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

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, April 2025

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, syntactic and semantic analysis stages. For Checkpoint 2, you will be working on the next phase: intermediate code generation and optimization, in this case, using the OO-based Low-Level Intermediate Representation (OLLIR).

1 OLLIR Generation

After the semantic analysis, if no errors are reported, the next stage to be executed is optimization. During this checkpoint, you will generate OLLIR code and apply optimizations both in the Abstract-Syntax Tree (AST), before the intermediate code generation, and directly over OLLIR. The semantic analysis has provided you with a symbol table, and potentially an AST with additional annotated information (e.g., types). 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 [3, 2]. We will be providing additional information regarding OLLIR, that will be collected in the Compilers support document. The converted OLLIR tree will be used during the last phase of the project 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.

1.1 Interfaces

For this stage, you will be working with the class *JmmOptimizationImpl*, an implementation of the *JmmOptimization* interface which defines three methods. For the OLLIR generation, 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.

1.2 Online Compiler

To help on the development of the Java-- to OLLIR translation, the Java– Online Compiler [1] now also shows the equivalent OLLIR code for some input Java-- 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;
- There will be bugs, use Slack to report anything out of the ordinary.

1.3 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
- ☐ Conditional instructions (if and if-else statements)
- ☐ Loops (while statement)

☐ Instructions related to arrays and varargs

- Declarations (use of the "Array" type): parameters, fields, ...
- Array accesses (b = a[0])
- Array assignments (a[0] = b)
- Array references (e.g. foo(a), where a is an array)

Note that "varargs" are just syntax-sugar for arrays. So, in the final OLLIR code, all past references to varargs, including method declarations and method calls, should have been translated to arrays. The only exception is the "varargs" keyword that appears in the signature of methods in OLLIR.

You can deal with this by changing the OLLIR generation to take it into consideration, or by changing the AST and symbol table, if needed.

You can see that the Jmm code in Figure 1 can be translated accordingly, as in Figure 2, paying attention to the translation of the method "foo".

```
class MyClass {
    int foo(int...a) {
        return 0;
    }

    int bar() {
        return this.foo(1, 2);
    }
}
```

Figure 1: Implementation of a class "MyClass" with a varargs method "foo" using a syntax accepted by Jmm.

```
MyClass {
    .construct MyClass().V {
    invokespecial(this, "<init>").V;
  }

    .method public varargs foo(a.array.i32).i32 {
        ret.i32 0.i32;
    }

    .method public bar().i32 {
        tmp0.array.i32 :=.array.i32 new(array, 2.i32).array.i32;
        tmp0[0.i32].i32 :=.i32 1.i32;
        tmp0[1.i32].i32 :=.i32 2.i32;
        tmp1.i32 :=.i32 invokevirtual(this, "foo", tmp0.array.i32).i32;
        return tmp1.i32;
    }
}
```

Figure 2: Translation of the class "MyClass" using a syntax accepted by OLLIR.

2 Optimizations in the jmm Compiler

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.

2.1 Register allocation

The option "–r=<n>" controls the register allocation, in the following way:

- $\boldsymbol{n \geq 1}$: The compiler will try to use at most <n> local variables when generating Jasmin instructions. Report the mapping between the local variables of each method and the corresponding local variable of the JVM. If the value of n is not enough to store every variable of the method, the compiler will abort, report an error and indicate the minimum number of JVM local variables required.
- $\boldsymbol{n = 0}$: The compiler will try to use as few local variables as it can. Report the mapping between the local variables of each method and the corresponding local variable of the JVM.
- $\boldsymbol{n = -1}$: The compiler will use as many variables as originally present in the OLLIR representation. This is the default value.

To implement this option, you need to use several algorithms, including calculating the lifetime of variables using data-flow analysis, and applying a graph coloring algorithm for the register allocation proper. We provide a support document [4] to help you implement this optimization.

2.2 Other optimizations

With the option "-o" the compiler will include the following two optimizations, which we suggest implementing as transformations in the AST:

- **Constant Propagation**: identify uses of the local variables that can be replaced 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.

  For example, consider the following code snippet, where all variables are integers:

  ```
  a = 10;
  b = a + 5;
  c = b + a;
  ```

  With constant propagation, the compiler would identify that the variable has a constant value of 10. The compiler would then replace all occurrences of a with 10, resulting in the following optimized code:

  ```
  a = 10;
  b = 10 + 5;
  c = b + 10;
  ```

- **Constant Folding**: the compiler analyzes the code and identifies expressions that involve constant values, such as numeric literals or string literals. The compiler then replaces the expressions with their resulting constant value, and the optimized code is generated.

  For example, consider the following code snippet:

  ```
  a = 10 + 5;
  ```

  With constant folding, the compiler would replace the expression 10 + 5 with its resulting constant value 15, resulting in the following optimized code:

  ```
  a = 15;
  ```

Notice that applying both optimizations in sequence might enable furthers optimizations. We recommend implementing each transformation as a separate visitor, with a boolean field that indicates if after the visit any modification were applied. This way, it is possible to execute the optimizations in a loop until it reaches a fixed point.

2.3 Interfaces

For this stage, you are required to implement the remaining methods of the *JmmOptimization* interface which are the optimization methods. The first works at the AST level and the second at the OLLIR level.

The method signature below illustrates the first optimize method:

JmmSemanticsResult optimize(JmmSemanticsResult semanticsResult);

The method signature below illustrates the second optimize method:

OllirResult optimize(OllirResult ollirResult);

2.4 Checklist

The following is a checklist of the optimizations you should implement:

- ☐ Register allocation
- ☐ Constant propagation and constant folding

References

[1] Java– online compiler. http://10.227.118.27:3000/.

[2] Ollir tool, v0.5, l.eic, feup, march 2025. https://docs.google.com/document/d/1c2b1TgODJ4WCbL91RNnt6mtTIDC3OKANrT57iSKOdrE/edit?usp=sharing.

[3] Oo-based low-level intermediate representation (ollir), v0.8, l.eic, feup, march 2025. https://docs.google.com/document/d/17X9M-gzO7g4zDeLax6qNoUOpVXIapE96Zrrok3aL498/edit?usp=sharing.

[4] Register allocation for the compilers project. https://docs.google.com/document/d/14_l17ffME6HbCc1F3-NH-Df8czx0aTg8Myfkt8WMDxE/edit?usp=sharing.