JVM Bytecode generation

Note: This document refers to slides and videos made by Nicolas Laurent who was the teacher of the LINFO2132 course from 2020 to 2021.

Introduction

After finishing the semantic analysis, it is time for your compiler to generate code that can be executed on a computer and that is (hopefully) equivalent to the source code. In the compiler pipeline that we saw in lecture 2, we are now in the last phase:

(As you can see, we are skipping the optional "Optimization" phase. We will come back to it later in the course)

There are different ways to generate code in a compiler:

1. The compiler can directly generate machine code for a specific CPU. This is not very practical because it means you have to write a new compiler for every CPU model. This type of compilation is called Ahead-Of-Time (AOT) compilation.
2. The compiler can generate an *Intermediate Representation* (IR) that is then translated by different backends to different CPU-specific binary codes. This is the current state of the art used in compilers such as GCC and the LLVM compiler framework. It's flexible and powerful because many interesting optimizations can be done on the CPU-independent intermediate representation before the machine code is generated. This is AOT compilation, too.
3. The compiler can generate code (called *bytecode*) for a virtual machine. The virtual machine can then interpret the bytecode (i.e., execute it instruction by instruction) or translate the bytecode during the program execution to machine instructions for the CPU the virtual machine is running on. The latter is called Just-In-Time (JIT) compilation.

For your project, we will use way number 3. Your job is to generate bytecode for the Java Virtual Machine (JVM). Doing so has several advantages. First, JVM implementations exist for many operating systems, which means the generated bytecode can be run on many different platforms and CPUs. Second, code generation for the JVM is relatively easy (easier than for an Intel CPU). And finally, there are JVM implementations with very powerful JIT compilation for the x64 and ARM architectures. For these reasons, the compilers of many popular programming languages, such as Kotlin, Clojure, and Scala, generate JVM bytecode.

For an introduction to the concept of bytecode and the JVM, read the slides "10a – JVM Bytecode.pdf" and watch the video https://www.youtube.com/watch?v=cijr3ARKtxo&list=PLOech0kWpH8-njQpmSNGSiQBPUvl8v3IM&index=22

Bytecode and the JVM

Although many languages can be compiled to JVM bytecode, the JVM is a virtual machine specifically designed for the Java language. The JVM reads the class files produced by the Java compiler and

executes them (starting with the `main()` method of the main class). This means that if we want to generate bytecode for our own programming language, we must use class files.

Class files contain a lot of information, not only the code for the methods, but also type information, the names of the class fields, code for the constructors, etc. For efficiency reasons, class files store all this information in a binary format. If you want to display the content of a class file in a human readable way, you can use the javap command line tool, or simply use a website like https://javap.yawk.at/ or https://godbolt.org/

Let's take the following simple method

```
public int square2(int num) {
    int s = num * num;
    return s*2;
}
```

and look at the information stored in the class file for it:

```
descriptor: (I)I
flags: (0x0001) ACC_PUBLIC
Code:
  stack=2, locals=3, args_size=2
     0: iload_1
     1: iload_1
     2: imul
     3: istore_2
     4: iload_2
     5: iconst_2
     6: imul
     7: ireturn
```

The descriptor gives the type of the method. The notation `(I)I` means that the method has one int-parameter and returns an int. You can find a complete description of this notation at the end of the slides 10c.

The flag `ACC_PUBLIC` indicates that the method is public.

The method has two parameters (`args_size=2`): the parameter num that you can see in the Java code (argument 1) and the invisible `this` parameter (always argument 0) that all non-static methods have and which is a reference to the object the method is executed on. The method has three local variables (`locals=3`), namely the local variable `s` and the two parameters `this` and `num` which also count as local variables. Therefore, we have as local variables:

Local variable 0:	`this`
Local variable 1:	`num`
Local variable 2:	`s`

In addition, the method uses a stack space of two "slots" (`stack=2`) for its calculations. In the JVM, a slot corresponds to one 32-bit value or one reference to an object. Larger values (such as `double` values) need two slots.

Lines 0 to 7 give the bytecode instructions of the method. Method execution always starts with a new empty stack. In line 0, we push the integer value of the first argument (`iload_1`) onto the stack. In line 1, we do it again, which means that the stack now contains twice the value of the variable `num` and looks like this:

Top of the stack: `num`
Second element on the stack: `num`

The `imul` instruction in line 2 removes the first two elements of the stack and pushes their product (i.e., `num*num`) onto the stack:

Top of the stack: `num*num`

In line 3, this value is removed from the stack and stored in local variable 2, i.e., the local variable `s`. The stack is now empty.

The value of `s` (local variable 2) is pushed again onto the stack in line 4, together with the integer constant 2 in line 5. Again, the instruction in line 6 removes the two values from the top of the stack and pushes the product onto the stack. Finally, the `ireturn` instruction in line 7 takes the value from the top of the stack and returns it.

Comparisons and loops are implemented with compare and jump instructions. For the Java method

```
public int test(int num) {
    if(num>3)
        return 1;
    else
        return 2;
}
```

the Java compiler generates the following code:

```
 0: iload_1
 1: iconst_3
 2: if_icmple 7          // jump to instruction 7 if num<=3
 5: iconst_1
 6: ireturn              // return 1
 7: iconst_2
 8: ireturn              // return 2
```

The instruction in line 2 works like this: It takes two integer values from the top of the stack and compares them. If the first value is less than or equal to the second value, the JVM jumps to line 7.

Constants in bytecode

In the two examples shown above, the bytecode used the `iconst_`*X* instructions to push the integer value *X* onto the stack. This only works for small integer values that can be directly stored in the bytecode instruction. There are also variants of the `iconst` instruction (`bipush` and `sipush`) that can push larger integer constants (up to 32767, i.e., a signed 16-bit value). However, for values larger than 16 bits the byte code needs to use the so-called *constant pool*. The constant pool is a table in the class file that contains *all* constant values used by the class. Not only integer values, but also:

- Long, float, double values
- Strings (the String object and its contents are two separate entries in the pool)
- The names of the classes and methods in the class file and from imported classes
- Type description strings for the fields and methods, like the `(I)I` from our first example above.

You can recognize bytecode instructions that refer to a value in the constant pool by the # character. For example, the instruction

```
ldc #7
```

pushes the constant defined in constant pool entry #7 onto the stack. Try it yourself! Replace the integer values in the above examples by larger values and see what happens in the bytecode and the constant pool. Note that you have to click a small button next to the words `Constant pool` on https://javap.yawk.at/ to see the pool entries.

An overview on all interesting instructions is given in the slides "10b – JVM Bytecode instructions.pdf" and the video https://www.youtube.com/watch?v=2-iCwCrvbQk&list=PLOech0kWpH8-njQpmSNGSiQBPUvl8v3IM&index=23

If you need more information about the different instructions, check the links on slide 8. It's also very useful to play with https://javap.yawk.at/ to see what the Java compiler would generate for an equivalent Java program.

Generating class files and bytecode

As already said, class files store the bytecode of the methods and additional information in a binary format. It would be *very* cumbersome to make your compiler directly generate class files in that format. Fortunately, there are some libraries that make this much easier. One such library is ASM from https://asm.ow2.io/

The ASM library provides classes and methods that help you to generate bytecode from your own compiler. In addition to taking care of the special binary format of class files, it can for example automatically calculate the number of local variables and stack space needed for your bytecode, manage the constant pool, and compute the targets (called "Label" in ASM) of the jump instructions used in if-statements and loops.

Generating a class file with ASM starts with a `ClassWriter` object. From this object, you can add new fields and methods, and then bytecode to the methods (with the `MethodVisitor` class). When you are finished, you can get the content of the class as a byte array (with the method `toByteArray` of the class writer) and store it in a file. For an introduction to ASM, see the slides "10c - Generating JVM Bytecode with ASM.pdf" and the video

https://www.youtube.com/watch?v=HqkH3nft2n0&list=PLOech0kWpH8-njQpmSNGSiQBPUvl8v3IM&index=24

On slide 8, there is a link to the Sigh demo language of Nicolas Laurent. There, you can find many examples of how to use the ASM library to generate code for different kinds of statements. https://github.com/norswap/sigh/blob/master/src/norswap/sigh/bytecode/BytecodeCompiler.java

Writing the code generator in your compiler

The language used in the project has some differences to Java. It has global variables which don't exist in Java. On the other hand, the language doesn't have classes and objects. But since the world of the JVM consists of classes and methods, we have to use them, too.

We propose the following for your compiler:

- Generate one main class for the program. The global variables will be the members of the class and the procedures will be the methods.
- Represent integer values by `int` and real values by `float`.

- Generate a class for each record type.
- Like the semantic analysis, write the code generator as a traversal of the AST.
- Feel free to call existing Java methods for the built-in I/O procedures (readInt, readReal,…) of the language.
- The JVM has a garbage collector to deallocate unused objects. This means the `delete` statement of our language used in the project is not really doing anything. However, you could implement some additional logic, for example check whether the programmer has tried to delete the same object twice etc.
- During development, use the `javap` command line tool to inspect the content of the class files generated by your compiler.