C Compiler Targeting the Java Virtual Machine

Jack Pien

Senior Honors Thesis (Advisor: Javed A. Aslam)

Dartmouth College

Computer Science Technical Report PCS-TR98-334

May 30, 1998

Abstract

One of the major drawbacks in the field of computer software development has been the inability for applications to compile once and execute across many different platforms. With the emergence of the Internet and the networking of many different platforms, the Java programming language and the Java Platform was created by Sun Microsystems to address this "Write Once, Run Anywhere" problem. What sets a compiled Java program apart from programs compiled from other high level languages is the ability of a Java Virtual Machine to execute the compiled Java program on any platform, as long as the Java Virtual Machine is running on top of that platform.

Java's cross platform capabilities can be extended to other high level languages such as C. The main objective of our project is to implement a compiler targeting the Java Platform for a subset of the C language. This will allow code written in that subset of C to be compiled into Java Virtual Machine instructions, also known as JVM bytecode, which can then be executed on a Java Virtual Machine running on any platform.

The reader is assumed to be intimately familiar with compiler construction, the use of the **flex** scanner generator, the use of the GNU **bison** parser generator, and the structure and implementation of the Java Virtual Machine.

1 Introduction

The main focus of our project is to implement a compiler for a particular subset of the C programming language which targets the Java Virtual Machine. The compiler is able to read in a *sourceFile*, written in the subset of C implemented, and compile that code to a JVM *.class* file called *targetFile*.class. The *targetFile*.class would then be able to be executed on a Java Virtual

Machine running on any platform. We feel that the particular subset of the C programming language chosen was sufficient in providing the essential backbone for allowing further extensions of other C language grammars to be added in the future.

Many different tools were used to implement the different segments of the compiler. The scanner was created using **flex** and the parser was created using **bison**.[1] The code generator was written in C++. It consists of a symbol table to look up functions and variables that need to be accessed. Along with the symbol table is a data structure representing the JVM Constant Pool table that holds all the Constant Pool entries that are needed by the Java Virtual Machine to execute the compiled C code. Given a *sourceFile* to compile, the compiler would first scan through the file creating a list of tokens. From the token list, a syntactical parse tree is created with the parser. The parse tree is then handed to the code generator for code generation into JVM bytecode.

Many major issues concerning the differences between the JVM and other stack based machine platforms came up while implementing the compiler. The main problem was coming up with a way to compile a non-object oriented language such as C into a target machine language that was designed to only support the fully object oriented nature of Java. Our compiler design also had to deal with the autonomous memory management of the JVM and the unique way the JVM accesses variable and function declarations.

2 Technical Description

2.1 C Programming Language Subset that was Implemented

The subset of C that was implemented into our compiler includes global variable declarations, local variable declarations, function declarations, *if-else* and *if* statements, *while* loop iterations, and *return* statements.

Variables can be declared as a single entity or as an array of a size indicated by a numeric value. The compiler only handles variable declarations of the type *int*.

Functions can return either nothing (i.e. *void*) or an *int*. The only types of arguments that can be passed into a function call are *int*'s or a reference to an *int* array.

The exact grammar that was implemented is indicated as follows:

[1] For more information on compiler construction and the use of **flex** and **bison** refer to [1].

Program	:	*DeclarationList*
DeclarationList	:	*Declaration DeclarationList*
	|	*Declaration*
Declaration	:	*VariableDeclaration*
	|	*FunctionDeclaration*
VariableDeclaration	:	*TypeSpec* **ID ;**
	|	*TypeSpec* **ID** [**NUM**] **;**
TypeSpec	:	**INT**
	|	**VOID**
FunctionDeclaration	:	*TypeSpec* **ID** (*Params*) *CompoundStatement*
Params	:	*ParamList*
	|	**VOID**
ParamList	:	*Param* , *ParamList*
	|	**VOID**
Param	:	*TypeSpec* **ID**
	|	*TypeSpec* **ID** [ ]
CompoundStatement	:	**{** *LocalDeclarations StatementList* **}**
LocalDeclarations	:	*VariableDeclaration LocalDeclarations*
	|	
StatementList	:	*Statement StatementList*
	|	
Statement	:	*ExpressionStatement*
	|	*CompoundStatement*
	|	*SelectionStatement*
	|	*IterationStatement*
	|	*ReturnStatement*
ExpressionStatement	:	*Expression* **;**
	|	**;**
SelectionStatement	:	**IF** (*Expression*) *Statement* **ELSE** *Statement*
	|	**IF** (*Expression*) *Statement*
IterationStatement	:	**WHILE** (*Expression*) *Statement*
ReturnStatement	:	**RETURN ;**
	|	**RETURN** *Expression* **;**
Expression	:	*Variable* = *Expression*
	|	*SimpleExpression*

Variable	:	**ID**
	:	**ID** [*Expression*]
SimpleExpression	:	*AdditiveExpression RelativeOperator AdditiveExpression*
	|	*AdditiveExpression*
RelativeOperator	:	**LESSEQUAL**
	|	**LESSTHAN**
	|	**GREATERTHAN**
	|	**GREATEREQUAL**
	|	**EQUAL**
	|	**NOTEQUAL**
AdditiveExpression	:	*AdditiveExpression AdditiveOperator Term*
	|	*Term*
AdditiveOperator	:	+
	|	−
Term	:	*Term MultiplicativeOperator Factor*
	|	*Factor*
MultiplicativeOperator	:	∗
	|	/
Factor	:	**(** *Expression* **)**
	|	*Variable*
	|	*Call*
	|	**NUM**
Call	:	**ID (** *Args* **)**
Args	:	*ArgList*
	|	
ArgList	:	*Expression* **,** *ArgList*
	|	*Expression*

Those that are bold faced are terminals which can either be tokens that have been extracted from the scanner or an ASCII character. The string equivalents of the tokens are listed in the table below:

Token	String Literal
ID	<identifier name>
NUM	<number>
INT	int
VOID	void
IF	if
ELSE	else
WHILE	while
RETURN	return
LESSEQUAL	<=
LESSTHAN	<
GREATERTHAN	>
GREATEREQUAL	>=
EQUAL	==
NOTEQUAL	!=

2.2 Structure of the Compiler

The compiler has mainly three parts associated with it: a scanner (created with **flex**), a parser (created with **bison**), and the code generator.

First, the scanner scans the C code and creates a list of tokens associated with the code. In addition to creating tokens, it also stores the numeric value associated with a NUM token and the identifier name of an ID token.

The parser then parses through the token list following the rules of the C language grammar being implemented. During the parse, the parser will verify the syntactical structure of the token list according to the grammar as well as produce a tree data structure representing the syntax of what was parsed. Essentially, the root of the tree would be the start point of the grammar (*Program*) and the leaves would represent the tokens or terminals of the grammar.

Finally, The code generator would then run through this tree and generate the correct Java Virtual Machine bytecode needed to execute the particular C code represented by that section of the tree. Semantic verifications are made during the operations of the code generator. The code generator would also have to recognize function calls from external C library (i.e. *stdio.h*) and generate the JVM bytecodes that handle linking to the other JVM ċlass files needed to execute those functions.

During the code generation segment, the code generator would be writing bytecode to four temporary files which contain the different segments of the complete JVM *.class* file. When code generation is completed, the four temporary files would be merged into the final *targetFile*.class file.

2.3 Code Generator Implementation

While implementing the code generator, it was necessary to understand the Java Virtual Machine's similarities to and differences from other stack based machine platforms (i.e. PowerPC, x86, MIPS, etc.).

One of the main differences is the JVM's use of a data structure called the Constant Pool table. The JVM uses the Constant Pool table to represent the various classes, functions, and variables in the *.class* file. Since the entries of the Constant Pool are part of the final JVM *.class* file, it is essential for the compiler to be able to generate the Constant Pool table and keep track of all its entries throughout the code generation.

Another difference is the JVM's method of referencing variables and functions. One of the qualities and restrictions in the design of the JVM is its ability to hide memory references and prohibit memory reference manipulations. By not being allowed to know or to access the memory locations of declared variables and functions, the compiler has to use the methods the JVM gives to reference the variables and functions. Also, the referencing of a global variable is very different from the referencing of a local variable. Since the design of the JVM is to mainly support the fully object oriented nature of Java, global variable declarations are not allowed, therefore the compiler has to find a way to represent a global variables.

Given a way to represent the JVM Constant Pool and to reference variables and functions, a number of other issues must be considered during the code generation segment. Although the JVM is a stack based machine making the bytecode generation directly effecting the frame (or Java Operand Stack) similar to other stack based machines, administrative code related to memory management, memory allocation, and program counter changes are extremely different from the administrative assembly code of other stack based machines. These were the major issues that needed to be dealt with in order to successfully generate the JVM bytecode from a given C file.

2.3.1 Implementing the Constant Pool Table

During the execution of a *.class* file, the Java Virtual Machine relies heavily upon that particular *.class* file's Constant Pool to represent the various string constants, class names, field (variable) names, method (function) names, and other references in the file. In order to reference certain classes, variables, and functions, the JVM needs to access the various entries in the Constant Pool table in order to obtain information about a certain class, function, or variable.

The compiler would need to create and organize the Constant Pool table before generating the JVM bytecode representing it upon the completion of the code generation. Since it is necessary to access the entries in the Constant Pool and to insert new Constant Pool entries fast, the dynamically growing array became the ideal data structure for implementing the Constant Pool table within the code generator. Each element in the array represents a Constant Pool entry and all the information associated with that particular entry.

Constant Pool entries are added as the code generator parses through syntactical parse tree given by the parser. Each entry is added to the end of the Constant Pool table. The format of each entry depends on the type of the entry [3, pages 92–101]. The Constant Pool data structure is accessed whenever the code generator comes across a function call or a global variable access. The code generator would look into the Constant Pool and generate the JVM bytecode needed to access the Constant Pool entry associated to that function or variable.

Since the JVM *.class* file format requires the Constant Pool to precede the executable bytecode, it is necessary to write the JVM bytecode representing each Constant Pool entry to a temporary file upon completing the code generation segment when all the essential Constant Pool entries would have been realized.

2.3.2 Referencing Variables and Functions

Besides not being able access memory locations, one of the reasons why referencing variables and functions is different from other stack based machines is due to the object oriented nature of the Java Virtual Machine. Being fully object oriented would mean that at the global scope, there are only objects and classes. Since function declarations and variable declarations must be made within those objects and classes, globally declared functions and variables are impossible to be represented in JVM. The method that was chosen to solve the problem of compiling globally declared functions and variables of a non-object oriented language such as C into the Java Virtual Machine is to have the compiler “wrap” the compiled code within a fabricated class. The details of creating this class will be discuss in 2.3.3 but the solution essentially makes all global variable declarations and function declarations, field and method declarations (respectively) within that class.

With a class enclosure around the compiled code, referencing a global variable or a function is now just a question of referencing the Constant Pool entry associated with the field or method within the enclosed class that represents the global variable or function. Since the JVM allows for the static declarations of the fields and methods within a class, an instance of a class is not needed to access a field or method that has been declared static. Thus when the code generator needs to generate bytecode that references a global variable or function, it would just generate the bytecode used to access the index of the Constant Pool entry representing that global variable or function.

One the other hand, local variables are not referenced through the Constant Pool. The JVM definition of local variables include those variables declared within the scope of a function as well as the arguments of that particular function. For each frame (or Java Operand Stack) allocated for a function call, four bytes or one word is allocated for each function argument and then for each local variable declaration (two words for *double* and *long* variables) in that function. The i minus first index reference, for all i greater than or equal to one and less than or equal to the total number of arguments and local variables declared, refers to the i-th word from the top of the frame; the top of the frame holds the first word representing the first argument and the bottom of the frame refers to the last word representing the last local variable declared.[2] The code generator uses that same index number to generate bytecodes to access the location or value of a variable within that function call.

2.3.3 Generating Bytecode

As briefly explained in 2.3.2, the C code in the *sourceFile* has to be to be enclosed within a class since the Java Virtual Machine was designed for the object oriented language, Java. Therefore, unlike other stack-based machine platforms, the code generator first virtually encloses the *sourceFile*

[2] This is assuming that the JVM method is declared static. For more information on local variables, refer to [3, pages 66].

code within a class, which it names the *targetFile* class. All functions and global variables declared within *sourceFile* thus becomes publically and statically declared methods and fields of the class *targetFile*. The code generator then adds all the Constant Pool entries into the Constant Pool structure necessary for the JVM to acknowledge the existence of the *targetFile* class. Also, the bytecodes to initialize the *targetFile* class are written to a temporary file dedicated for this *targetFile* class constructor.

Once the administrative task of creating the *targetFile* class is completed, the code generator can begin to parse for global variable and function declarations which will be converted to member variable and function declarations of that class. The code generator stores all variable and function identifiers in a symbol table that's implemented as a stack of hash tables. Each level of the stack represents a different scope level and as a scope begins or ends, a new hash table is added to the top or popped off of the symbol table stack. The identifier name is added into the current scope of the symbol table, along with all the related function or variable information. When an identifier needs to be looked up in the symbol table, the information that was added along with the identifier name will be used to access the function or variable of that identifier.

Currently, there are three types of identifiers that are implemented in the compiler: a global variable, a local variable, and a function. When the code generator comes across a global variable declaration, the Constant Pool entries needed to access the variable are created and added the compiler's Constant Pool data structure. All the JVM bytecode needed to make the JVM aware of this global variable's existence is then written to a temporary file dedicated for global variables. Essentially, the bytecode tells the JVM that this particular global variable is a field of a particular type that's within the class *targetFile*. If the global variable is an array pointer, then the bytecodes needed to initialize the array elements are added to the *targetFile* constructor (to the temporary file dedicated for the constructor of the *targetFile* class. So the array elements get initialized when the targetFile class gets initialized). The code generator then finally stores the identifier name of the variable into the symbol table. The identifier name of the variable and the reference index to the Constant Pool entry referring to the variable is then added to the symbol table.

The code generator treats a function declaration similarly. Constant Pool entries relating to the identification of this function (i.e. argument types, return types, etc.) need to be created as well as Constant Pool entries needed to call this function in the future. After these entries are created and stored into the compiler's Constant Pool data structure, the JVM bytecodes related to the operation of the function are written to another temporary file dedicated for function declarations. Then the identifier name of the function, along with the Constant Pool reference information, the JVM bytecodes needed to to "prepare" for the execution of this function, the JVM bytecodes needed to execute this function after all the arguments needed to pass into this function have been pushed on to the frame (also known as the Java Operand Stack), and other Constant Pool entries needed for the JVM to execute the function, are added in to the symbol table.

If a local variable declaration is parsed out within a function, the code generator only needs to increment the total number of other local variables previously declared plus the number of arguments of the function.[3] This number becomes the reference index to access this local variable in the future. The code generator would then add the variable identifier name along with the reference index into the symbol table.

Most of the JVM bytecode related to frame operations and instructions operate similarly to the assembly language instructions of other machine platforms. The JVM allocates a frame (or Java Operand Stack) for each function call. Most of the bytecodes of a particular function pops values from the Operand Stack, operates on the values, and then returns a result to that same Operand Stack. A majority of the bytecode instructions make some use of the Operand Stack, including function invoking bytecode which pops the passing arguments (which passes by value) from the Operand Stack.

One of the main differences between generating code for the JVM and other stack-based machine platforms comes from the JVM's method of memory management and the way its program counter operates. Although similar to the other machine platforms in that the JVM is also stack and heap based and allocates frames (Java Operand Stacks) upon function calls, the JVM is also quite different since it performs all the memory managements automatically within the virtual machine and does not allow others to directly access, make changes in, or allocate memory. Since the allocation of memory for the stack, heap, and frames are all performed automatically by the JVM, the code generator does not need (nor is it allowed) to generate code that explicitly move or store the old stack or frame pointers upon invoking a function. The code generator only needs to generate

[3] Assuming the local variable is of type *int* which is 4 bytes or one word wide. Future implementations of types *long* and *double* would require incrementing the total count by two since they are two words wide.

the code that tells the JVM the amount of memory to allocate. The operation of storing and restoring the stack and frame pointers are all automatically performed by the JVM when function calls and function returns are made.

Likewise, the program counter cannot be changed directly by the code generator and is incremented and manipulated directly by the JVM. Therefore the bytecode used to jump to other parts of the program during conditional and iteration statements are done through bytecode counting rather than program counter manipulation. Instead of changing the value of the program counter, the code generator will generate bytecode to jump to a particular byte in the executing code in the function being executed.[4]

Since the format of the JVM *.class* file requires the Constant Pool information to appear first, then class field declarations (or our global declarations), and finally class method declarations (including the constructor), it was necessary to write the different code generations of each segment to different files. The temporary file for the constructor was needed since the declaration of global array variables can be made throughout the C code in the *sourceFile* so the compiler would need a way to be able to add bytecode to the constructor whenever it parses a global array variable declaration. When the code generation is complete, the four temporary files for the Constant Pool, global variable declarations, function declarations, and the constructor for our *targetFile* class are combined in the respective order into the complete *targetFile*.class Java Virtual Machine *.class* file.

[4] For more information on bytecode counting and jumping, refer to [3, pages 348–349].

3 Conclusion

With the number of different machine platforms in existence and the increasing influence of the Internet, it becomes more and more important to write applications that has the ability to run on all platforms. With the increasing popularity of the Java programming language as being a cross-platform language, our project gave the widely known C language this ability to "Write Once, Run Anywhere". Despite the fact that the Java Virtual Machine was designed for the Java programming language, we wanted to show that C can be efficiently compiled to the JVM.

Since the compiler is a one-pass compiler and the *sourceFile* is read through only once, the compiler executes relatively efficiently. All the code was written in C++ and the code generation segment of the compiler is object orientedly designed. Each object was designed to be small and efficient, at the expense of having less objects, but this makes the code extremely readable and easy to understand, which we thought is important to in motivating any future interests in our project.

Currently, any C code in the *sourceFile* which conforms to the grammar described in 2.1 can be compiled into JVM bytecode. Future extensions of the project include extending the implemented C grammar and adding C++ grammar to allow for class objects. More immediate grammar extensions include adding the *#include* rule. Currently, including our own version of a *stdio.h* library is hard coded into the compiler. The symbol table in the code generator automatically reads in a header file called *stdio.hjava* and adds the information in the file into the global scope of the symbol table. This file includes all the information (i.e. Constant Pool, executable bytecode, etc.) the symbol table needs to allow the functions, *int input()* and *void output(int)*, to be called.[5] These functions give the C programmer using the compiler an interface to the standard input and standard output. The format of the *.hjava* file is straight forward and very similar to the format of a method declaration in a JVM *.class* file.[6]

Other extensions to the compiler can be the implementation of a stronger semantic verifier within the code generator. Although, the JVM does semantic checking at the virtual machine runtime level, it would be more efficient and more predictable to also have more semantic checking performed at the compiler level.

In conclusion, we hope our project will spur future interests in combining the popularity of the

[5] The details of *int input()* and *void output(int)* are discussed in Appendix A.

[6] The structure of the *.hjava* header file is documented in the file *SymbolTable.cc*

C programming language with the cross-platform cability the Java Virtual Machine has to offer.

A User's Instructions

All the source code for our project is in the following directory until mid-July:

```
/usr/tahoe1/slash/Thesis/CodeGen/
```

To compile the code that incorporates a subset of the C programming language as described in 2.1, execute:

```
JVMcc targetFile < sourceFile
```

Where:

JVMcc: is the C compiler

sourceFile: is the C programming being compiled

targetFile: is the name of the JVM *.class* file (without the .class suffix) being compiled to

JVMcc reads in the redirected *sourceFile* and will parse, scan, and generate JVM bytecodes into a file it creates called *targetFile.class.* The *.class* suffix is attached to the end of *targetFile* to indicate that targetFile is a JVM classfile.

To execute *targetFile.class*, run the Java Interpreter with the *targetFile* as it's input file:

```
java targetFile
```

Note on the C code in the *sourceFile*

To read from standard in, call the function *int input()* which reads an integer input from the standard input and returns the integer read.

To write an integer to standard out, call the function *void output(int)* which writes the integer argument to the standard output.

References

[1] Louden, Kenneth C. *Compiler Construction: Principles and Practice.* PWS Publishing Company, Boston MA. 1997.

[2] Meyer, Jon and Downing, Troy *Java Virtual Machine.* O'Reilly and Associates, Inc., Sebastopol, CA. 1997

[3] Lindholm, Tim and Yellin, Frank *The Java Virtual Machine Specification* Addison–Wesley, Mountain View, CA. 1997