Design and Implementation of the JAVA-- language Compiler
Checkpoint 2

Compilers - L.EIC026 - 2023/2024

Dr. João Bispo, Dr. Tiago Carvalho, Lázaro Costa,
Pedro Pinto, Susana Lima and Alexandre Abreu
University of Porto/FEUP
Department of Informatics Engineering

version 1.1, March 2024

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 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 lexical and syntactic analysis stages and on generating the Symbol Table. For Checkpoint 2, you will be working on the remaining phases: Semantic Analysis, OO-based Low-Level Intermediate Representation (OLLIR) Generation, and Jasmin Generation. Since there are three phases, one way of splitting the work can be each student to take care of one phase.

1 Semantic Analysis

This compiler stage expects you to analyse and validate the Abstract-Syntax Tree (AST) in terms of semantics. This includes, for instance, verification of the type of operations in invoked methods, if the methods for a call exists, if the types of the arguments match, etc. In this phase you will be using the results of the previous checkpoint, more specifically the resulting AST from the parsing phase, and the generated symbol table.

Semantic analysis is used both to verify the validity of the input code (e.g. if a given variable that is used exists, if operands have the correct types) and to extract contextual information from the AST (e.g. calculate the return type of a given operation). You will be doing these two operations (more or less) at the same time, and to aid the analysis, you will use the symbol table.

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 analysed (i.e., calls to imported methods are assumed to be semantically correct). For a list of the main analyses you will need to perform please refer to the checklist at Section 1.2. This list already includes the main expected semantic checks for checkpoint 2.

If the analysis detects semantic errors, they must be reported, and the compiler will abort execution after completing the semantic analysis. Note that the sometimes it may make sense to stop the analysis immediately after certain errors, and other times you can just add the error to the "list of reports" and keep doing the semantic analysis. This will be left open for you to explore. We provide a base implementation (i.e., *JmmAnalysisImpl*) for the semantic analysis, that you must complete.

Tip: Use the analysis process to annotate the tree with extra information when you think it is relevant. For example, during OLLIR generation, you need to convert the AST to OLLIR code, which requires all operands and operations to be annotated with the resulting type. For the expression "1 + 2", since "1" and "2" are integers, and the result of the add operator in this case is also an integer, the equivalent OLLIR code will be somethig like "1.i32 +.i32 2.i32". Since for each expression node you will need to know its type, it can be beneficial to add this information to the AST while you are "analysing" those nodes (e.g. add a property "type"="int" in the node). Alternatively, you can write a (recursive) function that receives a node and calculates on-the-fly the type of the node.

1.1 Interfaces

This stage expands on the use of the *JmmAnalysis* interface, using the *JmmAnalysisImpl* class that you can find in your project, in package `pt.up.fe.comp2024.analysis`. You can always use the IntelliJ shortcut to find the class (i.e., `Ctrl+N` or double tap shift and write `JmmAnalysisImpl` to quickly find the class). Please consider the method in this class that has the following signature:

JmmSemanticsResult semanticAnalysis(JmmParserResult parserResult);

You can see that this stage now expects as input the result of the previous stage: the *JmmParserResult*. If you have not done it yet, please take a look at the class *JmmParserImpl*, in package `pt.up.fe.comp2024.parser`, and also refer to the previous document (project description and CP1). Regardless, the output of the previous stage, *JmmParserResult*, will contain the root *JmmNode* of the parsed code, and the input configuration.

In this stage you will develop the semantic analysis, and the output of that analysis will be an instance of the *JmmSemanticsResult* class. This instance is built based on the previous *JmmParserResult* (*JmmNode* root, reports and configuration), and additionally contains a *SymbolTable* instance (representing the symbol table). The `SymbolTable` should already be implemented from the previous checkpoint, so the idea now is to continue the development to the semantic analysis.

This analysis is composed by doing a set of passes throughout the AST (starting from the root node), where each pass tries to provide one or more semantic validations. Therefore, the idea of each pass is to visit the AST in a way that goes through the target type of nodes (and possibly other related nodes) and verifies if they are semantically correct.

To help in the development of this phase, the `JmmAnalysisImpl` class already provides logic to execute a list of analysis passes. Looking at this class you can see that it contains a field to store the analysis passes to perform in a list of `AnalysisPass` instances. An `AnalysisPass` is an interface that contains a single method:

List<Report> analyze(JmmNode root, SymbolTable table);

The method *analyze* receives the root node of the AST and the symbol table, and returns a list of Report instances. If any problem is detected, there should be at least a Report with ReportType ERROR in the list.

You can have many implementations of *AnalysisPass*, each for a single semantic analysis, or very few implementations where each implementation performs several analyses. If you add instances of your *AnalysisPass* implementations to the list *analysisPasses* inside *JmmAnalysisImpl*, they will be automatically called with the current code you have. Note that the current implementation will call all *AnalysisPass* instances before stopping the analysis, if you want to stop early, you will have to modify the code.

You can implement an *AnalysisPass* without using the Visitor pattern, but for certain analyses it can help. We provide the abstract class *AnalysisVisitor* which already implements *AnalysisPass*, and extends the *PreorderJmmVisitor*, which automatically traverses the AST, in preorder fashion [7]. Please check the class *UndeclaredVariable* for an example of how you can extend *AnalysisVisitor* to implement an *AnalysisPass*.

Tip: You can use these passes to add extra information to the AST, such as calculating and adding the type of each expression. This might help other passes and aid in the OLLIR generation phase and avoid traversing the AST multiple times to obtain the same information again and again. In the other hand, do not be afraid to do multiple visits over the AST, if that helps with the readability and organization of your code.

1.2 Checklist

The following are analyses that we will test and that must report an error if the rule is not respected.

1.2.1 Types and Declarations Verification

- Verify if identifiers used in the code have a corresponding declaration, either as a local variable, a method parameter, a field of the class or an imported class
- Operands of an operation must have types compatible with the operation (e.g. int + boolean is an error because + expects two integers.)
- Array cannot be used in arithmetic operations (e.g. array1 + array2 is an error)
- Array access is done over an array
- Array access index is an expression of type integer
- Type of the assignee must be compatible with the assigned (an_int = a_bool is an error)
- Expressions in conditions must return a boolean (`if(2+3)` is an error)
- "this" expression cannot be used in a static method
- "this" can be used as an "object" (e.g. `A a; a = this;` is correct if the declared class is A or the declared class extends A)
- A vararg type when used, must always be the type of the last parameter in a method declaration. Also, only one parameter can be vararg, but the method can have several parameters
- Variable declarations, field declarations and method returns cannot be vararg
- Array initializer (e.g., `[1, 2, 3]`) can be used in all places (i.e., expressions) that can accept an array of integers

1.2.2 Method Verification

- When calling methods of the class declared in the code, verify if the types of arguments of the call are compatible with the types in the method declaration
- If the calling method accepts varargs, it can accept both a variable number of arguments of the same type as an array, or directly an array
- In case the method does not exist, verify if the class extends an imported class and report an error if it does not.
  - If the class extends another class, assume the method exists in one of the super classes, and that is being correctly called
- When calling methods that belong to other classes other than the class declared in the code, verify if the classes are being imported.
  - As already explained in section 1.2 of the CP1 document, if a class is being imported, assume the types of the expression where it is used are correct. For instance, for the code `bool a; a = M.foo();`, if `M` is an imported class, then assume it has a method named foo without parameters that returns a boolean.

An important remark here is that, while you will only implement part of the OLLIR and Jasmin generation (e.g., you do not have to implement support for arrays or loops yet), semantic analysis is to be completely supported by the end of checkpoint 2, i.e. it will include the analysis of array types and verification of expressions in while statements. This way, in checkpoint 3 you will focus on completing the generation of OLLIR and Jasmin code, and in code optimization.

2 Generate OLLIR

After the semantic analysis, if no errors are reported, two more stages are executed: optimization and backend. During this checkpoint, you will generate OLLIR code in the optimization stage (code optimization will only be done in Checkpoint 3). In the backend stage, you will generate Jasmin code, which is a Java assembler format. The semantic analysis provides you with a symbol table, and potentially an AST with additional annotated information. Along with the AST, all that information will be used to convert the high-level AST you have used until now into a low-level intermediate representation named OLLIR (Object-Oriented Low-Level Intermediate Representation).

To understand what is OLLIR, we have made available several documents [6, 5]. We will be providing additional information regarding OLLIR, that will be collected in the FAQ document. The converted OLLIR tree will be used for the optimization phase during the final checkpoint, and to generate Java bytecode in the Jasmin format.

We provide a base implementation (i.e., *JmmOptimizationImpl*) and a generator (i.e., *OllirGeneratorVisitor*) for the OLLIR generation, that you must complete.

2.1 Interfaces

For this stage, you will be working with the class *JmmOptimizationImpl*, an implementation of the *JmmOptimization* interface which defines three methods. However, for the Checkpoint 2 you are only required to work on the *toOllir* method, which converts the AST to the OLLIR format. The method signature below illustrates this interface:

OllirResult toOllir(JmmSemanticsResult semanticsResult);

The output of this stage is an instance of the *OllirResult* class which is built using the previous *JmmSemanticsResult*, a string that will contain the generated OLLIR code, and possibly additional reports.

2.2 Online Translator

To help on the development of the JAVA-- to OLLIR translation, we provide a website [2] where you can put valid JAVA-- code, and it will return equivalent OLLIR code. Some caveats:

- The resulting OLLIR code is just an example of a possible translation. You do not have to (and probably will not) generate exactly the same code;
- The website does not support JAVA-- code with varargs. It will be part of your work figuring out how to handle varargs generation in the final assignment;
- There might be bugs, use Slack to report anything strange.

2.3 Checklist

By the end of **checkpoint 2**, it is expected that you can generate **OLLIR** for:

- Basic class structure (including constructor <init>)
- Class fields
- Method structure
- Assignments
- Arithmetic operations (with correct precedence)
- Method invocation

This means that **the following topics are not required for checkpoint 2 and are only to be supported in the final checkpoint:**

- Support to arrays (array types, varargs, array access, array initializer...)
- If statements
- While statements

3 Generate Jasmin code

The next step after the OLLIR generation is the generation of the Jasmin code from OLLIR code. Check Jasmin's webpage [1] for further details on the Jasmin format and assembler operation. You can immediately start working on the OLLIR to Jasmin generation by creating an instance of *OllirResult* which receives a string with OLLIR code. This constructor will parse the OLLIR code and return a *ClassUnit*, which represents the root node from where you can start navigating the OLLIR Intermediate Representation (IR).

You have several examples of OLLIR code in the package `pt.up.fe.comp.cp2.jasmin` of your *test* folder. For more details on JVM instructions and the generation of Java bytecode, please refer to Section 4.

3.1 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.

3.2 Checklist

By the end of **checkpoint 2**, it is expected that you can generate **Jasmin** for:

- Basic class structure (including constructor <init>)
- Class fields
- Method structure (in Jasmin, you can ignore stack and local limits for now, use 'limit stack 99' and 'limit locals 99')
- Assignments
- Arithmetic operations (with correct precedence)
- Method invocation

This means that **the following topics are not required for checkpoint 2 and are only to be supported in the final checkpoint:**

- Support to arrays
- If statements
- While statements
- Calculate limit stack/locals

4 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:

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.

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

```
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.

```
; 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.

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.

4.1 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.

```
1 public class Hello extends java.lang.Object{
2 public Hello();
3   Code:
4     0: aload_0
5     1: invokespecial #1; //Method java/lang/Object."<init>":()V
6     4: return
7 public static void main(java.lang.String[]);
8   Code:
9     0: getstatic     #2; //Field java/lang/System.out:Ljava/io/PrintStream;
10    3: ldc     #3; //String Hello World!
11    5: invokevirtual #4;//Method java/io/PrintStream.println:(Ljava/lang/String;)V
12    8: return
13 }
```

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

```
1 HelloWorld.java:4: error: cannot find symbol
2  ioPlus.printHelloWorld();
3  ^
4  symbol: variable ioPlus
5  location: class HelloWorld
6  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.

```
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*.

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].

References

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

[2] Java– to ollir. https://specs.fe.up.pt/comp2024-ollir.

[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.

[5] Ollir tool, v0.4, l.eic, feup, march 2024. https://docs.google.com/document/d/1cCC-YlpZ4cSlLsfVH78A1dPF5208R9S_GpYzYbYH_5I/edit?usp=sharing.

[6] Oo-based low-level intermediate representation (ollir), v0.7, l.eic, feup, march 2024. https://docs.google.com/document/d/1c0WkHU8K6J0DDRywR6atdwfAavKIgwYxhT0GXY-e0II/edit?usp=sharing.

[7] Preorder tree traversal. https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR.