Design and Implementation of the JAVA-- language Compiler
Checkpoint 3

Compilers - L.EIC026 - 2024/2025

Dr. João Bispo, Dr. Tiago Carvalho, Alexandre Abreu,
José Ferreira, Nuno Cardoso, Pedro Gonçalo Correia, Tiago Santos

University of Porto/FEUP
Department of Informatics Engineering

Based on the document by Dr. João Bispo, Dr. Tiago Carvalho, Lázaro Costa, Pedro Pinto, Susana Lima and Alexandre Abreu

version 1.2, May 2025

Contents

1 Generate Jasmin code **2**
1.1 Interfaces . . . 2
1.2 Optimizations . . . 2
1.3 `.limit` directives . . . 2
1.4 Checklist . . . 3

2 JVM Instructions and the generation of Java Bytecodes **3**
2.1 When in doubt, reverse engineer it . . . 5

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 Java Virtual Machine (JVM) instructions in the *jasmin* format, which are then translated into Java *bytecodes* by the jasmin assembler.

In the previous checkpoint, you focused on the semantic analysis of the code and generation of OLLIR code and optimizations for both AST and register base intermediate representations. In this checkpoint, you will implement the final stage in the compiler, the backend, by generating Jasmin code (a Java bytecode assembler) and implementing a set of optimizations at this level.

1 Generate Jasmin code

The next step after the OO-based Low-Level Intermediate Representation (OLLIR) generation is the generation of the Jasmin code from OLLIR code. Similarly to what you did when converting from the AST to OLLIR code, now you will repeat the process, but starting from OLLIR's root node, the *ClassUnit* that you can access from an instance of *OllirResult*.

OLLIR does not provide visitors to its internal representation, but it has a tree structure where you can ask for the children of each element, which the class `JasminGenerator` uses to manually implement a visitor. You should expand on this class in order to generate the Jasmin code that is missing.

Check Jasmin's webpage [1] for further details on the Jasmin format and assembler operation. For more details on JVM instructions and the generation of Java bytecode, please refer to Section 2. When generating Jasmin code, you will need to convert the OLLIR types to Java bytecode type descriptors, you can check the Java documentation for details on the descriptor format [2].

1.1 Interfaces

You will output Jasmin code with the class *JasminBackendImpl*, which implements the interface *JasminBackend*. This interface contains a method *toJasmin* which converts an OLLIR *ClassUnit* to a String representing Jasmin bytecode. This method expects as input the result of the previous stage, a *OllirResult*, and returns a *JasminResult* instance, as follows:

JasminResult toJasmin(OllirResult ollirResult);

The output of this stage is an instance of the *JasminResult* class, which can be built using the previous *OllirResult*, a String with the generated Jasmin bytecode, and possibly additional reports. This result is used by the Jasmin tool to generate actual Java bytecode.

Note that now that you have finished the last stage of the compiler, you can actually execute the code, and the class *JasminResult* has methods for that (e.g., several overloads of `run()`).

1.2 Optimizations

The JVM has some instructions that are a specialization of other more generic instructions. For instance, you can use the instruction `ldc` to load any valid constant, but there are instructions for when the constant has a certain size (e.g., `sipush` when the constant fits in a short).

These specialized instructions usually have performance benefits (e.g., code size), and should always be used when possible, even when optimizations are disabled (i.e., the flag "–o"). The expected instructions that you should specialize include: iload, istore, astore, aload, loading constants to the stack, use of iinc and comparisons to zero (e.g., iflt).

1.3 `.limit` directives

For each JVM method, you have to add a preamble where you set the values for the directives `.limit stack` and `.limit locals`.

The value of `.limit locals` corresponds to the maximum number of registers required for that method. Take into account that:

- if you have an instance method, register 0 always contains `this`;
- an instruction such as `istore 5` implies at least 6 registers;
- the parameters of the method also count towards `.limit locals`.

The value of `.limit stack` corresponds to the maximum stack size that is needed for a given method. The stack contains all the arguments for invocations and other instructions. Keep in mind that, if you think back to the JAVA-- code, after executing any JAVA-- statement, the stack should

be empty. If the statement contains multiple subexpressions, during its execution, the stack will hold the necessary intermediate values.

The initial version of the project sets the values of the limits to 99, you will have to calculate the correct values for each method.

1.4 Checklist

By the end of **checkpoint 3**, it is expected that you can generate **Jasmin** for:

- Basic class structure (including constructor <init>)
- Class fields
- Method structure (
- Assignments
- Arithmetic operations (with correct precedence)
- Method invocation
- Conditional instructions (if and if-else statements)
- Loops (while statement)
- Instructions related to arrays
  - Declarations (use of "Array" type): parameters, fields, ...
  - Array accesses (`b = a[0]`)
  - Array assignments (`a[0] = b`)
  - Array reference (e.g. `foo(a)`, where a is an array)
- Calculate `.limit locals` and `.limit stack`
- low cost instructions
  - `iload_x`, `istore_x`, `astore_x`,`aload_x` (e.g., instead of `iload x`)
  - `iconst_0`, `bipush`, `sipush`, `ldc` (load constants to the stack with the appropriate instruction)
  - use of `iinc` (replace `i=i+1` with `i++`)
  - `iflt`, `ifne`, etc (compare against zero, instead of two values, e.g., `if_icmplt`)

2 JVM Instructions and the generation of Java Bytecodes

This sections is intended to give you an overview of what are bytecodes, how the Java compiler generates bytecodes, and how they relate to the Jasmin tool. As an introduction to JVM instructions and the Java bytecodes, consider the following example. Figure 1 shows a simple Java class to print "Hello World".

```
1 class Hello {
2   public static void main(String args[]) {
3     System.out.println("Hello World!");
4   }
5 }
```

Figure 1: Java "Hello" class (file "Hello.java").

After compiling the class above with `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 and obtain the JVM instructions, you can use the "javap -c Hello" command, which will output something similar to what you can see in Figure 2:

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

Programming directly Java bytecode 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.

Figure 3 presents the same code of the previous Hello class in Java bytecodes, but written in Jasmin.

```
; 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 3: Programming of the "Hello" class (file "Hello.j") using JVM instructions in a syntax accepted by jasmin.

Assuming the Jasmin code is in a file called Hello.j, we can now generate the Java bytecodes for this class with the command:

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. During the project, you will not have to call Jasmin manually. Instead, the class *JasminResult*, returned by the interface *JasminBackend*, contains a method *compile()*, which returns the compiled class file, and a method *run()*, which directly executes your Jasmin code. You have overloads for the method run that accept input arguments (as if called by command line) and even simulated user input, por testing interactive applications.

2.1 When in doubt, reverse engineer it

Although we only provide one example of Jasmin code, you can generate as many examples of Java bytecodes as necessary. JAVA-- code is (almost) fully compatible with Java, so you can compile a `.jmm` file as if it was a Java file and then decompile it. You just need to be careful with imports that have a single identifier, such as `import A;`, Java does not support imports from the default package. In these cases, remove such imports.

For instance, consider the file HelloWorld.jmm (in package `pt.up.fe.comp.jmm` of your *test* folder). First, we rename HelloWorld.jmm to HelloWorld.java and use *javac* to compile it:

javac HelloWorld.java

However, if we try to compile it, we will have an error like the one shown in Figure 4. This error occurs because HelloWorld.jmm uses an import that is in the default package (has no package), and that is not allowed in Java. Therefore, we have to comment/remove that import and compile it again.

```
.\HelloWorld.java:1: error: '.' expected
 import ioPlus;
 ^
 1 error
```

Figure 4: Error message after compiling the code in HelloWorld.java.

However, in Figure 5 we have another error. Now, the error occurs because we need to tell *javac* where *ioPlus* is. This is a library to assist your Jasmin code, and it is located in the folder *libs-jmm/compiled*. We have to provide that folder as part of the class path, with the flag *cp* as follows:

javac -cp ./libs-jmm/compiled HelloWorld.java

```
HelloWorld.java:4: error: cannot find symbol
 ioPlus.printHelloWorld();
 ^
 symbol: variable ioPlus
 location: class HelloWorld
 1 error
```

Figure 5: Error message after compiling the code in HelloWorld.java.

Note that if you want to add more than one folder to the class path, you need to separate the folders with a character that depends on the OS you are. If you are in Windows, you use ; (e.g. -cp lib1;lib2), in Linux you use : (e.g. -cp lib1:lib2).

Now we have compiled the Java code to Java bytecode successfully, and we should have a file HelloWorld.class in the folder. To decompile it and obtain the Java bytecodes, we can use the command *javap* as follows:

javap -c HelloWorld.class

This command should output the Java bytecodes for the class HelloWorld as shown in Figure 6.

This output, despite not being equal to the Jasmin bytecode format, gives you an idea of how the bytecodes should be generated. Note that for the actual code inside a method, the JVM bytecodes are very similar to the Jasmin bytecodes, with just a few differences. For instance, in the code, the instruction labeled as "1", instead of using invokespecial #1 (i.e. instead of accessing the "constant pool"), you will write the actual signature of the method (e.g., invokespecial java/lang/Object/<init>()V). More details regarding JVM bytecode instructions can be seen in Jasmin's webpage [3] and Oracle's Java Virtual Machine Specification [1]. You can also use Wikipedia's list of Java bytecode instructions as quick-reference material [4].

```
Compiled from "HelloWorld.java"
 class HelloWorld {
 HelloWorld();
 Code:
 0: aload_0
 1: invokespecial #1 // Method java/lang/Object."<init>":()V
 4: return
public static void main(java.lang.String[]);
 Code:
 0: invokestatic #7 // Method ioPlus.printHelloWorld:()V
 3: return
 }
```

Figure 6: JVM instructions in the bytecodes obtained after compilation of the "Hello" class with *javac*.

References

[1] Jasmin home page. http://jasmin.sourceforge.net/.

[2] Java bytecode type descriptors. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.2-200.

[3] The java virtual machine instruction set. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html.

[4] List of java bytecode instructions. https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions.