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

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 1.0, April 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 semantic analysis of the code and generation of OLLIR and Jasmin Code for specific structures and operations. In this checkpoint, you will generate the remaining OLLIR and Jasmin code and implement a set of optimizations.

1 Generate OLLIR

Continue the work done on Checkpoint 2 to generate Ollir code for the remaining structures and program logic. Remember to use the OLLIR documentation that we have made available [1, 2]. The converted OLLIR tree will be used for the optimization phase during the final checkpoint, and to generate Java bytecode in the Jasmin format.

1.1 Interfaces

Same as in Checkpoint 2.

1.2 Checklist

For the final checkpoint, you will generate OLLIR for the remaining parts of the project:

- 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 and jasmin.

You can deal with this by changing the OLLIR generation to take it into consideration (or the direct translation to jasmin for groups of two elements), 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".

```
1 class MyClass {
2     int foo(int...a) {
3         return 0;
4     }
5
6     int bar() {
7         return this.foo(1, 2);
8     }
9 }
```

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

2 Optimizations 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). If no optimization is developed, the maximum possible grade of the project is 18.

```
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);
        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.

Even without optimizations enabled (i.e., without the option "–o"), the compiler should generate JVM 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 and comparisons to zero (e.g., iflt).

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 Optimizations

2.1.1 Option –r=<n>

The option "–r" 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 calculate the lifetime of variables using data-flow 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 to as "register assignment", 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.

2.1.2 Option "-o"

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:

```
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.

2.2 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);

2.3 Checklist

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

- register allocation
- constant propagation and constant folding

3 Generate Jasmin code

Continue the work done on checkpoint 2 to generate Jasmin code for the remaining structures and program logic. Remember that you have several examples of OLLIR code in the package `pt.up.fe.comp.ollir` of your *test* folder.

Additionally, you should calculate the correct values for `.limit stack` and `.limit locals`, and always use lower cost instructions.

The value of `.limit locals` corresponds to the maximum number of registers used in the Jasmin code. 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 after executing any JAVA-- statement, the stack should be empty. If the statement contains multiple subexpressions, during its execution, the stack will hold the intermediate values needed.

3.1 Interfaces

Same as in Checkpoint 2.

3.2 Checklist

For the final checkpoint, you will generate Jasmin code for the remaining parts of the project:

- 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`)

References

[1] Oo-based low-level intermediate representation (ollir) tool - compiler backend tool, v0.4, l.eic, feup, april 2024. https://docs.google.com/document/d/1cCC-YlpZ4cSlLsfVH78A1dPF5208R9S_GpYzYbYH_5I/edit?usp=sharing.

[2] Oo-based low-level intermediate representation (ollir) tool - intermediate representation for the compiler backend, v0.4, l.eic, feup, april 2024. https://docs.google.com/document/d/1cOWkHU8K6JODDRywR6atdwfAavKIgwYxhTOGXY-eOII/edit?usp=sharing.