JVM Bytecode Backend for SimpliC

Lukas Tröjer
Lund University, Sweden
lukas.trojer@gmail.com

Suleyman Zahi
Lund University, Sweden
su4117za-s@student.lu.se

Abstract

In this paper we introduce a Java Virtual Machine (JVM) backend for SimpliC, a small language similar to C. A program written in SimpliC can thus be compiled to Java bytecode and run on the JVM. This allows for the use of many features that are often desirable, like platform independence and runtime optimizations. We then evaluate the performance of our generated bytecode using various benchmarks. Our results show that, compared to the bytecode generated by javac, our generated bytecode has virtually identical steady-state performance and only slightly worse start-up performance. Furthermore, the performance only becomes better with newer versions of Java. Our generated bytecode has not been able to be on par with optimized x86 assembly compiled from C, but can easily beat unoptimized x86 assembly on longer-running tasks.

1 Introduction

As part of an undergraduate compilers course, we implemented a compiler for SimpliC. SimpliC is a programming language that is similar to C, but implements only basic constructs. It supports integer variables, basic control flow and I/O like reading user input and printing. The SimpliC compiler generates x86 assembly code, which is assembled to machine code and linked to an executable. Having the compiler generate machine code is a quite common approach used by many languages, such as C, C++ and Go. These languages generate highly optimized machine code, whereas our compiler performs no optimizations. SimpliC also lacks memory management and other common features found in other languages. Since the compiler only generates x86 code, it will not run directly on other computer architectures, such as RISC, without using emulation or virtualization methods.

An alternative to generating machine code is to instead let the compiler emit bytecode that will run on an emulator or virtual machine. The Java virtual machine, JVM, is an example of such a platform. The JVM handles memory management and optimises the code at runtime using a JIT (Just-In-Time) compiler. Furthermore, it is also platform independent, so a JVM program can run without modifications on any platform that supports the Java Runtime Environment (JRE), which includes all popular hardware platforms. A language that runs on the JVM can adopt all the previously mentioned features without having to implement them. Because of this, there are several popular non-Java languages that run on the JVM and utilize the features it provides [10]. The main goal of this course paper is to implement a JVM backend for SimpliC, enabling it to generate bytecode which can then be run on the JVM. A secondary goal is to take advantage of JVM's memory management facilities and extend SimpliC with integer arrays, which have native support in JVM.

Course paper, EDAN70, Lund University, Sweden
January 15, 2024.

2 Background

2.1 SimpliC and JastAdd

A brief introduction to SimpliC was given in the previous section. The scanner for the compiler was implemented using *JFlex*, where one defines the regular expressions for each of the tokens in a specification. The parser was generated using *NeoBeaver*[8]. In the semantic actions of the parsing grammar, the abstract syntax tree, AST, is built. The AST is created with an abstract grammar file in the JastAdd meta-compiler [4]. The abstract grammar is a simpler version of the parsing grammar, stripping tokens with no semantic value and removing nonterminals that were introduced to resolve ambiguities in the parsing grammar [7]. JastAdd is a Java framework for implementing compilers and related tools. An abstract grammar in JastAdd is defined by a list of productions, where a production denotes an AST node. We can further define child nodes of the AST node, as well specify a superclass for it. When the grammar specification is compiled, JastAdd will generate concrete Java classes for all nodes, as well as methods for accessing child nodes and intrinsic attributes, for example the numerical value of a node representing an integer literal [7]. This and some other key techniques in JastAdd allows us to implement the backend of the compiler, whether that is generating machine code or JVM bytecode.

2.2 JVM

The Java Virtual Machine, JVM, is a virtual machine that executes a specialized instruction set, called Java bytecodes. JVM takes input in the form of a binary format called `class` file format. The class file contains the Java bytecode. A bytecode is not too dissimilar to a regular x86 machine code instruction. It has a one-byte opcode which denotes what operation it performs, optionally followed by operands. The amount of operands depends on the instruction [11].

JVM is a multithreaded platform, meaning it supports multiple threads of exection, commonly shortened to threads. A JVM thread has its own stack, where frames are stored. Every time a method is invoked, a new frame is put on the stack. Once the method invocation is done, the frame is simply removed. A frame contains an array of local variables and an operand stack. This architecture makes JVM stack-based. All bytecodes interact with the operand stack, whether it's for adding new data or grabbing operands from the stack. To illustrate this, consider the *iadd* instruction. The *i* prefix denotes that this instruction performs integer addition, so other datatypes such as *double* have their own addition instructions. The *iadd* instruction takes 2 integers as operands. These can be pushed to the operand stack, from previous instructions, as constants or loaded from the local variable array. The instruction pops the operand stack two times to get the values, performs the addition and pushes the result back to the stack. Now, subsequent instructions can use the value added to the stack [11].

In addition to having a operand stack and a local variable area, a frame also stores a reference to a class-specific constant pool. This constant pool is allocated from the heap. The heap is available to all JVM threads, it is here that object instances and arrays are stored. JVM uses a garbage collector to manage memory in the heap, though the constant pool is not garbage collected [11].

A programming language that correctly compiles to the `class` file format can be executed on the JVM. Such a language can be considered a JVM language. The most significant JVM language is Java. This is because the JVM was designed as part of the Java platform to execute programs written in Java. As such, it has support for special constructs defined in Java such as classes, arrays, objects and exceptions. Other JVM languages are Kotlin and Scala, to name a few.

3 Extending SimpliC with arrays

In order to be able to generate and run more interesting programs, we decided to extend SimpliC with integer arrays. Having the idea that we will generate Java bytecode in mind, this is the only area where SimpliC departs from working like C. Functionally it's closer to Java in two ways. First, our SimpliC arrays can be reassigned to have a different length than it did originally. This is unlike C where the length of an array is part of its type. Second, the arrays contain information about their lengths. This makes it much more convenient to pass arrays around to functions, since functions operating on arrays don't have to include an extra parameter containing the size.

The array syntax is however different from both C and Java, and is shown in listing 1. The function concat12345 concatenates an array with the five numbers [1, 2, 3, 4, 5] and returns the result. The number sign (#) is used for the length operator, a syntactic choice that is also found in a few other languages like Lua and J.

```
int[] concat12345(int[] arr) {
    // int[n] creates a zeroed array of size n
    int[] res = int[#arr + 5]; // # means size
    // array literals use [], not {}
    int[] nums = [1, 2, 3, 4, 5];
    int i = 0;
    while (i < #arr) {
        res[i] = arr[i];
        i = i + 1;
    }
    while (i < #res) {
        res[i] = nums[i - #arr];
        i = i + 1;
    }
    return res;
}
```

Listing 1. Example of arrays in SimpliC

3.1 Implementation

Adding support for arrays in the scanner, parser, name analyser, and type checker was not that difficult. We were able to reuse much of the structure we already have. For example, for type checking we have a JastAdd attribute type() which computes the actual type of some expression and another attribute expectedType() which computes the type it *should* have. When these attributes conflict, there's a type error. Adding support for arrays here was simply a matter of enumerating the different cases for the new constructs and specifying which types it has or should have.

The biggest changes that had to be done to the existing code involved function declarations and parameters. Previously they didn't need to contain any type information, but now we distinguish between integers and arrays of integers for return and parameter types. This means that in several places in our code we treat the two cases separately. If we were to develop this further it would be natural to refactor the code to handle type information more independently that it currently is. This would make it easier to add more types to the language in the future, but at this point it's not really necessary.

4 JVM Backend

The Java class file format is well documented, and it is possible to generate class files directly in binary form by following the class file specification. However, this can be a time consuming and error prone task, so we went a different route. There exists bytecode transformation libraries that

abstracts away the low-level details of the class file format into a higher-level API. The API can express common operations in the form of methods. For example, the bytecode to initialise a variable with a value, which involves multiple instructions can be compounded into a method.

4.1 Bytecode library

There are several bytecode libraries to choose from. From our comparison, each one has a particular niche which it is specialized in. Our particular use case involves generating bytecode and not so much analysing existing bytecode. For this reason we have opted to use Apache Commons Bytecode Engineering Library (BCEL) [2]. The library was previously called JavaClass[3], before being renamed to BCEL. We felt that its API was intuitive and more geared towards bytecode generation. Examples of other bytecode transformation libraries include ASM [13], Javassist [1] and Byte Buddy [14].

4.2 Implementation

As was discussed in section 2.1, the parser builds the AST for a given source file. The nodes in the AST represent language constructs, thus we can have each node define what bytecode it will generate. This is convenient to do in JastAdd, because of its support for static aspect-oriented programming. Instead of extending each class with the code we want to add, we can implement it as a separate module, or *aspect*, with inter-type declarations. An example of this is illustrated in listing 2. When JastAdd compiles the abstract grammar specification and the declared aspects, the aspect code will be added to the respective Java classes generated from the grammar [7].

```
aspect Bytecode {
    public void Add.bytecodeGen() {
        // ...
    }
    public void Mul.bytecodeGen() {
        // ...
    }
}
```

Listing 2. Adding an inter-type declared method bytecodeGen() to AST nodes Add and Mul in a separate aspect.

The root AST node in SimpliC is `Program`, which represents a source file. As the `class` file name implies, such a file is created for each top-level class defined in a Java file. Thus the `Program` node is represented as a Java class in our implementation. In BCEL, this is done by creating a `ClassGen` object. As mentioned in section 2.1, each class has an associated constant pool. Likewise, the `ClassGen` class has an instance attribute of `ConstantPoolGen`, which models the JVM constant pool. This attribute is significant since it will be passed as an argument for each declared SimpliC function that we generate bytecode for. As the compiler traverses the AST of a program, it will attach methods, fields and other code to the `ClassGen` object. When complete, we can simply call `getJavaClass()` to generate the final Java class [2].

A SimpliC program consists of function declarations and a `main` function which is the entry point for execution. To create functions, BCEL provides the `MethodGen` class. The classes we have discussed so far create the structure of a `class` file, so they are analogous to scaffolding. To add the JVM instructions, bytecodes, we utilize a class named `InstructionList`. A function normally contains a list of statements and those in turn can contain expressions. When we iterate through those statements, we can pass an `InstructionList` to them. They will in turn add bytecodes to the list, building up the function. The list instance will be unique for each function. An example of bytecode generation is given in listing 3. In the example, the getter methods are child nodes representing the operands. When the bytecodes for the operands have been generated, we simply generate the final `iadd` bytecode. In this case the bytecode was just an enum, but this depends on the instruction. Some bytecodes like `istore`, which create a local integer variable, requires more code since we have to interact with the array of local variables. BCEL provides a convenient API to achieve that. When the traversal of the function's child nodes is done, a `MethodGen` is finalized as a `Method`. The `Method` then is just added to the `ClassGen` [2].

```
public void Add.bytecodeGen(ConstantPoolGen cp,
    InstructionList il) {
    getLeft().bytecodeGen(cp, il);
    getRight().bytecodeGen(cp, il);
    il.append(InstructionConst.IADD);
}
```

Listing 3. Bytecode generation for an AST node performing addition.

We will now look at how control flow is implemented. Structurally it is similar to the implementation for x86 assembly. A comparison bytecode, `if_icmp`, is used to check the boolean condition. Each comparison operator has a specific bytecode. Like x86 assembly, a negated condition is used to skip the body if the condition is not fulfilled. To jump to a specific point, we can set a target directly on the `if_icmp` or `goto` bytecode. A target in this case is a byte offset within the bytecode array. Since our implementation uses dynamic dispatch to generate bytecode for AST nodes, the exact instructions aren't known at compile-time. As a workaround we insert no-operation, `NOP`, instructions at the points we need to jump to. Functionally, the `NOP` instructions are then equivalent to jump labels in x86 code. When the bytecode

list is finalized, we can iterate through all branch instructions and update their targets. New target will be the next bytecode appearing after the `NOP` they targetted previously. Thus, the `NOP` bytecodes can be safely deleted.

4.3 Bytecode arrays

Lastly, we will describe how bytecode generation for arrays is done. By implementing arrays, more AST nodes were added. These nodes represent constructs like arrays literals, declaration, assignments and element access. To create a new array instance, we use the `newarray` bytecode. It takes the size as an operand. The JVM will then allocate space for the array from the heap. To access an element from an array or assign a value to an element, we can utilize `iaload` and `iastore` respectively.

Listing 4 shows how array literals are created. The comment on the first line showcases what an array literal is, an array and all the expressions its elements contain. First, the size of the array is pushed to the operand stack, which `newarray` will pop to instantiate the array reference. Then, we loop through all elements and store them in the the array. `iastore` receives as operands the array reference, the index and the expression. We can use `dup` to duplicate the array reference to the stack, instead of loading in the variable repeatedly.

```
// [1,2,3,4,5] is an ArrayLiteral
public void ArrayLiteral.bytecodeGen(
ConstantPoolGen cp,
InstructionList il) {
    il.append(new PUSH(cp, getNumElem()));
    il.append(new NEWARRAY(Type.INT));
    for(int i = 0; i < getNumElem(); i++) {
        il.append(new DUP());
        il.append(new PUSH(cp, i));
        getElem(i).bytecodeGen(cp, il);
        il.append(new IASTORE());
    }
}
```

Listing 4. Bytecode generation for creating a new array literal in JVM.

5 Evaluation

5.1 Introduction

After verifying that our implementation works correctly using rigorous automated test cases, we evaluate our compiler by running benchmarks for different scenarios. One important thing to evaluate is the performance difference between SimpliC programs compiled with our compiler and equivalent Java programs compiled with javac. Since most optimizations happen during runtime we expect that the SimpliC programs won't differ that much in performance compared to the Java programs. We also evaluate the performance of our generated bytecode compared to x86 machine code generated in two different ways. On one hand we have the existing SimpliC to x86 assembly compiler that we implemented in the compilers course. The code that is generated by this compiler is entirely unoptimized. On the other hand we have gcc, which we use to compile equivalent C code using optimization level O2. For shorter-running programs, we expect that our generated bytecode will be significantly slower as the JVM takes some time to start. For longer-running tasks, this might not be the case however. Lastly, we compare running the same SimpliC programs using different versions of Java, in order to see if there has been any significant improvements over time.

The first time a program is run on the JVM it might not be very fast. However, if you let it run several times more on the same JVM instance it will likely be significantly faster. The reason for this is twofold. First, the JVM has a significant start-up time. Second, the bytecode will be better optimized the more times it is run. It is therefore important to distinguish between the performance at start-up, and the performance at the steady-state when no more optimizations can be performed. We will produce benchmarks measuring both.

As individual benchmarks can be subject to substantial random variations, it's also important that we run each benchmark several times. There are however many ways to consolidate the individual benchmarks into one or a few values that we use to draw our conclusions. We could pick the best time, the worst time, the mean, or the median. We could pick the interval between the best and worst values, or we could produce a confidence interval at some level of confidence, to just name a few options. It is of utmost importance that the benchmarking strategy is statistically rigorous. Otherwise, we face the risk that our conclusions are misleading, or simply false at worst. These issues are discussed by Georges, Buytaert, and Eeckhout [5] in their paper *Statistically Rigorous Java Performance Evaluation*. They suggest a methodology using confidence intervals that we follow closely.

5.2 Method

For the start-up benchmarks we use the Bash `time` command to run the same program from start to finish $n = 20$ times. From this sample we calculate the sample mean and a 95% confidence interval for the true mean.

For the steady-state benchmarks, the process is more complicated, and works as follows. First the JVM is started. Then we iteratively run the program, measuring the time it took using Java's `System.nanoTime()` method. The program is

run again and again until the last 100 iterations has a coefficient of variation (the ratio of standard deviation over mean) at or lower than 0.02. This is the threshold at which we determine that a steady-state has been reached, so we then return the mean of the last 100 iterations. This whole process is repeated $n = 20$ times, restarting the JVM between each time. Like for the start-up benchmarks, we calculate the sample mean and a 95% confidence interval for the true mean using this sample.

5.3 Result and Discussion

In figure 1 we compare the start-up performance of our generated bytecode from SimpliC to bytecode generated from Java using javac. We use two different kinds of tasks. The first is to construct an array in reverse order and sort it using bubble sort, while the second is to simulate Conway's Game of Life on a small board for a number of generations. For the start-up performance we see that our bytecode is slightly slower than that of javac. We think this is because the bytecode that java produces is a bit more optimimal than ours. While most of the optimizations happen during runtime by the JIT compiler, javac does optimize the code to some degree. For the steady-state performance, we see not significant difference at all. This is unsurprising as the JIT compilation will likely have produced very similar or even identical code when the steady state has been reached.

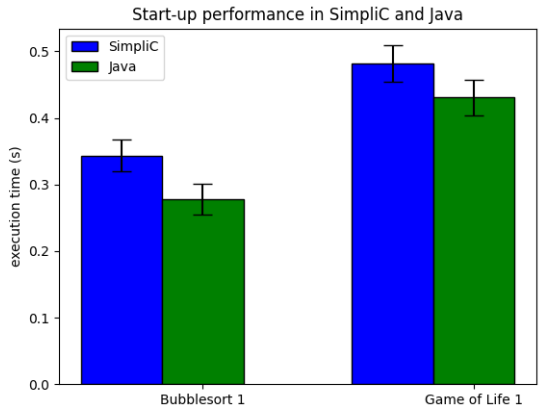

Figure 1. Start-up performance in SimpliC and Java

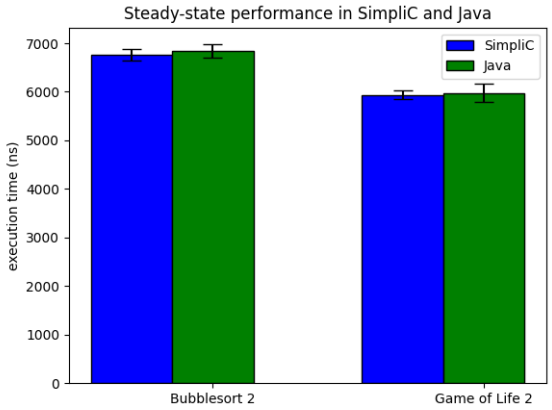

Figure 2. Steady-state performance in SimpliC and Java

In figure 3 we compare the start-up performance of our generated bytecode to that of x86 machine code, generated in the two ways described above. When counting the number of primes up to 1000, our generated bytecode is slower by several orders of magnitude. When the limit is increased to 50000 however, the situation looks very different. Now the x86 code generated from SimpliC is by far the slowest. This is not unexpected as it hasn't been optimized at all. The optimized x86 code generated using gcc is the fastest. However, the difference in performance between the bytecode program and the C to x86 program is not significantly different between when the limit is 1000 and when it is 50000. That is, the "extra work" that is needed when the limit is 50000 takes roughly the same amount of time. This shows that, at least in this case, the JIT compilation is working as it should, optimizing the code to be pretty fast.

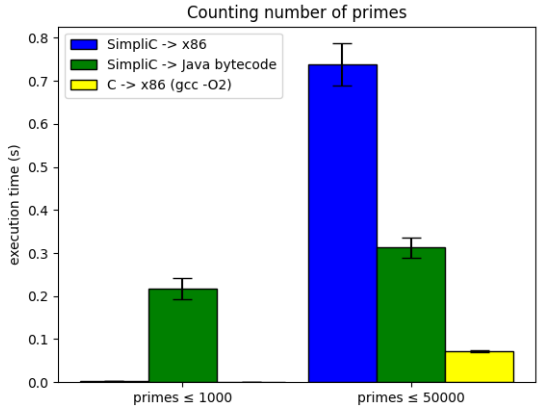

Figure 3. Counting number of primes

We also performed benchmarks running the same SimpliC programs using three different versions of Java, namely Java

11, Java 17, and Java 21. The task is a similar bubble sort task as before. For the steady-state (figure 5) we saw a significant performance increase between Java 17 and Java 21. For the start-up performance (figure 4) we also saw a significant performance increase, this time between Java 11 and Java 17. We may see further improvements in future versions of Java, and SimpliC programs will then be able to run faster "for free". This is yet another benefit of having our compiler target an existing virtual machine like the JVM.

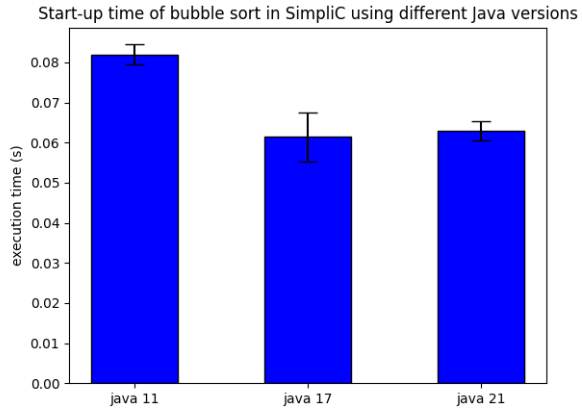

Figure 4. Start-up time of bubble sort in SimpliC using different Java versions

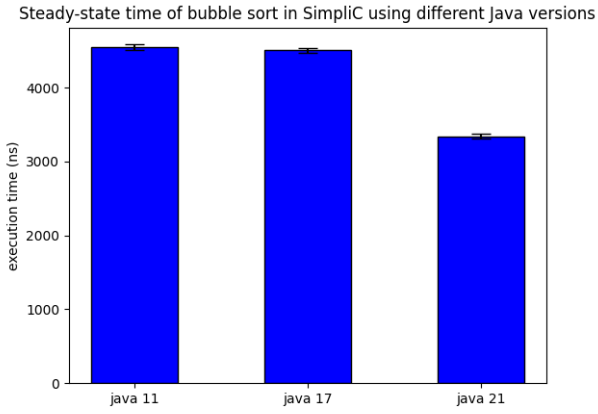

Figure 5. Steady-state time of bubble sort in SimpliC using different Java versions

6 Related work

Attribute grammars, AGs, are a formalism that allow us to assign attributes to nonterminal symbols in a grammar. An attribute can either be synthesized or inherited, whereby they are defined by attributes of child symbols or parent symbols [9]. Reference attributed grammars, RAGs, build upon AGs and let AST nodes reference attributes of other nodes in the tree [6]. Our implementation of the SimpliC compiler extensively uses JastAdd, which itself supports RAGs.

ExtendJ is a Java compiler that is designed to further extend Java with new constructs. It was implemented with JastAdd [12]. Like our compiler, it also generates Java bytecode. The difference is that ExtendJ's implementation is made from scratch by following the JVM specification, whereas we use a bytecode library.

7 Conclusion

The aim of this project was to implement a JVM backend for SimpliC. The motivation was to make the compiler faster and more efficient but also extend the language with more constructs, namely arrays. To create the JVM backend, we had to generate Java bytecode. BCEL was the tool used to achieve this goal. It was used in tandem with JastAdd, which let us write the bytecode generation for all AST nodes, previously defined in JastAdd as part of an abstract grammar. Our evaluation consisted of comparing the performance of the JVM backend to the x86 one and the C language. We also looked at the performance for different versions of the JVM. The results we obtained show that the steady-state performance of SimpliC was on par with that of Java, whilst the start-up performance was slightly worse. When performing a task with few iterations, both x86 SimpliC and C performed better. But when we increased the number of iterations, JVM SimpliC performs better than the x86, though still lags behind C. Lastly, we saw that newer JVM versions performed better on the same benchmark.

What we can conclude is that we can gain significant performance improvements by adopting a JVM backend, compared to unoptimized assembly code. We get the better performance at no cost, since JVM itself performs the optimizations. We also benefit from future releases and improvements of the JVM platform, if we opt to use them. One downside to using JVM that the results showcased is that we incur a penalty to the performance because of the start-up overhead of the virtual machine. In some scenarios, this may not be ideal. Still, for language implementers, targeting a virtual machine like the JVM can be a viable option to boost their languages performance and increase the feature set.

References

[1] Shigeru Chiba. 2000. Load-time structural reflection in Java. In *European Conference on Object-Oriented Programming*. Springer, 313–336.
[2] Markus Dahm. 1999. Byte code engineering. In *JIT'99: Java-Informations-Tage 1999*. Springer, 267–277.
[3] Markus Dahm. 2001/2002. The Byte Code Engineering Library. https://bcel.sourceforge.net/. (2001/2002). Accessed: 2023-11-21.
[4] Ekman, Torbjörn and Hedin, Görel. 2007. The JastAdd system — modular extensible compiler construction. 69, 1-3 (2007), 14–26. https:

//doi.org/{10.1016/j.scico.2007.02.003}

[5] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of OOPSLA 2007*. ACM, Montreal, Canada, 57–76. http://doi.acm.org/10.1145/1297027.1297033

[6] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[7] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0 Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

[8] Jesper Öqvist. [n. d.]. NeoBeaver. ([n. d.]). https://bitbucket.org/joqvist/neobeaver/src/master/ [Online; accessed 10-Decemeber-2023].

[9] Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511

[10] Wing Hang Li, David R. White, and Jeremy Singer. 2013. JVM-Hosted Languages: They Talk the Talk, but Do They Walk the Walk?. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/2500828.2500838

[11] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2023. *The Java Virtual Machine Specification*. https://docs.oracle.com/javase/specs/jvms/se21/html/index.html

[12] Jesper Öqvist. 2018. ExtendJ: extensible Java compiler. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. 234–235.

[13] OW2. [n. d.]. ASM. ([n. d.]). https://asm.ow2.io/index.html [Online; accessed 30-November-2023].

[14] Rafael Winterhalter. [n. d.]. ByteBuddy. ([n. d.]). https://bytebuddy.net/# [Online; accessed 30-November-2023].