, one for each stage (i.e., ParserClass, AnalysisClass, OptimizationClass, BackendClass). The values associated with these properties are the names of the classes responsible for implementing the stage. Initially it only has one value, *pt.up.fe.comp. SimpleParser*, associated with the first stage. During the development of your compiler, you will update this file to set up the classes that implement each of the other compilation stages.

10 Project Evaluation and Group Grading

As a group project, the structure, and thus the effort of the elements of the group, is naturally split into stages that essentially correspond to steps in the flow of a traditional (or classical) compiler. It is natural and expected that during this project students face some of the problems that may be present when developing a traditional compiler. The stages of the project allow the students to manage the efforts and dedication to the project according to their desired level of performance.

Note that 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 lecturers and documented in the final report. While in a given group individual students' grades are typically identical, unbalanced workload and thus level of effort can result in distinct grades.

Table 1 presents how the 2 points (out of 20) of the project that are assigned to optimizations are distributed. “Others” are additional optimizations, selected by your group. The optimization phase is required if you would like your compiler project to get a score greater than 18.

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 %

11 References

[1] The Java Virtual Machine Specification, http://java.sun.com/docs/books/jvms/

[2] Jasmin Home Page, http://jasmin.sourceforge.net/

[3] JavaCC21, https://github.com/javacc21/javacc21

[4] Java Operators, https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

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

[6] The Specs Java library. https://github.com/specs-feup/specs-java-libs/tree/master/SpecsUtils