Extending a Small Language with a Java Bytecode Back End

Project in Computer Science – EDAN70
January 25, 2016

Philip Mårtensson
D10, Lund University, Sweden
ada10pma@student.lu.se

Elliot Jalgard
D10, Lund University, Sweden
ada10eja@student.lu.se

Abstract

A simple language called *SimpliC* with only the core constructs from the C language and an x86 back end had been implemented in a previous course. This project set out to extend this language with new language constructs such as structs, global variables and more data types. Another new addition to the language was a back end for generating Java bytecode runnable in the Java Virtual Machine. The result was compared in terms of execution speed with another back end implementation using *LLVM* for a similar SimpliC language and corresponding extensions developed by two other groups.

1. Introduction

This project was done in the context of a follow up course for a compiler course, where the task was to work on a project related to compilers. The purpose of this specific project was to extend a small previously written language called *SimpliC* with new language constructs and a new back end to support code generation for the *Java Virtual Machine* (JVM). The base language, SimpliC, already supported many of the necessary language constructs from other languages. Among these were functions, if- and while statements, local variables, boolean comparisons and arithmetic operators. Moreover, this language had to use integers for everything. The extended language constructed in this project added support for structs, variables with global scope as well as the primitive data types booleans and floats.

The extension of language constructs was done in order to create a more interesting language. Seeing as structs are basically public classes, it would be a very interesting feature to add to this simple language. However, the main interest in this project was not the extra language constructs, but rather the implementation of a Java bytecode back end. Many compilers today already compile into bytecode in order to run on the JVM, for example *Scala* (Odersky et al. 2004), *Jython* (Pedroni and Rappin 2002) and *JRuby* (Nutter et al. 2011). The SimpliC language is by no means fairly comparable to those languages since it is much smaller. But it is a good project for the purpose of understanding compiler construction with a simple Java bytecode back end. Being able to run a custom language in the JVM has many advantages because of the huge install base of the JVM in various devices.

The result of the project was a complete back end for the extended SimpliC language. All of the mentioned extensions were done and some evaluation programs were constructed and the result of those were compared to other projects implementing the same language but with a another back end for LLVM instead of Java bytecode. The evaluation results showed that our implementation is not as fast as fast as the code generated by LLVM, but still acceptable considering its short development time and scope.

2. Background

The construction of a compiler can be abstracted to a front end and a back end. The front end is responsible for the *lexical-*, *syntactic-* and *semantic* analysis. The back end is responsible for the actual code generation and optimizations. The compiler written in this project made use of a couple of compiler construction tools to implement both the front end and the back end. *JFlex* (Klein et al. 2005) was used to handle the lexical analysis, that is scanning of tokens from the source code. *Beaver* (Demenchuk 2006) was used to parse the tokens received from the scanner and perform the syntactic analysis. Finally, *JastAdd* (Hedin and Magnusson 2003) was used to provide reference attribute grammars and static aspect oriented programming in order to efficiently and modularly be able to constructs parts of the semantic analysis and the back end code generation.

2.1 Java bytecode

Writing a compiler back end to support Java bytecode allows the code to run in the JVM. Being able to run code in the JVM is a great advantage due to the fact that the JVM has been ported to many devices and operating platforms. In other words, the extension of a Java bytecode back end would effectively make any language platform independent as long a the JVM is available for the platform.

Java bytecode has similarities with many other low level languages, for example the common arithmetic instructions such as *add*, *mul* and *sub* are all present (Lindholm et al. 2013, Chapter 6). The same goes for instructions to call methods and return from them. However, Java bytecode is also very different from many low level languages, such as x86- and ARM assembly. This is due to the fact that a lot of the instructions actually map directly to the high level constructs seen in Java. For example the *new* and *instanceof* instructions which have the same purpose as in the Java language. Another difference is that Java bytecode lack the common move-instructions often used in low level languages to move data across registers.

One more thing about Java bytecode that is relevant to the back end implementation is that there is no official standalone assembler for generating runnable Java bytecode from a textual representation of it. However, there are several third party solutions to this, for example *Jasmin* (Meyer).

2.2 The Java Virtual Machine

The JVM has some important features relevant to the back end implementation. One thing is that there are very few registers in the JVM, or more specifically, there are no general purpose registers usable by the programmer. The JVM instead expects the bytecode to make use of the stack in order to store intermediate calculations, return values and function arguments (Venners 1996).

Another feature of the JVM is that there is no way for the bytecode programmer to access direct memory locations of the program loaded into the JVM without the use of a special class (Katsov 2012). The bytecode instead makes use of something called *local variables* in the JVM, which have the ability to store the different data types available in the JVM. These local variables are reset for each method and is counted from index zero and upwards. However, the first n local variables are reserved for the method arguments (Lindholm et al. 2013, Chapter 2.6).

The final important feature is the fact that the JVM takes complete control of the program counter and stack pointer, and there is no way to manipulate these registers through Java bytecode (Lindholm et al. 2013, Chapter 2.5).

2.3 Jasmin

Due to the lack of any official assembler for Java bytecode, a third party tool called Jasmin can be used for exactly that purpose. Jasmin takes as input the textual representation of the Java bytecode describing a class. It will then output a `.class`-file runnable in the JVM.

Jasmin extends upon Java bytecode by introducing assembler directives and labels for jumps. These assembler directives are used to for example specify which part of the code is a method or a field, but it can also be used to specify limits for the stack and number of local variables in the JVM (Meyer 1996).

In Jasmin, the Java types when used in the context of method declarations or use of objects use a specific name. The ones used by this compiler are the following:

- `int` - `I`
- `boolean` - `Z`
- `float` - `F`
- An object - `L`*class_name*`;`

3. The front end implementation

The first thing that was to be done in the project was to extend the SimpliC front end with support for the new language constructs. Each of these constructs are presented below along with more details of how they were implemented.

3.1 Floats and booleans

The support for floats and booleans was implemented by constructing tokens for `float` and `bool` strings in the JFlex scanner, thus adding them as keywords to the language. The implementation of the actual use of these data types was also defined as tokens in JFlex, `true` and `false` were made to return boolean types. The floats were defined with the regular expression:

```
([0-9]+ "." [0-9]*) | ("." [0-9]+)
```

This allowed floats to be used without unnecessary zeros either before or after the dot, but having at least one digit.

Afterwards, since the only primitive data type available in the base language was integers, there needed to be some additional type checks implemented in the front end. This was done by adding types for the new data types as well as a type called `TypeMismatch` which is received when two different primitive types are compared in a program. Using this approach the errors could easily be collected during the semantic analysis with JastAdd collection attributes.

Since the SimpliC language does not support type casts, several predefined functions were defined in order to convert between the new data types that were introduced.

3.2 Global variables

In order to support global variables, the abstract grammar describing the program structure had to be modified. Previously, the start node `Program` only consisted of several function declarations and no variables. The `Program` node was thus changed to consist of a list of the abstract node `GlobalDecl`, describing a global declaration. Every node describing a function, struct or variable defined or declared at a global scope were made to inherit from `GlobalDecl`.

When the abstract grammar and related lexical and syntactic analysis were done, all that needed to be done to complete the support for global variables was to extend the lookup pattern for ID uses so that declarations for variables also can be found at global scope. This was done using reference attribute grammars to state that if no declaration was found within a function, then it should do a lookup at the global scope as well.

3.3 Structs

The final and most complex language construct to be added was the support for structs. The main difficulty of structs resided in the fact that declarations for the struct member variables resided inside of the structs own compound statement, which is not visible from the rest of the program.

The structs were designed to be similar to the ones found in the C language, but with the exclusion of some more complex and optional functionality. The abstract grammar for the structs are simply defined as:

```
StructDecl : IdDecl ::= Declaration*;
```

Which is stating that the struct in itself is a declaration for itself, but it also contains many other declarations (i.e. the member variables).

An example of a complete struct in the language may look like the following:

```
struct S {
    int a;
    float b;
    S s;
};
```

As seen, structs are declared using the `struct` keyword and then an ID to name the struct type. The struct can then be declared for example as `S myS;` and the member variables can be accessed by writing for example `myS.a` or `myS.s.b`. When an ID use which accesses member variables is used, a `StructUse` node is constructed in the abstract syntax tree. The `StructUse` node is defined to inherit from `IdUse`, as well as having an additional `IdUse` to represent the field member. In JastAdd abstract grammar this is written as:

```
StructUse : IdUse ::= Field:IdUse;
```

This design allowed for chained uses of member variables while still retaining the use of the `IdUse` node which has a well established lookup pattern.

However, the lookup pattern for `IdUse` is not enough due to the fact that member variables are declared in their own scope in the struct definition. It may also be the case that an intermediate `StructUse` in the member access chain is not declared. Due to this a more elaborate approach was needed in order to implement the structs during semantic analysis. The chosen approach to this was to introduce a new method evaluated for `StructUse`s during the declaration lookup. This method was called `fullName()` and returned a string representing the "full name" of the `StructUse`. This string could then be split up and recursively be evaluated in the usual lookup pattern for ID uses. To provide and example of the functionality of `fullName()`, consider the `StructUse` `a.b.c.d`, where `a`, `b` and `c` are instances of the structs `S1`, `S2` and `S3` respectively. The resulting string for the `StructUse` when it calls `fullName()` would be the string `S1.S2.S3.d`. If any member along the chain is missing a declaration, then that member would be replaced by the string `<UknownStruct>`, for example if `c` was undeclared in the previous example, the resulting string would have been `S1.S2.<UnknownStruct>.d`. Note that the use of `<UnknownStruct>` will not conflict with any existing struct declaration due to the fact that the special characters `<` and `>` can not appear in a SimpliC program.

In addition to this, the structs also have their own lookup methods in order to make sure that the declaration of the member variables are unique and correct. However the lookup will only look at the declarations within the struct definition since structs members are in their own name space.

Finally, a new keyword `delete` was introduced for structs. This was done in order to be able to free the memory on the heap, as there is no support for any `null` keyword in the SimpliC language. This keyword was implemented as a statement which accepts a variable declared with a struct type.

4. The initial back end approach

When implementing the back end we considered two different approaches. The first one was to generate Java bytecode directly, the second one was to generate a textual representation of the bytecode which could later be assembled into the actual bytecode. Since the latter approach required a third party assembler, a decision was made to try doing the first alternative.

Generating Java bytecode directly proved to be more difficult than initially assumed. This was mainly due to the fact that the JVM requires a lot of additional information in the `.class`-files aside from the actual bytecode in order to be able to run it. As a result all of this extra information needs to be generated by the back end.

The contents required in a `class`-file is documented by the Java authors (Lindholm et al. 2013, Chapter 5). The most interesting content is the constant pool. This structure contains a lot of information about the class described by the `class`-file, and contains UTF-8 strings of all class and method names used in the class, among other things.

An attempt was made to implement the generation of data needed in the `class`-files. The attempt was partly successful as the back end was able to generate a boilerplate `class`-file able to run bytecode inside the Java `main` method. While this approach worked, it ended up needing a lot of generation of boilerplate code for every new construct. Because of this, this approach was abandoned since it would just result in a lot of uninteresting data generation for the JVM. The original thought was that the focus of the project should lie in the generation of the bytecode, not the extra data needed by the JVM, even though it is important. A decision was thus made to switch the back end implementation to instead make use of the third party Java bytecode assembler Jasmin.

5. A back end with Jasmin

Using Jasmin allowed the back end to be implemented in a very similar way to the already existing x86 back end for the base SimpliC language, that is the generated code is represented as a readable text suitable to be used with an assembler. In the following subsections, the implementation of each of the major language constructs will be presented.

5.1 Java boilerplate code

Jasmin and the JVM expects the bytecode to follow the Java standards, such as requiring the declaration of the `main`-method to exist and to be declared with public and static identifiers. This boilerplate code gets generated by the compiler by creating a new `class`-file containing a class named `SimpliC`, this class contains the Java `main`-method and the obligatory constructor calling the constructor of the Java class `Object`. The reason the `Object` constructor is called is due to the fact that all Java classes inherit from this class, even if it is not visible in the source code for the class.

5.2 The global variables and functions

The first things that needed to be generated from the SimpliC language were the global variables. This was due to the fact that global variables were implemented as the public and static fields of the class, and Jasmin requires all fields to be generated before any methods.

When the global variables are generated, the compiler moves on to generate the functions and global structs. The functions are generated as Java methods with public and static identifiers. Jasmin requires the methods to specify the maximum limit of the method stack and local variables with assembler directives. In this compiler these values are hard-coded to be sufficiently large for all of our test cases. In a more elaborate compiler these values might be able to be computed for better efficiency.

5.3 The structs

Structs were chosen to be represented as a normal Java classes where the member variables are represented as public fields in that class. This was the most reasonable representation since Java does not have any support for structs, but structs can be seen as a class with only public members. Since a Java `class`-file can only contain one class definition, the generation of each struct must use a new file stream and produce a new `class`-file containing the obligatory constructor. The need for new files actually ends up making the implementation of structs defined inside of functions much easier, as they are generated in a separate file and will not interfere with the other code inside the function.

Accessing struct members can be done by looping through a `StructUse` until the final `IdUse` is found. For each step in the loop, the struct object is loaded from it's location with the `aload`/`getstatic` or `getfield` instructions depending on if it is a local/global declaration or a struct member.

The actual objects created by defining a variable with a struct type are created by using the `new` instruction in Java bytecode. This `new` is the same as the one used in the Java language, and will thus construct the object on the heap of the JVM. To be able to free the memory used by a struct variable, the delete keyword is used. This will set the object reference to null so that the garbage collector will be able to remove it from the heap.

5.4 Jump constructs

The jump constructs, that is `if`- and `while`-statements were implemented using the bytecode instructions `ifeq` and `goto`, along with Jasmin labels. Since labels needed to be unique for each method, JastAdd reference attribute grammars were used to calculate a unique name based on the source code line number. This functionality was actually already present in the base SimpliC language, but slightly modified and extended due to the extensions, but the idea was the same.

5.5 Arithmetic operations

The arithmetic operations add, subtract, multiply, divide and modulo were all easily implemented with corresponding Java bytecode instructions. The values used in the calculations are simply put on the stack before calling the arithmetic instruction.

5.6 Boolean comparisons

The implementation of boolean comparisons became a bit more complex due to the new data types introduced. Java bytecode contain different comparison and jump methods depending on the type of the input. For integers and object types the instructions `if_icmpxx` and `if_acmpxx` could be used respectively. The `xx` part is replaced by the postfixes `eq` and `ne` in the case of `if_acmpxx`, and `eq`, `ge`, `gt`, `le`, `lt` and `ne` in the case of `if_icmpxx`. In these cases all different supported comparisons are supported by the postfixes. However, there is a lack of postfixes to support all comparisons for the float type in Java bytecode. In the case of float types, only the instructions `fcmpl` and `fcmpg` can be used to compare *less than* and *greater than* respectively. These instructions can be combined with the more general instruction `ifxx`, which has the same postfixes as the `if_icmpxx` instruction, in order to provide better float comparisons even though floats should not be compared with equality.

5.7 Predefined functions

The predefined functions were hard-coded directly into the back end generation, constructing all of the predefined functions as normal Java methods.

The predefined functions that were defined for the language were:

- `print()` - Prints argument to standard out. Accepts ints, floats and bools.
- `readInt()` - Reads from standard in. Accepts ints.
- `readFloat()` - Reads from standard in. Accepts floats.
- `readBool()` - Reads from standard in. Accepts booleans as the strings ”true” or ”false”.
- `trunc()` - Casts a float to an int.
- `intToBool()` - Casts an int to a boolean.
- `floatToBool()` - Casts a float to a boolean.

6. Evaluation

The performance of the language was evaluated with three different tests using binary trees. All test cases were run three times, and then the mean value of the execution time was noted. The execution time was measured with a bash script calculating the difference between start time and end time of the actual execution. The results were compared with the result of two other groups who had done the same project but with an LLVM back end instead of a Java bytecode back end. The three tests were the following:

- Test 1: To build a tree recursively, and then iterate through the nodes.
- Test 2: To build a tree iteratively, and traverse it recursively.
- Test 3: To build a tree statically using variables.

The tests were done for different sizes of the trees and on the same computer. The result for the first test and second test is presented in Table 1. The result for the third test is presented in Table 2. All times include the startup time for the JVM.

Size	2000	20000	200000	200000	8000000
Test 1	97 ms	123 ms	227 ms	2094 ms	15328 ms
Test 2	97 ms	90 ms	162 ms	1039 ms	6042 ms

Table 1. Execution time for the recursive build, iterative traverse in Test 1, and vice versa in Test 2. Size is specified in number of tree nodes used.

Size	20	100	200	500
Test 3	86 ms	87 ms	84 ms	Stack overflow

Table 2. Execution time for the statically built tree in Test 3. Size is specified in number of tree nodes used.

Similarly, the results from the first and second tests from the two other groups using LLVM are presented in Table 3 and Table 4 respectively.

Size	2000	20000	200000	2000000	8000000
Test 1	4 ms	9 ms	53 ms	409 ms	1593 ms
Test 2	4 ms	10 ms	38 ms	Segfault	-

Table 3. Execution time for Test 1 and Test 2 for the first of the other groups. Size is specified in number of tree nodes used.

Size	2000	20000	100000	2000000	8000000
Test 1	-	-	-	-	-
Test 2	3 ms	9 ms	14 ms	Segfault	-

Table 4. Execution time for Test 1 and Test 2 for the second of other groups. Size is specified in number of tree nodes used.

Finally, the results from third test for the two other groups can be seen in Table 5 and Table 6.

Size	20	100	200	2000	20000
Test 3	5 ms	5 ms	5 ms	3 ms	-

Table 5. Execution time for Test 3 for the first of the other groups. Size is specified in number of tree nodes used.

Size	20	100	200	2000	20000
Test 3	4 ms	4 ms	4 ms	4 ms	-

Table 6. Execution time for Test 3 for the second of the other groups. Size is specified in number of tree nodes used.

The first two tests completed successfully for the Java bytecode implementation. In contrast, both of the LLVM back ends had problems with Test 2 for larger tree sizes, with both resulting in segmentation faults. The tests thus showed that the generated code handles recursion and large data structures well. However, the execution time using an LLVM back end turns out to be much faster than the Java bytecode one. This can be due to various factors, such as the fact that code is generated differently, but also due to the fact that LLVM does a lot of optimizations on the generated code before making it executable (Lattner and Adve 2004). It is also important to note that for the faster execution times the JVM startup time will have a greater impact on the result. No measurement was taken on this startup time, but it was estimated to be around 70-90 ms on the computer in question.

Test 3 yielded some rather bad results in the case of the Java bytecode back end. The code failed at very small tree size of 500 nodes with a stack overflow. This might have been able to be fixed by increasing the size of the stack for the JVM or in Jasmin. But this was never tried during the evaluation. However, the code executed fast for the working smaller tree sizes.

As for the LLVM groups, they both yielded similar results in Test 3, managing much larger tree sizes than the Java bytecode back end. Both groups had trouble compiling the final test with 20000 nodes, as it had not completed the compilation even after 15 minutes.

7. Related work

Due to the small scale of the project, most of the related work is done on larger systems taking many more aspects into consideration. A lot of focus is put on optimization in many works, which is something not taken into much consideration in this project.

One of the works in the realm of optimizations is referenced in (Vallée-Rai et al. 2000). Which is discussing the feasibility of using the Soot framework to optimize Java bytecode. This is interesting because the Soot framework has support for Jasmin, the low lever intermediate code generated by the compiler in this project. This might be able to be used to provide more optimized code for SimpliC.

Another work, more similar to this project, is referenced in (Benton et al. 1998). In this paper a Java bytecode back end is implemented for the Standard ML language, with many interesting features because of Standard MLs differences due to being a functional programming language.

Another interesting work where a language called X10 is provided with a back end for Java bytecode is referenced in (Takeuchi et al. 2011). This work is interesting and share common problems as seen in this project, such as the implementation of structs in Java bytecode. This work took a similar approach and implemented structs as Java classes, but those classes implemented an interface to make it work better with their system.

8. Concluding discussion

In this project the base language SimpliC which was developed in a previous course was extended with new language constructs and a new back end to support Java bytecode. The new back end allowed SimpliC programs to run on the Java Virtual Machine.

All constructs were successfully implemented, including structs which Java lacks official support for. However, most of the code runs very slow as program complexity increases. This is most likely due to the lack of optimizations done on the bytecode. Though slow, the code handles large structures and recursion very well as seen in the evaluation. As opposed to the LLVM back ends the project was compared to. The generated code does not appear to handle to many variables on the stack however, as stack overflows start occurring for a small number of allocated variables. This could perhaps be easily fixed by tweaking the stack size in either Jasmin or the JVM.

As the results show, it is most likely more efficient to implement a good LLVM back end than a Java bytecode one, due to the better speed and options. LLVM even has support to output Java bytecode, which is most likely better than the code generated by the compiler in this project due to all optimizations in LLVM.

This project does not really have any real world interests, as there are already many languages out there which provide more and better features. Though it is interesting for learning more about compiler construction. Further improvements to this could be to add optimizations, maybe through Soot as discussed in the related works section. Or to extend the language with more unique language constructs to make it a more interesting alternative for actual use.

References

N. Benton, A. Kennedy, and G. Russell. Compiling standard ML to java bytecodes. In M. Felleisen, P. Hudak, and C. Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 129–140. ACM, 1998. doi: 10.1145/289423.289435. URL `http://doi.acm.org/10.1145/289423.289435`.

A. Demenchuk. Beaver-a lalr parser generator, 2006.

G. Hedin and E. Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1): 37 – 58, 2003. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(02)00109-0. URL `http://www.sciencedirect.com/science/article/pii/S0167642302001090`. Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

I. Katsov. Tricks with direct memory access in java, 2012. URL `https://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/`. Accessed: 2015-12-29.

G. Klein, S. Rowe, and R. Décamps. Jflex-the fast scanner generator for java. *online source*, 2005.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013. ISBN 0133260445, 9780133260441.

J. Meyer. The jasmin bytecode assembler.

J. Meyer. Jasmin user guide, 1996. URL `http://jasmin.sourceforge.net/guide.html`. Accessed: 2015-12-29.

C. O. Nutter, T. Enebo, N. Sieger, O. Bini, and I. Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 2011.

M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004.

S. Pedroni and N. Rappin. *Jython essentials*. " O'Reilly Media, Inc.", 2002.

M. Takeuchi, Y. Makino, K. Kawachiya, H. Horii, T. Suzumura, T. Suganuma, and T. Onodera. Compiling x10 to java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 3:1–3:10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0770-3. doi: 10.1145/2212736.2212739. URL `http://doi.acm.org/10.1145/2212736.2212739`.

R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25 - April 2, 2000, Proceedings*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2000. doi: 10.1007/3-540-46423-9_2. URL `http://dx.doi.org/10.1007/3-540-46423-9_2`.

B. Venners. The lean, mean, virtual machine. *Java World*, 1996.