VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

SOFTWARE ENGINEERING STUDY PROGRAMME

Translating C into Java Bytecode
C kalba parašytų programų transliavimas į Java JVM baitų kodą

Farrukh Nabiyev

Supervisor : prof. Saulius Gražulis

Reviewer : Gediminas Rimša

Vilnius
2025

Acknowledgements

The author is thankful the Information Technology Research Center, Faculty of Mathematics and Informatics, Vilnius University, for providing the High-Performance Computing (HPC) resources for this research.

Summary

In chemo-informatics several popular libraries exist: OpenBabel [GCC*20], written in C++, and CDK [CYS*03], OpenChemLib [Joe18], Jumbo [Bob97], all written in Java. There is a need to process data from files with crystallographic information in these libraries. Crystallographic Information File/Framework (CIF) [HWS*06, BBB*16] specifies a standardized file format for the archiving and distribution of crystallographic information. It is promoted by the International Union for Crystallography (IUCr) [FWB*06]. To read information into programs from CIF files a COD CIF parser is needed. Unfortunately, there does not exist a high-quality parser for CIF files for Java language. Therefore, it is desirable to port existing CIF parsers written in C to a pure Java environment without using Java Native Interface (JNI). This thesis will investigate how to create a C to Java bytecode compiler.

The thesis investigates possible ways to translate C into Java bytecode. The main challenge is to simulate the C memory inside JVM, for that, we propose a single class which will simulate behavior of malloc() and free() functions. Those functions are custom and implemented only in this work. The developed compiler cannot compile entire COD CIF parser into Java bytecode, but it solves the gap between C and Java. The supported features of developed compiler are enough to cover pointers, functions, structures, method calls and successfully translate them into bytecode.

Keywords: C to Java compiler, cheminformatics, Crystallographic Information File (CIF), ANTLR, Java bytecode, ASM (Java library), CIF parser, simulated memory model.

Santrauka

Cheminėje informatikoje egzistuoja kelios populiarios bibliotekos: OpenBabel [GCC20], parašyta C++ kalba, bei CDK [CYS03], OpenChemLib [Joe18], Jumbo [Bob97], kurios visos parašytos Java kalba. Šiose bibliotekose reikia apdoroti failus, kuriuose yra kristalografinių duomenų. Kristalografinės informacijos failas (CIF) [HWS06, BBB16] apibrėžia standartizuotą failo formatą, skirtą kristalografinių duomenų archyvavimui ir sklaidai. Jį palaiko Tarptautinė Kristalografijos Sąjunga (IUCr) [FWB*06]. Norint iš CIF failų įkelti informaciją į programas, reikalingas COD CIF analizatorius. Deja, šiuo metu nėra aukštos kokybės CIF failų analizatoriaus, sukurto Java kalbai. Todėl pageidautina perkelti esamus CIF analizatorius, parašytus C kalba, į gryną Java aplinką, nenaudojant Java Native Interface (JNI). Šiame darbe bus nagrinėjama, kaip sukurti C į Java baitkodo kompiliatorių.

Šiame darbe nagrinėjami galimi C kalbos vertimo į Java baito kodą būdai. Pagrindinis iššūkis – imituoti C kalbos atmintį JVM aplinkoje. Tam siūlome vieną klasę, kuri simuliuoja malloc() ir free() funkcijų veikimą. Šios funkcijos yra individualios ir įgyvendintos tik šiame darbe. Sukurtas kompiliatorius negali išversti viso COD CIF parserio į Java baito kodą, tačiau jis sprendžia esamą atotrūkį tarp C ir Java kalbų. Palaikomos funkcijos leidžia dirbti su rodyklėmis, funkcijomis, struktūromis, metodų kvietimais ir sėkmingai jas versti į baito kodą.

Raktiniai žodžiai: C į Java kompiliatorius, cheminė informatika, Kristalografinės informacijos failas (CIF), ANTLR, Java baitinis kodas, ASM (Java biblioteka), CIF analizatorius, simuliuotas atminties modelis.

List of Figures

List of Tables

Contents

Introduction

The Crystallographic Information File (CIF) format is the most widespread standard for representing data in crystallography. It was introduced in the 1990s by Hall, Allen, and Brown [HAB91], from then CIF is used as the standard way of documenting and transferring crystallographic data. It is used by the International Union of Crystallography (IUCr). The structure of CIF files is well defined but needs special parser to interpret and manipulate it. Over the years, several parsers have been developed, including pycifrw [Hes06], vcif2 [Her08], and most notably the COD CIF parser, which is part of the Crystallography Open Database (COD) Tools. The COD CIF parser is known for its speed and flexibility, it does not only parse valid CIFs, but also corrects the formatting [MVB*16].

Despite the fact that this compiler is mature, it has a limitation. COD CIF parser was written in C programming language and therefore it cannot be used by modern Java based chemo-informatics libraries. This is because many prominent chemo-informatics tools—such as the Chemistry Development Kit (CDK), OpenChemLib, and JUMBO—are developed entirely in Java. Unfortunately, direct use of native C libraries from Java requires the Java Native Interface (JNI), which introduces significant complications. JNI code must be compiled and maintained separately for each target platform, undermining the portability and maintainability that Java normally offers. Moreover, improper use of JNI can lead to runtime crashes, memory safety violations, and debugging complexities, all of which are generally avoided in “pure Java” development.

The one way to enable chemo-informatics to use COD CIF parser is to implement it in Java language. However, it is not a reliable approach. The parser is under continuous development and change in one language implies reimplementation of the same change in other languages. Moreover, small changes which will be implemented in C will stack and in future the implemented compiler will lack most functionality. Also, if reimplementation will require the integration with other libraries, there is no guarantee that the same library will exist in both languages. Maintaining parallel implementations across different languages is not sustainable.

A better solution is to translate the implementation to another language automatically. It will guarantee that the original logic is preserved, and both implementations are up to date. This approach would require the development of a C-to-Java bytecode compiler. Compiler should understand and translate all C features used in COD CIF parser. The development of such compiler would not only solve the CIF parser integration issue but also open the possibility to translate and use other C-based scientific libraries within Java environments.

This thesis will investigate the possible ways to implement such compiler. The main purpose of this

work is to understand is this approach is possible to implement. The idea is to create a system that takes C source code and generates Java bytecode which is then run on JVM. The approach should not rely on JNI or other possible caller, which can execute C code and then feed the result into JVM.

The C Programming Language was created in the early 1970s in the name of the Unix operating system project. It was designed for efficiency, expressiveness, and low-level access to system services. The definitive reference, for over 10 years before being officially standardized by ANSI, was Dennis Ritchie and Brian Kernighan's book The C Programming Language [KRi88]. In the 1980s, C became one of the most widely used and fastest programming languages because of its portability. It is still used as a base of many contemporary systems, with particular use in scientific and embedded computing. But C does put a lot on the programmer, especially in matters like memory control and type security.

However, Java emphasizes modifiability and safety. JVM executes Java programs by first converting them into a bytecode format. Though, one of the most fundamental innovations of Java is its automatic memory management - the "garbage collector" mechanism relieves programmer from being responsible for tracking every last allocation of memory. Java developers get to think high level the logic of things and should be concerned less with managing low level resources. As a result, they have less chance for memory leaks or errors caused by wrong pointer manipulation.

Memory management in Java is automatically done by the garbage collector, which detects unused memory and frees it up. It runs in the background and periodically walks through the object reference graph to identify memory that is no longer reachable (i.e. referenced) so it can be released. This creates a reliability pattern with a performance tradeoff between the two. Contrast this with C, where the programmer must allocate and deallocate memory explicitly using malloc() and free(). This is often the root of subtle bugs and memory safety issues, particularly in large, long-running applications.

With a successful transformation of the COD CIF parser to Java bytecode, Java-based chemo-informatics tools can provide full support to CIF as well as its extensions (e.g., CIF2). A particularly useful application of this would be accessing crystallographic data in the Crystallography Open Database (COD). The COD is a publicly writable repository for thousands of crystal structures of small- and medium-size molecules [GDM*12]. Such data is crucial for numerous fields, such as drug design, material science, and structural chemistry. Incorporation of these data into the Java library would provide the researchers with tools of choice for their automated data analysis, visualization and simulation.

For example, the Chemistry Development Kit (CDK) contains classes, such as CIFReader, which allow CIF files to be read. Nonetheless, this support is partial and does not implement everything in the CIF specification or its successors, CIF2. These libraries could be improved with general support for

crystallographic data with a fully functional parser integrated like the COD CIF parser. This advance would enable new applications within cheminformatics, such structure-based screening, descriptor generation and machine learning.

The same can be said for other popular Java based projects like JUMBO, OpenChemLib, and Jmol that would highly profit from improved CIF parsing. Some libraries may provide a minimal CIF parser as a convenience tool, but without significant independent CIF expertise these functionalities tend to be built around (and thereby limited by) the overall functionality of the library, reducing the utility for a considerable fraction of crystallographic datasets. Automating C-to-Java compilation would overcome this limitation, enabling such support with little to no major rewrites of existing code bases.

Abstract In this talk, we describe our experience designing and building a C-to-Java compiler that generates Java bytecode, and we specifically focus on the issues and challenges of translating the COD CIF parser. Not only does this solution fulfill the pressing need for better CIF support in Java-based chemo-informatics but it also provides the basis for more generic interoperability between C libraries and the JVM ecosystem.

1. The goal and tasks of the research

The goal of this thesis is to investigate the feasibility of translating C code into Java bytecode in order to integrate legacy C libraries into Java-based environments. This work focuses on developing a prototype compiler capable of translating a selected subset of C language features into JVM-compatible bytecode. The motivation stems from the need to support integration of C-based scientific tools, such as the COD CIF parser, into Java-based cheminformatics libraries, while maintaining portability and safety. Although full support for the COD CIF parser has not yet been achieved, this thesis sets the foundation for such integration by implementing and evaluating the required infrastructure for a partial translation pipeline.

To achieve the goal the following tasks are planned:

1) Explore the subset of C language features needed for translation of COD CIF library.

2) Design compilation strategy for incomplete feature list used in COD CIF library.

3) Implement the small set of C features features in compiler.

1.1. Background

Figure 1 C compilation process (image was adapted from [Ter10]*)*

The compilation process consists of several stages. Figure 1 shows the stages through which the pipeline goes. Before compilation, preprocessor should handle includes and macros. The preprocessor spits out pure C code with some line number directives understood by the compiler gives object code as an output [Ter10]. The final stage is performed after the linker resulting in an executable file.

The Java programming language presents the possibility of compiling output that can run on any computer for which a Java virtual machine is provided. Using a virtual machine, the output can be recompiled at the execution platform by a compiler [Nal18]. The bytecode is not dependent on any platform and is not native code. JVM has a compiler which is known as JIT (Just In Time) [MDP*01]. JVM takes bytecode and gives it to the JIT and JIT compiler outputs machine-dependent native code [Nal18].

There exist three methods to translate Java language to C language. The first way is to take .c files and to generate a human-readable program written in the .java format. This method is called source-to-source translation [Pou89]. It often does translation well, but the difficulty of this method lies in the difference in semantics of the languages. Despite that, existing systems employ this technique. These

translators can be divided into two categories: partial translation which is completed by a human and total translation which gives an error on a large class of input files. The translator can not convert every possible C construct. An example system of the first category is Jazillian. It takes C source codes and produces readable Java code. However, it can translate only a small subset of the C language. Tools of the second category are c2j [Daw19], Ephedra [Mar02]. They provide great translation but none of them covers a large set of C language standards.

Another possible transition method is to generate machine code from .c files and then convert that machine code to a Java bytecode [Ora20]. This method is called binary-to-binary translation. One such translator is NestedVM [BAd09]. It does not deal with source code as input. This gives several advantages. It is independent of the source language. It can support any language for which MIPS [Moh20] targeted compiler exists. It uses compiled output files as its input. Therefore it skips semantic analysis. It does not deal with preprocessor usage and library locations. NestedVM supports all non-privileged instructions, floating point processor, add/multiply unit found on MIPS R2000 CPU.

Third option is to generate .class files compiling .c files. This method is called source-to-binary translation. In this translation compiler emits bytecode from the C source file. One of such translators is "JVM Back end" for GCC compiler known as egcs-jvm. But this compiler does not allow pointer math. Thus compiler fails for a huge class of programs. Another available way is the Java back end for the LCC [Dav14] compiler. The system is known as lcc-java. The main disadvantage of this tool is the lack of libc [Mic16] library support. The standard library responsible for the main system calls such as malloc, open, printf. After such a translation, very few programs can run without customization.

2. Review of existing tools

Language C was widely used before modern programming languages such as Java was introduced. Modern languages can use existing libraries written on C. Solution can be done through a Java Native Interface (JNI) that calls native code within a virtual machine. However, JNI cannot be used everywhere because of security and portability concerns. Methods called through JNI have a risk of heap corruption. Native code requires it to be compiled ahead of time on each architecture where it will be run. That restricts the use of JNI if architecture is not known beforehand. Another solution is to translate one programming language to another. When compiling source code, it is useful to represent it in machine independent way instead of direct compilation to machine code. Machine independent representation is called IR (Intermediate Representation). Common optimizations such as elimination of unused code, value to constant transformation is done on IR level. IR is used by compilers such as GCC (GIMPLE), Roslyn (CIL), LLVM (LLVM IR).

A way of representing C code is to use LLVM IR. This representation can be obtained using LLVM compiler. LLVM IR is an intermediate representation used by LLVM framework. LLVM front end clang translates a source program to an LLVM IR. Then the LLVM framework makes optimizations and later LLVM back end generates machine code. (Figure 2)

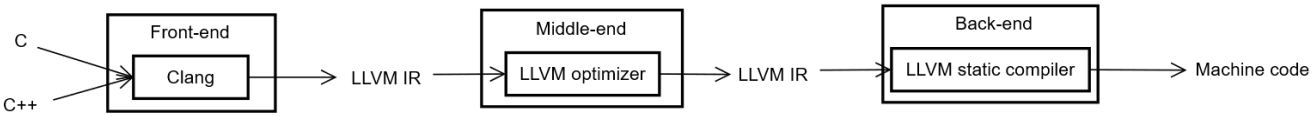

Figure 2 LLVM pipeline

It is possible to use LLVM IR to call methods written in C language from Java code. The project which allows this is called Sulong. Sulong is yet another LLVM IR interpreter. The goal of Sulong [RGW16] is to execute C language on GraalVM. GraalVM is a universal virtual machine that tries to execute different languages on a single runtime. With a Truffle framework implemented on the top of GraalVM, it is possible to run LLVM IR targeted languages. Truffle framework allows running Truffle AST generated by the Sulong project on GraalVM. Figure 3 represents the pipeline of the Sulong project.

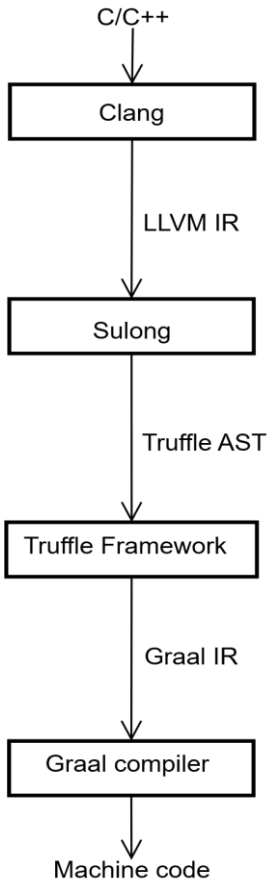

Figure 3 Sulong pipeline (image was adapted from [SEi19])

The main part of the GraalVM stack is the Graal compiler that compiles Java bytecode to machine code. The Truffle framework produces Graal IR that is directly interpreted by JVM. However, GraalVM together with the Sulong project can only run small single-threaded C programs. The part of the Sulong project that is responsible for producing Truffle AST from LLVM IR is TruffleC. TruffleC [GRM*2014] is a combination of a self-optimizing AST interpreter and a Just In Time (JIT) compiler. TruffleC can make optimizations on AST, such as function inlining, changing variables to constants. The problem with Truffle is that languages implemented on top of the Truffle framework use Truffle frames. Truffle frames are arrays that hold values of local variables used in functions. These variables are never allocated on the Java heap. It means that arrays can never be referenced by pointers because they do not have a memory location. This is not a case for C code. TruffleC can not use Truffle frames for variables that are referenced via pointers in the C program. To fulfill C specification TruffleC stores all global or static variables in a memory block of the native heap. It guarantees that variables will not be garbage collected during execution and pointers can be used to reference this data. This heap is accessed using Java Unsafe

API [MPM*15] which is available in OpenJDK. Any kind of pointer arithmetic is available in TruffleC because all addresses, used to access data in the native heap are long (64 bit) values. Another difference between Java and C lies in a stack frame. C can keep arrays, unions and structures in a stack instead of the heap. TruffleC solves this problem by allocating a second memory block in a native heap like a stack. The second block holds all arrays, unions and structures allocated by the C program on the stack. This native heap memory model allows referencing values using pointers. However the Graal compiler makes all optimizations on the Truffle frame. Therefore TruffleC stores local variables that are never referenced by pointers in a Truffle frame array. Another intermediate representation that can bring C library into the Java environment is MIPS [CPS19] binary. The code is compiled into MIPS binary and then translated into Java bytecode. The advantage in such an approach is that it takes the binary representation of the code and does not deal with language. The system that uses described technique is called NestedVM. NestedVM allows taking the binary representation of code as an input. That means NestedVM does not care about parsing and code generation. Also it is freed from language specifications. NestedVM can support any language for which MIPS targeted compiler exists. Avoiding compilation process NestedVM guarantee that code was checked against conditions introduced by the compiler. NestedVM supports all instructions found on MIPS 2000 CPU. There exist many similarities between MIPS ISA and Java Virtual Machine. All MIPS instructions are 32 bits long and well supported by GNU Compiler. NestedVM does not support threading.

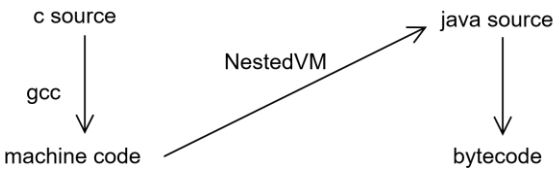

Figure 4 C source to Java source (image was adapted from [BAd09])

NestedVM has two modes to operate on code. The first mode is binary to source code (Figure 4). In binary to source, the C code is compiled into a binary, including any libraries, then the binary is taken by NestedVM that produces Java source code as a result. The result then can be given to the javac compiler which generates Java bytecode.

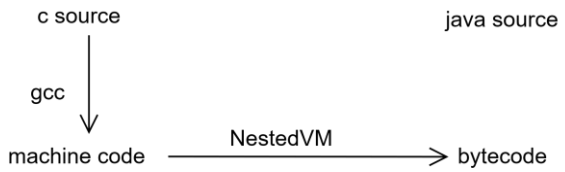

Figure 5 C source to Java bytecode (image was adapted from [BAd09])

The second mode is binary to binary translation (Figure 5). This case translate MIPS binary directly into Java bytecode. Second mode has several advantages over the first one. In this mode we can eliminate use of javac compiler to get bytecode. Another advantage is that MIPS binary can contain sequence of instructions that can not be presented as a source code. Another way to represent C code is to use the clang front end for LLVM but instead of LLVM use clang AST. Clang AST [Kli13] was designed to help in the development of tools for C/C++ languages. For instance clang-check tool is a simple syntax checker which traverses clang AST and checks the syntax of the source code. Clang AST is fully type resolved [Appendix 1]. Clang AST is being used in the Clank [KVo17] project which performs source to source translation of code from C/C++ to Java.

Despite known approaches which use LLVM IR, this project takes clang AST as an intermediate representation. The project takes C/C++ source code and transfers it to clang AST. After that Clank traverses clang AST and generates Java source code. This method gives pure Java code which is as close to the original as possible. This results in the portability of code on all platforms available where JVM can run. As an alternative for clang AST the parse tree of ANTLR4 [Ter13b] can be used.

ANTLR4 is a tool that a developer can use to create a parser/interpreter for a language. It accepts an input grammar that resembles the structure of a language, and emits a parser that can consume that language. ANTLR4 is a compiler tool that takes grammar and generates a parser for it. ANTLR4 is implemented in Java, and you can generate parsers with this tool for various languages: Java, C#, Python, JavaScript, etc. It supports most grammars, and implements advanced features like tree building, error recovery, and support for left-recursive rules. ANTLR4 is widely utilized in such tools/libraries as Eclipse IDE, Hadoop, and Groovy - just to name a few. If you want to parse or interpret language, an option is ANTLR4.

In order to directly generate Java bytecode from C code clang AST and ANTLR4 can be utilized with Java bytecode manipulating libraries. Some of them are: BCEL [DAH99], SERP [WHI07], ASM [BLC02]. BCEL is a Java class which manipulates class files in three steps. The first step is deserialization of the byte array. During that process, all bytecode instructions are converted to objects. The second step is the implementation of modifications. And the last step is to serialize modified objects

into the new byte array. This approach solves slow serialization and deserialization problems. It does not change anything in the byte array but modify deserialized objects.

SERP uses a similar approach as BCEL based on object representation of classes. The SERP also contains classes to manage symbol tables. It keeps the references to the symbol table up to date. The indexes in symbol table are automatically changed if a new constant is added in or removed. However, SERP, like BCEL, does not remove unused constants from the symbol table.

Another option is ASM. ASM is a highly scalable and fast Java library for generating, transforming and analyzing Java bytecode. It includes an API to create and modify Java byte code and an API for analyzing and reading the byte code of Java classes. ASM is small, fast, and simple to operate with; thus, it has become a high-quality representative of class file framework broadly used in the fields of custom class loaders, code generation, and Java program modification.

ASM is used by many popular Java tools and libraries, such as the Spring Framework, Hibernate, and Apache Commons BCEL. The main idea of ASM is not to use object representation of classes. It takes a lot of time and memory to de-serialize the class. The solution made in ASM is to use the same approach as in SERP and BCEL but without explicitly representing tree with objects. The design pattern used in ASM is called visitor. The visitor design pattern can manipulate the code of methods without creating one object per instruction. The approach based on the visitor design pattern is, therefore, more efficient than the approach used in BCEL or SERP. Along with manipulation it allows to create class files. ASM has ClassWriter class which is used to construct classes directly.

3. Translating C into Java Bytecode

This part of the document will outline the compilation strategies which are intended to translate code written in the C programming language into a bytecode format.

3.1. Compilation strategies.

The hierarchy of a library built using a C to Java compiler should not follow anything like that of a C library. In a Java .c and .h files can be treated like a class and interface. But this does not capture the full picture, since C has structures, which are essentially classes themselves, and the compilation strategy for C structures should then emit a separate .class file with a class for each structure. However, because we are going to store these objects in serialized .class files, every field in a structure should be marked as an instance field.

The C programming language requires manual memory management, which entails the explicit allocation and release of memory. Conversely, Java does not mandate explicit memory allocation. JVM automatically calls garbage collector to identify and remove any references that are not being utilized within a project. As a consequence, functions that allocate memory in C can be replaced with object creation in Java bytecode, and functions that deallocate memory can be omitted from the bytecode generation process.

Java does not have the ability to use pointers, so one way to simulate them is with arrays. We could replace each pointer (which is used in C) that is used in Java with an array. A single element array representing a pointer to a single element However, we never want to lose the ability to calculate the pointer arithmetic, so we will use an array to represent pointers. Pointer arithmetic to enumerators works differently. In C, you can enumerate through arrays by incrementing a pointer. But this cannot be done in Java, since we cannot compute references to an array this way. In order to point into a C array from Java, we need to first load an integer variable to the stack, then load reference, and then treat this combination of reference and index as a pointer.

C and Java approach their character data types in very different ways. For example, C needs only one byte for a character and Java uses two. Since C chars are promoted to the Java byte data type we need to fill this gap when generating the bytecode. Automatic variables in C need not be initialized before being used. On the contrary, according to Java standards, all the variables should be initialized before using them. To bridge this gap between the two languages, one solution is to push the null reference onto the stack together with the variable when declaring automatic variables in Java. This fixes the difference between C and Java in terms of automatic variables by ensuring all variables are initialized before they are used.

Java does not natively support C-style pointers, so our strategy is to represent pointers as 64-bit integer values (longs) that encode addresses within the simulated memory regions. In Java, pointers are replaced by indices or handles; for instance, a pointer to a single element can be represented as an index plus a block identifier for the memory segment (stack or heap). It is crucial to retain pointer arithmetic capabilities: in C, one can iterate through an array by incrementing a pointer. The compiler preserves this by interpreting pointer arithmetic as arithmetic on these 64-bit address values. If a pointer p points to an element in the simulated memory, incrementing p by 1 (for, say, an int*) will add the size of int (4 bytes) to the address value. Our bytecode generation uses arithmetic instructions (IADD, LADD, etc.) to accomplish pointer addition. The lack of actual machine addresses in Java means we cannot compute an address to an arbitrary location, but by confining addresses to our simulated memory arrays, pointer arithmetic is made feasible and safe (since it only affects indices within our arrays).

In summary, our compilation strategy for mapping C to Java bytecode involves organizing output classes in a logical manner (e.g., one per struct, one main class for global context), implementing a simulated memory for manual memory management and pointers, and translating C control structures and operations into equivalent bytecode sequences. This strategy underpins both designs of the compiler described in the following sections.

4. Implementation of C Language Features in the Custom Compiler

Using ASM library, our custom C compiler emits Java bytecode targeting the Java Virtual Machine (JVM). Since the JVM doesn't support things like pointer arithmetic and direct memory addressing (which are pretty fundamental to C), the compiler maintains a virtual C memory model, where all memory is allocated through Java byte arrays and 64 bit addresses. In this model, all C data (primitives, arrays, structures, etc.) lives in a managed byte-array heap, and pointer values are just 64-bit integers that encode a memory address (with segments and offsets). This layout is instrumental for the compiler to mimic C low-level memory semantics on the JVM with respect to pointer arithmetic and explicit memory management, and address manipulations that must behave in a way which is consistent with native C. Thesis will describe incomplete feature list (25) of C language for COD CIF parser in next subsections. Work goes through every feature explaining its purpose in C and how the compiler supports it or would support it in terms of simulated memory model, symbol table management, and ASM-based bytecode generation.

4.1. Primitive Types

The compiler understands common primitive C types (char, int, long) and translates them to how these are represented in the JVM. Each base type is given a fixed size within the model so that it uses a consistent amount of space (e.g. char 1 byte, int 4 bytes, long 8 bytes), since those are the common C definitions. By representing the types used in the generated code by Java primitive type (e.g. 32 bits int for C int), the implementation can rely on the JVM arithmetic instructions while using the correct C representation width. Casting normally handled by appropriate conversion bytecode generation or if the data types differ in size by reinterpreting the values; converting from a larger type into a smaller (where the value would be too large to be stored in the smaller type) could truncate the value is semantics similar to that of standard C.

4.2. Structs

C struct types are implemented by creating a corresponding layout in the simulated memory and then, behind the scenes, generating helper constructs that operate over this layout. For example, when we hit the struct type, a Java class will be created to represent the schema of the struct. For every C struct definition, a class is generated with the same name as the struct and its fields defined in the same order as the C struct. The compiler can compute field offsets and struct size (eg, through Java reflection on the generated class) using this class. The size of a struct is the sum of the sizes of its fields and that size

is allocated contiguously from the byte-array heap whenever a struct variable is used.

We compile field access into a sequence of bytecode that calculates the address of the field before reading or writing the value. As an example, if the compiler sees a line of code with a pointer to a struct, it knows that for a given instance of that struct, each field has a constant offset, so the compiler will generate an address calculation base_address + field_offset to read/write the field memory location. It then emits the suitable load or store instruction, either to load the field value from the byte array or to store the value of the field in the byte array. It preserves C's rules for memory layout by treating structs like contiguous memory blocks, using pre-calculated offsets for fields.

The use of an internal Java class for each struct is primarily for bookkeeping purposes (ensuring that fields are ordered correctly and the struct size is calculated properly) and does not incur much runtime overhead. We still store all data in the simulated C heap. This gives us a clean separation between struct type metadata (the generated class) and struct instances (in the byte-array memory), allowing us to easily support sizeof on structs and generate correct code for struct assignments.

4.3. Pointers

Pointers are a central construct of C and the compiler directly maps pointer semantics onto the JVM. Pointer value is a number (long) that mimics an address in simulated memory (64-bit) In practice, the address is divided into a high-order "block" index and a low-order offset, which are packed within the 64-bit value. The block index is used to identify a section of the "simulated heap" (for example, one block may represent the static stack area, while the others will be for heap allocations), and the offset will be the byte position in that block. With this scheme, you can nicely handle pointer arithmetic and access schemes for memory segments. As an example, the null pointer is represented as a special block index (block 0) pointing to an empty byte array at the start of the memory; if a pointer has a block 0, it is interpreted as NULL.

When an expression consists of a pointer with an addition or subtraction with an integer number (such as p + 3), the compiler calculates the numbers by multiplying the integer with the size of the pointer type and add that value too pointer's offset. In a native C environment, pointer increments move the offset by the size of the pointed to elements (i.e. an int* advanced by 1 adds 4 to the offset) and this is preserved in our compiler as well. This arithmetic does not change the block index, so pointer still points to the same memory block, just with a new byte offset. Such calculations are emitted as normal arithmetic bytecode, plus constant multipliers based on type size.

When compiling a pointer dereference (*p), the compiler emits an instruction sequence that uses the block and offset of the pointer to access simulated memory. With a byte-array heap the compiler creates

code to index into it: it loads the global heap array and the pointer's block index selects a byte sub-array, and the offset loads the value from that sub-array. As an example, pointer p might represent value L, a 64-bit address, where the high 32 bits of L represent heap index and low 32 bits offset. Likewise, storing through a pointer (p = value) creates code that writes into the byte array at the address which was calculated. We use appropriate load/store instruction according to the pointee type (byte, int, etc.).

The numeric address value for a char* and an int* is the same so to the specific pointer type that driver provides, notice that the casting between pointer types (from char* to an int*) only changes the interpretation of the pointer but does nothing in our model, it does not change the actual address value. Since our pointers are simply addresses, pointer casts become no-ops in the bytecode (with the exception of re-attaching a type in the semantic analysis). You still have the same long value stored in the pointer; it just determines how to handle subsequent pointer arithmetic, or how to dereference. This closely mimics the way C handles pointers.

In particular, the compiler models pointers as 64-bit addresses and provides primitive operations for all arithmetic and dereference operations, thereby restoring the raw memory access of C on the JVM. This was important considering both the lack of native pointer support in the JVM, and the need to have complex pointer usages (involving multiple levels of indirection and pointer comparisons) act correctly. Pointer comparisons (for example, p == q) are implemented by comparing 64-bit address values, which compares in the same way C does (without attempting to interpret the content, but simply compare the numeric addresses).

4.4. Arrays

C arrays are implemented as a contiguous memory region in the simulated heap. Whenever we define an array (either globally or locally), the compiler allocates a block of memory corresponding to the array total size. So, for instance, a definition int arr[10]; would contain a 40 byte allocation (10 * 4 bytes) in the Java stack, or, if array is treated as a pointer, in emulated C memory section.

For example, accessing an array element (e.g. arr[i]) is done via pointer arithmetic under the hood: the base address of the array (which in C is the address of the first element) is added the index offset i sizeof(element) to get the address of the target element. The compiler will calculate the address and generate code that is either directly loads/stores from this address. In reality, if arr is a known local or global array, then the base address is a constant (in same block), and the code for arr[i] can compute base_offset + i * 4 and index into the byte array. If arr is accessed through a pointer, the base is a run-time pointer value (first 32 bits), and the math is done on that offset value (last 32 bits). This makes array indexing exactly the same as pointer arithmetic according to C semantics.

This has several consequences, one of which is that in expressions arrays decay to pointers which point to their first element. The compiler takes this into consideration, when an array is passed into a function or assigned to a pointer, the compiler converts the array's base address into a pointer that has the type of the array. In reality, the array never gets copied; only the address gets passed around, as a regular C implementation would do with function arguments by pointer Multidimensional arrays are not used in COD CIF, hence our compiler does not generate bytecode for them.

4.5. Function Pointers

In a JVM-hosted implementation, function pointers in C are both a difficult and special case to handle: the JVM does not support jumping to arbitrary code addresses. This problem is solved by the compiler by abstracting function addresses as indices that can be used to indirect a call. To be precise, each C function definition is given an identifier, and a function pointer internally a long value that stores that identifier (implementations might encode it in the same scheme that stores the block-index/offset of the pointer, or might hold it in an entirely separate symbol table to ensure that it will not conflict with a value that points to data).

Instead of a block index pointing at some memory, a function pointer may have a specially chosen block or encoding to indicate "code". The compiler, for example, could mark out a specific block index or range of addresses to be a type of function pointer, and then have the offset or lower bits encode the function ID. If code takes address of function itself (i.e. have something like int (*fp)(int,int) = add;), the compiler generates a constant long. This is simply saved like a normal pointer. Calling a function through a pointer (for example: (*fp)(a,b)) is an indirect call in the bytecode. Since Java bytecode is unable to invoke a method via a dynamic numeric address the compiler uses an indirect dispatch.

That is certainly some overhead, since this is a linear or table lookup, but for a thesis-level implementation, this is an appropriate amount of overhead for explaining correctness. This means that indirect function pointer calls in the program are resolved to calls to known (by the compiler) function bodies, ensuring safety (the compiler will also ensure type safety of the signature). The context (like local variables, call stack, etc.) for the call is handled by the normal function call mechanism (see the function calls/recursion section below), so the only thing special about a function pointer call is this dispatching logic. This respects C semantics: the same pointer value is always operated on, meaning that if you pass the same pointer value around, it will always end up calling the same function; additionally, comparisons of function pointers (like whether two function pointers are equal) compare on their identifiers as well.

In summary, function pointers are supported by mapping code addresses to identifiers and performing an explicit dispatch. While this differs from a native implementation (where a function

pointer might literally be a CPU address to jump to), it provides the equivalent behavior on the JVM.

4.6. Global Variables

Global (file-scope) variables must have static lifetime and must be accessible from any function. Our compiler elected to store its global variables in a special part of simulated memory, with compile-time fixed addresses. The simplest approach is just to allocate a chunk of memory for global data at program startup, either from the heap[1] byte array (the “stack” segment) or from some other memory block for all globals. Each global variable is assigned a constant offset from that region. So, if the first global is an int, it may be at offset 0 of the global block, the next global (say a struct) at offset 4, etc., properly spaced according to the size of each variable. These global offsets are tracked by the symbol table maintained by the compiler.

The compiler emits code to access the static address of that global when it generates code that uses it. This is usually done by loading the base pointer of the global segment (which may be the base pointer of some known block index or a stored base pointer) and the constant offset afterwards. To load a global int G, for example, the code might push the global block index and G's offset, and then do an array load from the simulated heap. With the address fixed, the compiler is able to hard-code those values into the bytecode, making access to globals efficient (with no dynamic calculation except for array indexing).

Global variables may equally be initialized with unchanging values or memory locations. If an early value is provided (e.g., int x = 5; at global scope), the compiler will generate code in the program's initialization (for instance, the class initializer approach in bytecode) to store that value into the suitable memory location before any user code executes. For pointer initializers that refer to other globals (e.g., int *p = &x; globally), the compiler computes the address of x and stores it in p's memory slot during initialization. Deriving the address of a global (&globalVar) is straightforward since the global’s address is constant: the compiler will yield the corresponding long pointer value (comprising the global block ID and the offset of that variable) as a constant. This permits pointers to global variables to behave similarly to pointers to any memory location.

By handling globals in this manner, the compiler ensures they behave as "static" storage with a single allocated location throughout the program's execution. All functions reads and writes reference exactly this slot, replicating C's design of global variables persisting at a fixed memory location during the program. The simulated memory technique also implies that globals can be treated similarly to other memory - specifically, identical byte-array access routines function for globals as well as locals or heap data, streamlining the runtime system. Additionally, the runtime must ensure thread-safety for globally shared memory, as concurrent reads or writes by multiple threads could potentially cause race conditions.

To address this, the compiler implicitly protects globals with locks to avoid unintended interference between threads.

Memory segmentation note: We effectively maintain two logical segments in the simulated memory: one for global/static data, and one for the stack (automatic variables). In our CMemoryImplementation, for instance, heap[1] serves as a regular heap. We avoid interference from global data with the runtime call stack by having separate subregions for each, or by having separate offset management for each. For example, there is no reason why the compiler shouldn't simply place the stack at heap[1], and place globals above it. Thus as the stack grows (due to recursion or for local use) it won't clobber the global variables. Such details ensure the integrity of global data throughout program execution.

4.7. If Statements

If statements are implemented using conditional branch instructions in the bytecode. The compiler evaluates the condition expression and an integer result (in C, all non-zero integers are "true" and zero is "false"). This result is usually int left on the stack so 0 - false and 1 – true.

The compiler must implement the if, so it creates a label for the else (or end section). Without an else, for a simple if (cond) {... } the pattern for generating bytecode is: evaluate cond, and then emit a conditional jump instruction that skips the true-block if the condition is false. After computing cond, for instance, the compiler would then emit IFEQ L_else (jump to label L_else if the condition equals 0), which would allow us NOT to execute the true branch at all if cond == 0. As the next step the compiler emits the true branch code which ends with the L_else label (jump target) IF-ElsE structure uses two labels: One at the beginning of the else-block and another one at the end of the whole if-else structure. When false, the compiler emits IFEQ L_else to jump to the else part, followed by an unconditional jump at the end of the true-part to go to the end label (skipping the else part). Finally, it emits the else-block and the end label.

The above form ensures only one of the true/else blocks is executed, just like in the high-level semantics. By using ASM's branch opcodes (IFEQ, IFNE, IF_ICMPEQ, etc.) the compiler can handle all forms of conditions, including relational expressions. For instance, a condition a < b would be evaluated by a relational operation that leaves a 0/1 on the stack (or alternatively, the compiler might directly use a comparison jump: e.g., IF_ICMPGE L_else to jump if a >= b which means the if (a<b) condition failed).

Both are in fact approachable. The bytecode uses goto labels to fork the control flow when it needs to. The resulting pattern also maintains short-circuit logic. For instance, if inside the conditional

expression it sees && or ||, it treats those separately but still correctly does lazy evaluation. On a high level the if statement compilation boils down to a mapping to conditional jumps in the bytecode, where labels denote the beginnings of the then and else blocks.

4.8. While Loops

While loops are implemented by the compiler using a combination of labels and conditional jumps, forming a loop structure in the bytecode. A while (condition) { body } is translated as:

- A label at the beginning of the loop (call it L_top), it marks the start of each iteration.
- Bytecode to evaluate the loop condition.
- A conditional jump out of the loop if the condition is false. For example, the compiler might emit IFEQ L_exit (jump to exit label if condition is 0/false), so the loop terminates.
- The loop body code is emitted next.
- An unconditional jump back to L_top is emitted at the end of the body, causing the program to loop back and re-evaluate the condition for the next iteration.
- A label L_exit is placed marking the point where control lands when the loop finishes (i.e., the condition was false).

With ASM this is done by creating Label objects to L_top and L_exit. The condition code is produced at L_top, and the jump by an ifeq or ifne instruction to L_exit, depending on how the condition is evaluated. When done, we GOTO L_top (unconditional jump) to jump to the top of the body of the loop. This is like how one would manually write a loop in assembly or pseudo-code. As a result, the body repeats as long as the condition is true.

The compiled while structure supports break and continue (described below) naturally—L_exit and L_top are both targets for break and continue, respectively. If a break emerges within the loop body the compiler emits a jump to L_exit while in the event of a continue it emits a jump to L_top (actually the continue in while loop jumps to reevaluate the condition is true). A little context-keeping in the compiler during code emission for a loop body lets you perform these translations. One can check the while-loop translation correctness in that — if the condition is false the first time checked, the body is skipped altogether (as the jump goes directly to exit), and if true, the body executes and finally jumps back to check again — exactly as in the source logic.

4.9. For Loops

For loops (for(init; cond; iter) { body }) are compiled in a manner similar to while loops, with some extra handling for the initialization and iteration expressions. The compiler breaks a for-loop into its components:

- The initialization section (init) runs one time, before the loop begins. The compiler emits code for init just like it would for any other statement (a declaration or an assignment expression), and that code gets emitted immediately right before the loop start label.
- The test cond is evaluated before every iteration, just like the condition of a while-loop. After the initialisation code, one has a label L_top, at the point where the condition will be checked on each iteration. It essentially compiles the condition to a boolean (0/false or non-zero/true) and emits a conditional jump (e.g. IFEQ L_exit) to quit the jump.
- Next, again, just as in while loop, the body of the loop is generated.
- The third step is executing the iteration expression (iter) after the body. If we take a look at the code emitted by the compiler, the iter step comes after the body. This generally increments a loop index or does some other forms of update. Important: After the iteration code has run, the compiler subsequently emits the GOTO L_top unconditional jump to re-test the condition for the next loop iteration.
- Similarly as while, L_exit label to the end of the loop, where control jumps to when the loop is over.

This ordering of the pieces (init, L_top, cond check/jump to exit, body, iter, goto L_top, L_exit) allows the generated bytecode to properly implement the for-loop semantics. The initialization phase runs once, the condition controls the entry to the body each time, and the step runs after every execution of the body. For instance, a for-loop like for(int i=0; i<10; i++){... } would compile something like: code to set i=0; label L_top; compare i < 10, if false jump to L_exit; loop body...; after body, increment i++; jump back to L_top; label L_exit The order of execution is exactly what we expect it to be.

From jump labels perspective, we re-use the same jumping label pattern as in while (one for top and one for exit) However, the existence of the iteration step means there is some extra emitted code between the body and the jump-back. The continue statement in a for-loop will go to the iteration step code (not L_top directly). Continue will skip the rest of the body, but it will only skip the part before incrementing and checking the next condition.

The compiler sees a continue in a for-loop context, it emits a jump to a label located right before the iteration code. What actually happens is that the compiler would internally generate some label for the

“continue target” (inside the iter code) which may differ from L_top, or L_top could be reused that requires the ordering of the code to be right. For instance: L_top for condition, L_continue for starting of iter code, L_exit for end. Then ends the body (or a continue) branches to L_continue, at L_continue the iter runs then a GOTO L_top. This will guarantee that the next iteration starts where it left. In summary, the for-loop compilation is the same correctness argument as while-loop compilation, but adding in reasoning about the initialization and increment parts in the correct locations.

4.10. Break and Continue

Break and Continue statements change the regular flow in loops; compiler implements them by using a direct jump to the desired loop label. When generating code, the compiler remembers the labels of the current loop: the label to go to for a break (the exit) and the label to go to for a continue. These labels are finalized upon entry to a loop: for a while-loop the continue target is the condition-check label (or loop top) and for a for-loop the continue target is the iteration-step label (as mentioned previously), while the exit-label is after the loop body in both cases.

The compiler emits an unconditional jump (ASM opcode GOTO) to the exit label of the loop when it encounters a break statement. This will cause the execution to skip the rest of the loop body and go to the end of the loop, thus ending the loop prematurely. Since each loop in such nested structures will have its own exit label, a break without argument breaks out of the innermost loop in which the break occurs.

In bytecode, jumping out with GOTO is straightforward since all loops in one function compile into a single method’s code – the only work is to have the correct label. In the case of a continue statement, the compiler generates a GOTO to the continue target label of the loop. In the case of a while-loop, this target will often be the top of the loop (the condition that gets run). So a continue in a while makes a jump to re-check the condition at the top, and it forgoes code underneath that continue in the body. As mentioned, the continue target in a for-loop is typically located just before the code for the iteration expression. This means a continue will jump to that label, performing the iteration step (which may for instance simply increment the loop index) and proceeding into the next iteration's condition check. This corresponds with the high-level behavior of continue in a for-loop.

Correctness of break/continue handling is merely a function of the compiler emitting the correct target labels. The semantic phase validates that these statements only appear inside of loop constructs (the parser would error break/continue outside loop). As for verification, the JVM bytecode verifier has no issue with the forward jumps (it's fine because break/continue are forward in the sense that, in the instruction stream, they jump out of or to the end of the loop construct), as long as destinations of jumps

are the same method (that's the case here). The compiled code suppresses any unintended instructions from executing by enforcing structured, predictable control flow. This approach keeps the loop control flow logically separate from the normal sequential flow in the bytecode.

4.11. Labels (Goto)

C language allows arbitrary labeled locations in a function and goto statement to jump to them. We implement labels and goto using the low-level branch instruction within bytecode. Each label in the C source (myLabel:) gets transposed to a new ASM Label in the generated method. In C, a label is nothing more than a position in the instruction stream to which we can goto. The visitLabel method is called in the middle of the code generation process, when the definition of a label is encountered by the compiler, to put the corresponding label in the current bytecode position. It itself does not generate any code, it is just a marker.

For every goto label; statement, the compiler generates an unconditional jump (GOTO), and it targets the ASM Label of the specific target label. Such gotos translate into intra-method jumps (we are in the same function/method context, C does not jump into a different function so those gotos have to translate into what the JVM supports). Forward or backward gotos in the source is the challenge that is being taken care by the compiler. Forward jumps are treated by ASM as placing a placeholder which is backpatched when the position of the target label is known, allowing the compiler to emit a GOTO even if the label is defined later in the source. By the end of code generation for a function, all gotos and labels have been resolved.

The first thing to consider, is that the use of goto cannot break any JVM verification rules (e.g. stack height must be consistent). We treat gotos as just a control flow change, which does neither push nor pop from the dataspace stack, so the stack depth at a label is always the same, no matter how it is reached (the compiler outputs structures and semantic checks enable this). As a result, we never run into things like jumping to the middle of a construct in an unbalanced stack.

In short, the label/goto structure is implemented almost directly: every C label is a bytecode label and every goto is a jump. This shows that even “unstructured” control in C can be expressed in Java bytecode. The resulting behavior is semantically identical to C: a goto statement jumps to the labeled statement, with no extra runtime cost except for the jump itself. ASM allowed labels, and the JVM branching mechanism itself can easily jump forward or backward to an arbitrary instruction within the same method.

4.12. Increment and Decrement

For unary increment (++) and decrement (--) forms, with both prefix and postfix variations, we emit load, modify, and store instructions, but this time they are directed towards the variable or memory location of the variable. The compiler will first decide if the operation is on an lvalue (a variable, array element, or dereferenced pointer, for instance) or a rvalue (basically, a simple value) in the context of the expression.

For an lvalue such as a variable x, a statement x++ or x-- is implemented like so: read the value of x, add or subtract one, and write to x's location. If the operator is prefix (e.g. ++x), the new incremented value is used (the return value of the expression); if it is postfix (x++), the old value (before increment) is the result of the expression and the side effect of the increment takes place.

For example, let's say we need to implement i++. The compiler will output a bytecode sequence: ILOAD i (push current value of local i), then the literal 1, then IADD to add one, then ISTORE i as a write back. But this sequence as written expresses prefix (because at the end of it, the incremented value is in memory and on the stack). For postfix, it needs to keep the original value for the result of the expression. In one way: ILOAD i (the original), ILOAD i (the original), ICONST_1, IADD (the incremented second copy), ISTORE i (store new value), and return the original (the first copy) on stack as the result. In practice the compiler uses an optimized sequence (DUP bytecode to duplicate the value on stack before incrementing) to avoid the need for multiple loads. For prefix (++i), it could either do the increment first and duplicate if necessary or just ensure that the value on stack is the new one.

The Compiler takes care of pointer logic when increment/decrement done via pointer, For Example: (*p)++ or even p++ (where p is pointer variable). This means to increment the value of the address that p points to ((*p)++). This will cause the compiler to load the pointer p, calculate the address based on its segment and offset, load the value at that address, specified for read, increment it, and write it back to the same address, specified for write. However, the outcome of the expression (for postfix) is the original value which was at (*p). These steps are more complex, but it still traces the same load-modify-store pattern, with some address calculation. The way in which p itself will be compiled when it is also a pointer being incremented (p++) is that the compiler will add the stride (according to the pointee type) to the value of p as described in the pointers section.

4.13. Assignment Operators

Assignment is straightforward: the compiler emits code to evaluate the right-hand side expression, and then to write the result into the left-hand side location. Whatever is on the left-hand side(LHS), it can be a variable, an array element, a struct field, or a dereferenced pointer, in any case, the compiler

can find out the address of LHS. After addressing (or target variable index) is resolved, and the value on the RHS is on the stack, the store instruction is dispatched. In the case of something like a = b + c;, the compiler will compile the code that computes b+c (pushing the result on the stack) then the appropriate ISTORE for the slot corresponding to a (if a is a local int) or a putfield/putstatic/array store if a is a struct field, global, or another pointed location in memory. If the target is a pointer like *p = value; the pointer p is loaded, the address is calculated, the value is calculated, and then a BASTORE/IASTORE (for the type) is emitted to store into the byte array at that address.

In addition to the simple = assignment, the compiler understands compound assignment operators such as +=, -=, *=, /=, &=, |=, <<=, and so forth. That is done by a read-modify-write sequence on them. This means that when you write an expression such as x += 5; it is essentially equivalent to x = x + 5; in terms of effect. The compiler will make loading of x, then constant 5, then add, then store sum back to x, it can do this directly or use combined operation if such exists in the java bytecode level — which for Java bytecode itself does not have a single instruction for compound assignments, thus it was generally the same explicit sequence. Likewise, ptr -= 1; where ptr is a pointer, will lay out as ptr = ptr - 1; (after scaling 1 by pointee size as explained for pointers). The important part is that the compiler will identify the compound operator and will expand it to the underlying arithmetic or bitwise operation followed by the assignment.

One detail is evaluation order and side effects. For instance, in the case of a += f() the function f() has to obviously be called first (to produce the value), and only afterwards added to a. Our compiler proves to preserve the sequence by placing the RHS code generation before the LHS load + operation. We also must handle cases where the LHS might be an expression with side effects (though in standard C, the LHS of += is evaluated only once; we ensure to adhere to that by not double-loading it unnecessarily). For example, arr[i++] *= 2; will evaluate i++, get the old index, then fetch arr[old_index], double it, and store it back, and finally the side effect of i++ is completed. The compiler carefully orchestrates these to ensure compliance with C's semantics.

To summarize, assignment operators generate simple bytecode: evaluate RHS, get LHS address (if necessary), and write. Per addition, compound assignments simply add in an additional read of the original value alongside the operation. The result of an assignment expression (in C) is the assigned value (in the case of simple =, the RHS; in the case of compound (op)=, the new value after the operation). When an assignment is used as part of an expression, our generated code usually leaves that value on the stack as well (except for a statement context where it would not be used anyway).

4.14. Arithmetic Operators

Compiler supports all the basic arithmetic operators (+, -, *, /, %) and maps them to the corresponding arithmetic bytecode instructions. For arithmetic expressions, the compiler will evaluate all the operands, and then emit the operation. As an example when considering the case of x + y (x and y being integers), it will output code which will push a value for x (ILOAD or the equivalent), push a value for y, and emit the IADD instruction that sums the two, and leaves the sum on the stack. If fallbacks were longs it would then call LLOAD and LADD, and so forth. For Multiplication USE IMUL/LMUL, For Subtraction USE ISUB/LSUB and so on. Division and modulus (/, %) are a little special because bytecode-wise in Java they map to IDIV/IREM (and format versions, too).

The semantics of these are just like C's integer division for both positives and negatives (the exception being division by zero, where in Java this will throw an exception – which is effectively like the undefined behavior in C; our compiler doesn't check for divide-by-zero as it's believed the program is valid or giving it up to the JVM to throw an ArithmeticException, which would be like the equivalent of a C runtime error like crash). The compiler applies usual binary promotions and cast rules. As an example, C will elevate values of types char or short that are being used in arithmetic to type int prior to the operation. However, the generated code will load them as each as an int (we issue conversion instructions explicitly). So, the effect is that appropriate arithmetic occurs: 32-bit or 64-bit, depending on matching C rules (two int gives an int result, an int and a long gives a long result after promoting the int our compiler tracks this in the type symbol table, and emits the instruction combination like I2L to perform the conversion, if necessary). If the arithmetic expression is complex (e.g. a + b * c), then by the structured traversal of the abstract syntax tree the compiler will obey operator precedence and associativity. It will output the code for b and c first pushes that on the stack, it does the same for a then adds the top two stack entries, and you will get a + b * c, as expected in C.

A note about this is that a compiler does not use any high-level optimizations of arithmetic; it produces relatively straight translations. In a normal argument, this would look like: x * 2 (the JIT could optimize this too, but at the bytecode level, it's okay to just multiply x by 2). We never use efficient loading of constants (for instance, we never use ICONST_M1 to ICONST_5 for small constants, BIPUSH/SIPUSH for others, we always use LDC for all constants). This means the arithmetic operations in the output are not done as normal Java — they are not efficient.

In summary, each arithmetic operator in C corresponds to one or a small sequence of Java bytecode instructions that perform the equivalent calculation. The simulated memory model isn’t directly involved here (except if an operand is fetched from memory or a result stored to memory).

4.15. Bitwise Operators

Bitwise bytecodes of the JVM are called for bitwise operation (&, |, ^, ~, and the shift operators). Bitwise operations in C are on integers (integral promotions apply as necessary), and so our compiler is going to do exactly the same bitwise operations, operating on 32 bit int or 64 bit long as appropriate. The compilation, for bitwise AND, OR, XOR (the binary operators &, |, ^), is simple: the two operands are evaluated, and the appropriate instruction is emitted (IAND, IOR, IXOR for int, or the L... versions for long). For example, IAND which stands for u & v where u and v are ints It leaves its result as an int on the stack.

Such operations do not interact with the memory model besides loading the operands and ensuring to store the result (if necessary). Since Java byte code does not have a single instruction for bitwise NOT thus is a unary operator and implemented using a sequence. This causes the compiler to emit code to load the operand, and then to use an immediate value and XOR to flip the bits. If x is int (because x ^ -1 yields the bitwise complement x, x usually consists of ICONST_M1 (i.e. −1, all bits are 1 in two's complement) followed by IXOR. If x is long, then use LCONST_M1 and LXOR in a similar way. This is the same as doing a NOT. On the other hand, -1-x could be used by the compiler, but XOR with -1 is much more directly indicating that it is a bitwise operation. This is done respecting the priority of promotion (if needed — e.g., if x is a char to start with, it will be promoted to int first so that the operator ~ applies to this int). C shift operators (<<, >>) map to Java shift bytecodes (ISHL, ISHR for int; LSHL, LSHR for long). The compiler just leaves the result on the stack like any other arithmetic result. For example, in a context like if ((x & mask) == 0) ..., it will produce an IAND, then compare the result to 0 with an IFEQ jump. The compiler's parsing/AST ensures the correct order of evaluation; the code generation follows that order.

4.16. Logical Operators

C's logical AND (&&) and logical OR (||) are short-circuiting operators, which means the second operand is evaluated only if necessary. The compiler must emit code that respects this short-circuit behavior. We implement && and || using conditional jumps in a pattern similar to if-statements. For a logical AND expression expr1 && expr2: the compiler evaluates expr1 first. After computing the truth value of expr1, it emits a check: if expr1 is false (0), then the whole && is false and we can skip evaluating expr2. So the code: evaluate expr1, then IFEQ L_false (if zero, jump to false-case). If we reach this point, expr1 was non-zero, so then it proceeds to evaluate expr2. After that, it needs to combine the results.

The C short-circuiting operators logical AND (&&) and logical OR (||) — the second operand is

evaluated only when needed. The emitted code by the compiler must still respect this short-circuit behavior. Thus, we implement && and || as conditional jumps in-if-like fashion. For a logical AND expr1 && expr2: compiler first check the expr1, this emits check that after computing the truth value of expr1() , if expr1 are false (0) then whole && is false and expr2 evaluation can be skipped. The code then evaluate the first expression, then IFEQ L_false (zero, jump to false-case) Since expr1 was non-zero, if we got here then it evalutes expr2. From there it will need to aggregate those results.

A simple approach is to return a boolean (0/1) for the entire && expression. This we can do by using labels for the result, in this case, label L_false for the false case, label L_end for the end. The compiler could set a register or stack value to 1 (true) and jump to L_end if both are true; else, set 0 instead. One more direct alternative is: evaluate expr1; if expr1 is false goto L_false; else evaluate expr2; if expr2 is false goto L_false; else. Both are true push 1 and goto L_end; At L_false, push 0. Combining at L_end, the result is on the stack. That is the normal loop translation.

For logical OR expr1 || expr2, the compiler applies a similar pattern, but with polarity reversed: if expr1 is true (non-zero), then the entire OR is true, and we skip evaluating expr2. In other words: L_true : evaluate expr1, IFNE L_true (true if non-zero, short-circuit), evaluate expr2, IFNE L_true for the second, and if we fall through both being false that is false. The output is generated in the form of int value 0/1, which can be utilized in any subsequent arithmetic or logical operations (C also takes the result of && and || as an int value 0 or 1). The output is normalizing the result by the compiler below ICONST_0/1. But since anything other than 0 is considered true in C, we need to normalize (if expr2 returns 5 and expr1 returns 7, we need && to return 1 to indicate true, not 5 or 7).

The logical NOT operator ! is a simpler case: !expr is true if expr is false, and vice versa. The compiler can implement ! by evaluating the expression and then comparing to zero, yielding 0 or 1. One common bytecode idiom is: evaluate expr, IFEQ L_true (if zero, then !expr is true so result 1), otherwise (expr was non-zero) result is false.

This yields a 0/1 as needed. Alternatively, if expr is known to be 0/1 already, the compiler might do a simple XOR with 1 to flip it. In general, we ensure correctness applying branching logic. This results in bytecode are instructions that makes use of a set of conditional jumps and constants for logical ops, but otherwise, no actual logical instruction (because there is no explicit && or || instructions in Java, it expects such logic to go away at compile-time). This approach resembles how a C compiler would emit assemblies for these operators (jumping around for short-circuit). The logic for the rest of the expression integrates pretty much seamlessly: for example, in if (p && p > 0), the compiler will short circuit: if p is NULL (0) then it won't perform p > 0, and thus we never dereference a null – this is exactly why the semantics of C are defined that way, and our implementation is constructed to maintain this safety by

construction of the flow of the code.

4.17. Relational Operators

Relational operators (==, !=, <, <=, >, >=) get compiled into comparison instructions in the form of conditional checks that enable branching or create boolean values. In many cases, the if or logical context for a relational check will cause the compiler to simply generate a conditional jump. For instance, an if a > b then jump to else; that means the > condition failed. Similarly, == and! = IF_ICMPEQ or IF_ICMPNE for integer comparisons This prevents a 0/1 result from being materialized in the first place if the next operation is a jump.

However, if the relational expression's value is needed (for instance, int c = (a<b); or as part of a more complex arithmetic expression), the compiler will generate code to compute a 0-or-1 result. In such cases, a common strategy is: evaluate the operands, perform the comparison with a branch, and use a label technique similar to logical ops to produce a constant 1 or 0. After this, there will be an int (0 or 1) on the stack representing the truth of a<b. The compiler uses the appropriate conditional bytecode: for signed less-than we have IF_ICMPLT (branch if a<b), for greater-than IF_ICMPGT, and their negations.

For equality, IF_ICMPEQ and IF_ICMPNE are used. If the operands are 64-bit longs, Java bytecode doesn't have direct long comparisons; the compiler must either invoke helper runtime or do the compare by subtracting and checking or using the LCMP instruction followed by checking its result. The typical bytecode pattern for long compare is: use LCMP (which pushes -1,0,1 depending on <,=,>), then follow it with conditional branches on the result. Our compiler handles longs by inserting an LCMP and then an IF condition on the result.

Pointer comparisons are performed too, but since pointers are long values, we treat them the same as any numerical 64-bit compares. The equality of two pointers (p == q) is nothing but a numeric equality test on the 64-bit address. This can be achieved by doing a 'high vs low' parts comparison or a long compare (LCMP) in one invocation. Likewise, comparisons on pointers (e.g. p < q) are only technically defined for pointers to the same array/object in C (and even then, the result is meaningful only with respect to addresses). In this implementation, we'll just compare the 64-bit values, because we would interpret addresses as being in a flat address space. This is a fair approximation of C behaviour (as long as the pointers both fit into the same block).

As for these comparisons — the compiler makes sure of type conversions (for instance: it promotes shorter type to int in advance as well). Also, for unsigned comparisons the compiler would need to emulate an unsigned compare because Java only has signed comparisons. Most of the time this is done by changing value. Unsigned is not supported in our project, but signed comparisons are.

To summarize, relational operators are either processed by branching logic that directly affects control flow (e.g. in statements, not-expressions) or they return a boolean value. The ability to implement high-level comparisons accurately in the bytecode is realized through the use of ASM labels and conditional jump instructions such as IF_ICMPxx. A comparison always return value 0 or 1 (in C it is defined to be 0 or 1, but it is not arbitrary non-zero) The requirement is satisfied by the way our compiler explicitly pushes 0/1 constants in the flow.

4.18. Ternary Operator (?:)

The ternary conditional operator cond ? expr1 : expr2 is essentially an if-else in an expression context, and the compiler implements it in that way. If compiler encountering a ternary, it will generate code to evaluate the condition cond. Then it will use two labels: one for the "else" part (expr2) and one for the end (after the whole expression). It emits a conditional jump based on cond – if the condition is false (zero), jump to the label for expr2, otherwise continue straight into the code for expr1. After producing the value of expr1, the compiler jumps unconditionally to the end label, skipping the expr2 part. Then it generates the code for expr2 at the else label, and after that code, it places the end label.

The main difference from an if/else statement is that ternary evaluates to a value in an expression context. L_end is the right place to have exactly one value on the stack, because our bytecode sequence naturally finishes with this stack behavior. We need to be sure that both branches each return exactly one output of the same type. At L_end it is verified by the ASM verifier that the stacks heights and types are consistent. By design, we satisfy this: both expr1 and expr2 code paths each push one int (or one ref or one long, etc but they'll be same type due to C type rules applying to). In C, this means both alternative clauses are type-checked against each other—if one is a void, that's an error (ternary requires a non-void type unless both are void and used in void context). At runtime level, the ternary is no more than a small variant of the branching we already do for the if-else case, so it does not introduce new challenges.

4.19. Return Statements

In C, a return statement returns a value from a function or (for void functions) exits the function. The return statement indicates to the compiler that the code does not flow past this point, so the compiler simply emits the correct JVM return instruction at this point. If there is a return statement in the bytecode: if we consider the control-flow graph of the function, and arrive at a return, that path will be finished. For instance, in a function returning int, we would compile a return expr; as: calculate expr (so the return value ends up on the stack) and follow it by the IRETURN bytecode that returns that int from the method.

If the function returns a long or a pointer (which we represent as a long), the compiler uses

LRETURN. For void functions, a return; with no value becomes a simple RETURN instruction (void return in bytecode). Because the compiler uses Java methods to represent C functions, a return in C maps cleanly to a return from the Java method. The JVM will handle popping the call stack and returning control to the caller.

The one thing the compiler needs to deal with is some sort of cleanup that may need to happen before a return. In C, local variables usually only get popped as the stack frame is destructed; in our implementation, if we had allocated resources or needed to adjust the "machine" stack pointer, we would do it before the return. To take one example, if we were simulating locals with a stack pointer that we house in the simulated memory, then this represents a pointer that would need to be deallocated on return; that is, the stack pointer would need to be reset (or if the function gets called again, would need to be correctly set again in advance). But since our design is built on top of the Java call stack for function calls, and we might not reuse the same simulated stack segment for multiple activations at the same time, the main cleanup done may be just freeing local dynamic allocations, if those were performed automatically (and they are not — malloc/free are manual). So in the simple implementation, we don't clean up anything explicitly; the return just leaves the method. Returning from a function was originally the top level function or one of the entry points would be returning which means return to the shell (in C return from main ends the programm, and in our case return from generated main would mean end of that (own) thread of execution).

One detail is that for functions that have local variables in the simulated memory, those are not accessible (after return) anymore. In the case that we moved the local variable into the global byte array stack segment we don't explicitly wipe it, but as they are now out of scope anything in C trying to access them would be undefined (e.g., dangling pointers) — in this case the code did not call free, since there was no allocation. But we can simply leave the bytes as they are, it does not really matter. In C, dereferencing it might be undefined if a pointer to a local escaped, and we may still point to back to that part of heap[1] which may be reallocated by future calls. We're not actively protecting against that, but well defined programs still work correctly.

If the function has a struct or large object to return (like return someStruct; by value), the compiler would need to handle copying that struct into the caller's space or as a hidden pointer. However, as an implementation detail, returning a compound type would involve allocating space and copying, something that could be done with our memory model if needed.

To wrap up, every return in our C source eventually becomes a return instruction in bytecode, and the compiler makes sure that any possible return value is on the stack at that point. The use of the JVM's return opcodes integrates seamlessly: when a return executes, the JVM pops the frame and continues

execution in the caller method (it corresponds to the C caller function). In this manner, the C behaviour of function call / return is precisely reproduced in the target bytecode.

4.20. Function Definitions and Recursion

The compiler supports function definitions (including main and all user defined functions) and resolves each C function to a static method in the output Java class. A function definition gives the name, return type and parameter list; this helps compiler to create a method with the given descriptor (ASM generates method signature in JVM format). An example might be a C function int add(int a, int b) would become a Java static method add(II)I where the add indicates the name of the method, and (II)I indicates the args and return type.

Inside this method, the body of the function is compiled to byte code using the statements and expressions from earlier in this section. Function call activation records (stack frames) are partially taken care of by the Java call stack and partially by our memory abstraction as necessary. If one function calls another, then the compiler creates an INVOKESTATIC call to the callee's method. This results in JVM giving a new frame for the callee that has local variables of its own. If we're trying to slot C locals into Java locals in the most efficient way possible, recursion is something the JVM will take care of automatically – every new activation of a recursive method gets a completely new set of Java locals anyway. If we keep locals on the stack of the simulated memory, then we'll need to keep track of a stack pointer. For our implementation, instead, we used a mixed approach: small scalar locals could be stored directly in Java locals, but, whatever time we needed to take an address to a local, we have to move the variable to the simulated segment of the stack. On the even more concrete stack-pointer model, as is the case for CBytecodeGeneratorWithHeap, the compiler does essentially the following: upon entrance it uses some global stack pointer from heap[1], allocating count as space for the local variables by decrementing it and giving offset as an lvalue. For recursion, this means that each call's locals use their own slice of the heap[1] array, subject to the instruction pointer. Then the return sequence (or the caller when returning) can adjust the stack pointer back, and the freed local space can be reused. This is the same as a real CPU's stack when it comes to recursion.

By implementing recursive calls in our compiler, we were able to test for functions calling themselves (directly or via some other function), to ensure they generate accurate values. If using the Java call stack, recursion is simple: the JVM just creates new frames. If we're on a manual stack, we have to make sure to allocate a new frame on every call. In reality, we relied heavily on the Java call stack, which greatly simplifies recursion – a recursive call is simply another method call. The local variables at different recursion depths are handled separately by JVM, the JVM handles that frame once

a call returns. Provided we haven't put recursive data into what was meant to be static memory.

We also handle parameter passing according to C's call-by-value semantics. When generating a call, the compiler evaluates each argument expression and pushes it on the stack (or if the argument is a struct, allocate and copy it appropriately). The INVOKESTATIC then consumes those values, and they appear as the callee's parameters in that callee's method stack (ASM sets up the method to receive them). If a parameter is of pointer type, it's a 64-bit long value passed along.

Our pointers and values are already in a format that is compatible with Java (i.e., int, long, etc.), so we don't have to do any special treatment beyond normal expression evaluation. It also compiles the definition of main as a main entry point method into a program. If necessary it could emit a public static void main(string[] args) as true main, which would just call translated main (which possibly have a call signature to look like C's int main(int,char**)).

Scopewise and in terms of lifetimes, when a function returns, particularly a recursive one, any local variables that were on the simulated stack are no longer in scope. If we're looking at the stack model for the pointer and need to clear the variables, we either drop them by the pointer being moved back and re-used or just let them be written over by successive calls. However, we have to be careful with pointers to locals: if we had pointed to a local variable of a function that has since returned, what we would have there would be called a "dangling pointer". Our runtime doesn't clear it out explicitly; it just has its pointer into the stack segment, which can be another data on this place. This is in tact with how unsafe C is — our compiler doesn't add a run-time checks here, but it simulates it. Given a recursive factorial function in C, our compiler will produce a Java method that invokes itself. The local n and other variables of each recursive call go into Java locals or into separate slots in the stack array. When n hits 1 and returns, the stack will naturally unwind, and each return actually goes to the former stack frame with its own value of n, which is how the right result is calculated.

Thesis tried these scenarios to ensure that they work with our approach, leveraging provable mechanisms (the call stack of the JVM and an explicit stack that is well managed). In other words, function definitions compile to JVM methods with JVM bytecode for their bodies, and function calls compile to method invocations. First, recursion is completely covered by the above; no special-casing need apply other than ensuring any shared resources (such as the stack-pointer) are properly diamagnetically handled to isolate call instances. What we have demonstrated is that, with Java's method call semantics and sufficient discipline over the simulated C memory space, we can achieve the correct function call semantics as specified by the C language.

4.21. Dynamic Memory Allocation (malloc and free)

Dynamic memory allocation is supported by a simple runtime memory manager included in the support code of the compiler. We offer standard-library-like malloc() and free() that interact with our simulated memory. In a compiled program, a malloc(size) call becomes a call to runtime helper method of ours (e.g., a method declared as static in CMemoryImplementation class) that would return a pointer (a 64-bit long) to a newly allocated memory block. The same for free(ptr), it invokes a runtime function to deallocate the block. We wrap these in library functions in Java and enclose the allocation logic in it. The bytes blocks in the heap are kept in an array of byte arrays in the malloc implementation. First an array heap containing a certain number of slots (e.g., byte[]s) is prepared, so that we have set of slots with which to work, and it also keeps a simple free list of available slots. When malloc(n) is called, the runtime locates an available slot (or grows the array, if necessary), creates a new byte[n] array, and stores the array in that slot. It constructs a pointer value that is the high 32 bits of a 64-bit pointer, where the slot index becomes the high 32 bits and the offset within the block (which is 0 in the case of a new allocation's base address) becomes the low 32 bits. This pointer is passed back into the C code as a void* (or similar typed pointer). The allocation algorithm is simple; it's O(1) to pull a slot off the free list. If the heap array is full (no free space), the runtime will allocate a larger array.

The free call accepts pointer as a parameter, extracts the block index and offset from it, and frees the memory block by adding that slot back to the free list. Freeing in our model consists on the action heap[index] = null (and so letting that byte array be available for Java's garbage collector, i.e. its memory can be reclaimed) and adding that index to a list of available slots. We don't try to join adjacent blocks or anything complicated like that, since an allocation is its own array; fragmentation in terms of being contiguous in memory is not a concern for this model, except for the management of slot arrays. Note, though, that fragmentation across lots of small arrays might happen, although Java's garbage collector is responsible for ensuring that the memory backing those byte[] objects is dealt with. So far as the C program is concerned, this malloc and free act just like any regular heap allocator.

The allocator returned pointer can be combined with pointer arithmetic and dereferencing just as if these dynamically allocated blocks worked directly in our compiler's pointer representation and internal memory access routines. For instance, when called int *p = malloc(4 * sizeof(int)); the runtime allocates (say it's at slot 5) byte[16] and returns a pointer with block index = 5, offset = 0. When the C code does p[2] = 42; the compiler will emit code to load the address (block 5, offset 8), and store 42 at that address, which happens to be the right place in that byte[16]. This is where the integration comes in: the dynamic allocation only provides a new memory space within the same integrated addressing system.

Our compiler also supporting malloc(0) (which in C can return NULL, but also a unique pointer that should not be dereferenced). In our case, we handle malloc(0) as if it did return NULL. We may decided to return a non-null pointer of size 0 (nothing prevents us from doing this, since we can always allocate a byte[0], return a pointer to it, and cast our array to a byte[] returning a pointer to it). As we are reserving heap[0] for NULL, we would also not use index 0 for an allocate, which would mean that a check for NULL becomes simple.

Our memory manager is rudimentary and does not handle complex scenarios like realloc or calloc (unless specifically implemented similarly). It's single-threaded and does not protect against double-free or use-after-free errors – those remain the responsibility of the C program (just as in a typical C runtime, misuse can cause corruption, and in our case misuse might corrupt our heap structures or lead to unexpected behavior). For example, freeing a pointer twice in a row might put the same slot into the free list twice, potentially causing a later allocation to be given the same slot twice – a problem. In a robust system, you'd prevent that, but in our project we assume well-behaved input or focus on core functionality.

We need to stress: C does not define comparison of pointers between different allocations to be meaningful to compare in terms of ordering, but equality test will simply give you false in case the block index is different (which is ok). And we include malloc and free in the compiled program we link our CMemoryImplementation class. The calls to malloc/free in the C code are recognized by the compiler and turned into invocations of our Java static methods. For instance, a call malloc(expr) in C will be compiled to something like: evaluate expr (size) as an int, then invokestatic com/c2java/CMemoryImplementation.malloc(I)J. That returns a long (JVM long) which represents the pointer. The C code expects a void*, which we represent as a long, so that fits. Similarly, free(p) becomes invokestatic com/c2java/CMemoryImplementation.free(J)V, passing the long. The rest of the C program treats the returned pointer as usual. By implementing dynamic memory this way, our compiler enables data structures of dynamic size (e.g. using malloc to create buffers, linked list nodes) and manual memory management, which are important for many C programs.

The end result is that we've basically implemented a small stack allocator on top of Java memory that now uses malloc/free in a style familiar to programmers while keeping an array of bytes and using GC to keep track of our real memory usage. This fulfills the need of dynamic allocation support in the generated code, and the flexibility of the emulated memory model goes beyond fixed-size or static allocations.

4.22. Enumeration Types

An enum is a self-defined data type for the type with a finite number of unique values. In C, enumerations are way to give a meaningful name to the value but it also makes the code more readable and safer than being used directly. The first enumerator has a default initial value of 0, and each successive enumerator is incremented by 1, unless a specific value is specified. The C standard states that each enumeration creates a new type of integral, but in C, all enumeration constants are of int value. Consequently an enum introduces a new type of the type system of the language and its values behave as integer constants during compilation.

From the perspective of a compiler designer, the definition (at compile time) of the enum itself and the associated representation (at runtime) needs to be treated. The compiler needs to book-keep every enum inscription as a fresh, new type in the symbol table, with each enumerant name to have attached an integer value. Enumerator constants are determined and assigned values during semantic analysis (computing auto-incremented values if not explicitly provided). Such values can be saved in a 32-bit integer (this is consistent with the simulated memory model in the project, where integers and pointers are 64-bit and represented as two integers encoded in 64 bits). Since enumerators are essentially constants, they do not take runtime space, and uses of the enums can be substituted with a constant with the value from the enum in the created bytecode. Generating code for enums is therefore easy: resolution of an enumerator boils down to loading its constant value (i.e. one of the ICONST instructions in JVM bytecode). The actual ASM-based code generation need not consider these special case values, as no special runtime value needs to be created (aside from having its value appear in the.constant pool or be inlined as an immediate).

Despite enums being conceptually simple, the compiler must ensure correct scoping and type checking for them. Each enumeration type may be considered compatible with int, but maintaining the distinct type (e.g., enum Color vs enum State). This could be by the compiler translating each enum to have its own internal type descriptor and allowing implicit int casts when required, as per C's casting rules. And there is also the problem of selecting the size of the integer used for the enum type. C compilers are permitted to select an integer type large enough to hold all of the enumerator values.; for this we can safely assume we can have 32-bits for all enums (because that is what is used by the Java JVM for int as a native type). This makes it easier to be integrated with JVM bytecode. Int the end parsing would need to be adjusted to recognize enum declarations and its constants, as would the semantic analyzer when it assigns constant values and types and minor mods to code generation to make sure enum values are loaded in as constants.

Given the project's existing design, enumerations can be implemented with minimal impact on the runtime memory model. They are primarily affecting compile-time symbol handling and type semantics rather than introducing new runtime structures.

4.23. Type Aliases

Typedef simply creates an alternate identifier for a type. For instance, after typedef unsigned long size_t;, the identifier size_t can be used in place of unsigned long in declarations. The compiler, therefore, must record size_t as an alias of the underlying type rather than as a separate entity.

Implementing typedef in the custom compiler is primarily a matter of parser and symbol table management. The parser needs to recognize typedef declarations and distinguish them from regular variable or function declarations. In practice, C compilers handle this by maintaining a separate namespace (or flag in the symbol table) for type names introduced via typedef. When the parser encounters the typedef keyword, it should parse the following declarator as usual but instead of generating a variable definition, it should install a new type alias in the symbol table. For example, upon reading typedef int Matrix[10];, the compiler would record Matrix as a synonym for "array of 10 ints". Subsequent appearances of Matrix in the source should be treated as a type specifier. This requires the compiler to treat the set of typedef names as reserved identifiers in parsing expressions/declarations (to resolve the classic C ambiguity between identifiers that name types and those that name variables).

The use of typedef thus influences the lexical analysis or parsing stage: the compiler might need to tag new type names so that the grammar can differentiate between <type-name> and other identifiers. Once the alias is defined, however, the semantic analysis and code generation can effectively ignore the distinction – uses of the alias are replaced by the underlying type. At the ASM bytecode generation stage, a typedef has no direct effect; it does not produce any bytecode, since it's purely a compile-time convenience. The simulated memory model is unaffected by typedefs, because they introduce no new data representation — they simply allow referring to existing types under a different name.

One challenge is ensuring that typedef names do not conflict with other identifiers in inappropriate ways. In C, a typedef name can co-exist with other identifiers as they live in a separate namespace for types. The compiler must maintain this separation.

Additionally, the compiler must correctly handle storage class specifiers like static or extern in combination with typedef (e.g., typedef static int X; is not valid C, but static typedef int X; is also invalid since typedef is not a storage class; the compiler should emit appropriate diagnostics). Another challenge is properly supporting complex declarators: for example, given typedef int *ptr_int;, the alias ptr_int needs to carry the pointer type information such that a declaration ptr_int a, b; is correctly interpreted as

defining two integer pointers. This means the compiler's symbol table entry for ptr_int must encapsulate "pointer to int" type information and the parser must apply it to each declarator.

Overall, typedef integration is conceptually straightforward because it doesn't change the generated code – it affects only how the source is interpreted.

4.24. Switch Statements

C switch statement The switch statement in C is a multi-way branch statement that allows control to jump to different blocks of code, depending on the value of an integer expression. Conceptually, it is equivalent to a series of if-else statements but is generally more efficient and clearer. A case label indicates a constant (integral or enumerated) value, and the control jumps to the matching case block if the switch expression equals the constant value of the case label. A default label can be specified to handle the cases where none of the explicit values apply.

One important aspect of C's switch semantics is fall-through: unless a break statement (or other control transfer like return) is encountered, execution will continue from one case into the next in sequence. This gives flexibility (allowing multiple case labels to execute the same code) but requires the compiler to carefully arrange control flow. In terms of scope, switch statements do not create a new scope by themselves, but variables can be declared within switch blocks or case blocks, which the compiler must handle in its analysis.

Implementing a switch statement in the custom compiler involves generating efficient branching bytecode and managing jump targets for each case. At a high level, the compiler will transform the switch into a decision mechanism that compares the switch expression against the case constants and jumps to the corresponding code. Given the target is the JVM (via ASM), the compiler can take advantage of Java bytecode's dedicated instructions for switch: namely, the tableswitch and lookupswitch instructions. A tableswitch is ideal when the case values are dense (forming a contiguous range), as it uses the value as an index into a jump table. A lookupswitch handles sparse case values by mapping each value to a jump target in a lookup table. The code generation strategy would be as follows: first, evaluate the switch expression and leave the integer result on the stack. Then, if the range of case constants is small and contiguous, emit a tableswitch instruction with a low and high bound covering the range of case values, and provide a list of target labels corresponding to each value in that range (filling gaps with a default label for values that aren't explicitly handled).

If the case values are non-contiguous or very sparse, the compiler would emit a lookupswitch with only the actually present case values mapped to their targets. In either case, one of the targets will be the default case (or a jump to exit if no default is provided). Using these specialized bytecode instructions is

efficient: it allows the JVM to perform the branch in O(1) time by indexing into a table, mirroring the typical jump-table optimization that native compilers perform for switch.

This aligns with the project's design goal of leveraging JVM features via ASM for performance. If the dedicated switch instructions are not used in the first place for simplicity of implementation then a sequence of conditionals can be generated (in byte code an if-then-else chain). However, given that ASM makes it straightforward to emit a tableswitch/lookupswitch, the preferred approach is to use those.

Also, fall-through and break logic must be correctly done by the compiler. The label of each case in the input code must match with a bytecode label. When one case block ends without a break, the code should simply flow into the next case's code – this happens naturally if the compiler doesn't emit an unconditional jump at the end of a case. For break statements inside a switch, the compiler should emit a jump to the end-of-switch label (a label placed right after the switch's entire code block) to ensure execution exits the switch. This requires the compiler's code generation phase to maintain an active break target for the switch context so that any break inside the switch knows where to jump. A continue inside a switch (if it appears within a loop that encloses the switch) should be handled carefully, but that is a nested control-flow concern. The default label, if present, is simply another target in the switch's jump table or lookup structure, and if no default is given, the switch should jump to the end-of-switch for any unmatched value.

One thing is to ensure case values are handled somewhat properly for cases where you have duplicate values or for ranges. The compiler can then require that in one and the same switch no two case labels can be having same constant (that would be a compile time error). And it must deal with the coercion of the switch expression to the type of the case labels (C permits char or enum types in switch which are promoted to int). In the context of this compiler, the expression's value will be an int-sized value on the simulated stack (since smaller types would be promoted during expression evaluation).

Another challenge is the support for case ranges and large value spans. The tableswitch bytecode requires a contiguous range of indices; if the case constants have a large gap (e.g., case 1 and case 10000 with nothing in between), using a tableswitch would entail a large table mostly filled with default entries, which is inefficient. The compiler should choose lookupswitch in such cases. Implementing that decision requires collecting all case constants during code generation and analyzing their density. The ordering of code in the output is also a consideration: the natural approach is to emit the code for each case in source order and have all the jump targets point into this code. The compiler's intermediate representation can accumulate the case labels and their corresponding code blocks before output.

Also, if you create variable declarations within the switch (like near the beginning of a case body) the compiler would have to add a scope (low stack requirement or not) or have to do some kind of stack

manipulation to handle the lifetimes for these further down the road. Implementation do not have to go that way as often as the attribution of code to blocks labels would suggest. The compiler must be careful that the jump correctly generate, since these should not bypass initialization of local variables. A solution might be to require braces around case blocks whenever variables are introduced, or perhaps to treat the beginning of a case as starting with a new implicit block in semantic analysis.

4.25. Preprocessor Support

A C compiler should have complete support for the preprocessor, which is essential since the preprocessor (which includes features such as macro expansion and conditional compilation) works on the code before it's ever compiled. C preprocessor is mostly a text transformation stage: it processes directives (lines beginning with #) to include header files (#include), define macros (#define and #undef), conditionally compile code (#if, #ifdef, #ifndef, #else, #endif), and do a couple of other transformations, including macro stringification and token concatenation.

In the context of a custom compiler written in Java, adding preprocessor support means introducing an initial compilation phase that reads and transforms the source text according to these directives before the parsing phase. This is typically done by either integrating a preprocessing step into the compiler's front-end or by invoking an external preprocessor and then feeding the result to the compiler. Given the project's goals, an integrated approach is likely preferred to keep everything in pure Java. One approach is to build a preprocessing module that works on top of the input stream for the source code. This module would read the source file line by line and process preprocessor directives accordingly When for instance, we see #include "file. h", the preprocessor module would find the file, read its contents and push those contents into the compilation stream.

For macro definitions, encountering a directive like #define MAX 100 should cause the preprocessor to record a macro named MAX with replacement text 100. Then, every subsequent occurrence of MAX in the source (outside of strings and comments) should be replaced with 100 during preprocessing. Macros with parameters (e.g., #define SQR(x) ((x)*(x))) require parameter substitution when they are expanded; the preprocessor must detect macro invocations in the code and substitute the arguments appropriately into the macro body. Conditional directives like #ifdef DEBUG or #if X > 5 are handled by evaluating the condition (which may depend on macros or constants) and then including or skipping sections of code accordingly. Implementing these features will likely involve writing a mini-scanner and parser for the preprocessor directives themselves, as well as maintaining data structures for macro definitions (a macro symbol table) and an input buffer mechanism to handle included files and macro expansions. A recursive or stack-based inclusion mechanism is needed so that included files can

themselves contain includes or macros, and once an included file is fully processed the preprocessor returns to the original file's stream. Integration with the compiler: The output of the preprocessor phase is a stream of tokens or text that should then be fed into the existing C parser. This implies that the preprocessor either operates as a textual substitution system feeding a new combined source file to the main compilation, or it can be more tightly integrated by feeding tokens directly to the parser on the fly. A simpler design is to have the preprocessor produce a single expanded source string (or list of tokens) which is then given to the normal compilation pipeline. This keeps a clear separation of concerns: the preprocessor operates first, then the compiler's parser/semantic analysis runs on the expanded code.

However, care must be taken to maintain accurate line number information for error reporting; typically, preprocessors use line markers (like #line directives) or keep track of original file positions so that the compiler can report errors in terms of the original source files. Since the project uses ASM for code generation and simulates memory at the runtime level, the preprocessor mostly impacts the front-end and does not directly touch the memory model or bytecode generation.

Nonetheless, robust preprocessor support is vital for real-world C code, which often relies on macros and includes to function correctly. Writing a C preprocessor (even a subset of it) is non-trivial. One challenge is macro expansion complexity — macros can be recursive or self-referential (the preprocessor must detect and prevent infinite expansion loops), and macros can shadow other macros or be undefined later. The rules for expansion (when to expand, when not to, especially for macros used as arguments to other macros, or the use of the # and ## operators for stringizing and token concatenation) are intricate. The compiler implementer must carefully follow the standard's specification for macro expansion order.

Another challenge is implementing expression evaluation for #if directives. These expressions use a subset of C's expression syntax (constant expressions with integer arithmetic, defined operators, etc.) and the preprocessor must evaluate them (e.g., #if MAX_SIZE > 256). This might require writing a small evaluator or reusing the compiler's expression evaluator in a simplified form, operating on constant values and defined macro values. File inclusion also introduces the need for file path management (searching include directories) and preventing multiple inclusions (which is often handled by header guards or #pragma once — the preprocessor should support at least the conventional include guard idiom by processing multiple includes correctly).

Then there is the issue of producing the correct spacing so that tokens aren't accidentally concatenated during macro expansion (the preprocessor works in tokens, not raw text, to avoid this problem). One can utilize existing libraries, or even call an external cpp for preprocessing to not reinvent the wheel, but If this is a college project, it is very informative to implement a simple preprocessor in Java to understand how compilers work. The interface between preprocessor and parser needs to be

clean: one way to do that is to combine preprocessing with lexical analysis so that as the lexer reads tokens it simultaneously checks for macros to expand. That can be quite efficient, but it makes the lexer more complicated. This would be slightly less performant (in the same way that some source-to-source preprocessing is) but should be simpler to implement and debug.

In summary, adding preprocessor support will significantly enhance the compiler's ability to handle real C code, and it requires careful attention to the lexical handling of the language. It is a distinct phase that doesn't directly involve the ASM bytecode generator or the simulated memory model.

5. Design with Clang++

The first project consists of three parts. The design can be seen from the diagram below.

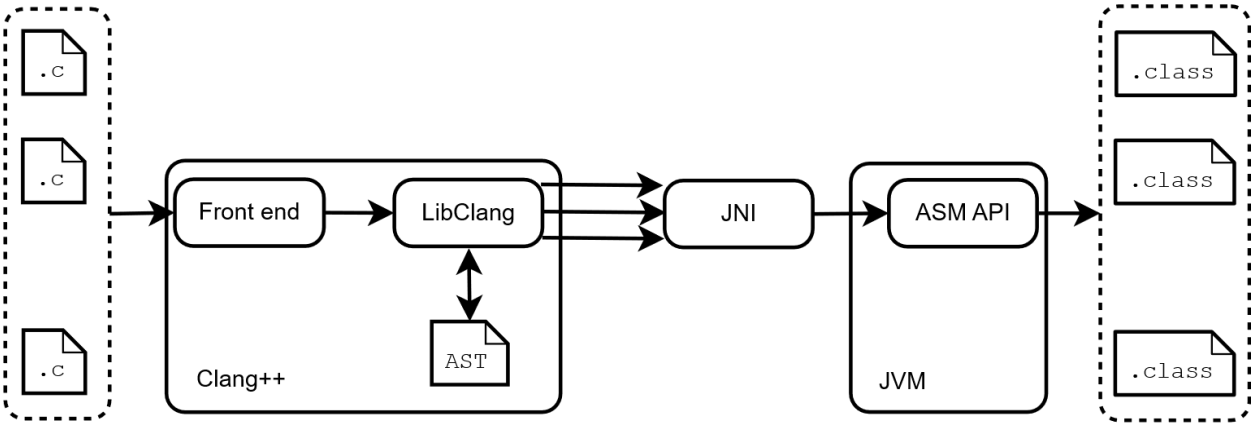

Figure 6 Design of project with clang++.

The design in Figure 6 shows the flow of translation of .c source code to a .class bytecode. To generate bytecode several steps are taken. Represent all .c files of a library as a single file. The way of doing it is to use clang++ front end. Clang++ can generate a clang AST that does not depend on source files. clang AST is an intermediate fully type-resolved representation of the whole C library structure.

5.1. Implementation and design decisions

There are three possibilities to generate and then traverse clang AST. One way is to write clang plugins. Clang plugins work during the compilation process. They give full control over clang AST. Another way is to use Lib Tooling. Lib Tooling can be used for writing stand-alone tools. Lib Tooling is a good way to manipulate clang AST as it Gives full control over it. Lib Tooling can also be used to call clang plugins. However, Lib Tooling is an unstable API that is being changed rapidly.

The most efficient way of manipulation of clang AST is to use Lib clang API. Lib Clang is a stable and backward-compatible API. It is a small API that allows parsing source code into an AST, loading parsed AST, traversing the AST, associating physical source locations with the AST elements. It gives an abstraction from clang AST. The abstraction is implemented through a cursor which helps to iterate clang AST. The cursor allows the user to be abstracted from clang AST.

The decision to use Lib Clang in a tree traversal was made. Lib Clang is relatively small, backward compatible and stable. In the process of traversal, the Java Native Interface (JNI) is called. JNI is called from C++ code. It acts as a one-way mediator between Lib Clang and JVM. JNI is responsible for passing

the data from the clang AST node to Java static methods. JNI is called before the cursor that traverses clang AST changes its location. JNI calls static methods on the Java side and passes the arguments array. The size of an array is dependent on the clang AST node. JNI is called according to cursor kind. If kind is a function declaration, the number of parameters and their types, return type of a function, and its signature is passed to a relevant static method. If kind is a declaration of a variable then its name and type are passed to a relevant static method. If kind is a structure or union then, their names are passed to a relevant static method. If kind is a field declaration then the parent name, field name, and field type are passed to a relevant static method. On the Java side, ASM API is used to generate files in bytecode format.

Compiled Java class consists of several sections. Section for access modifiers, name, superclass, interfaces, and annotations of the class. One section for each field declared in the class: modifiers, name, type, and annotation. One section for each method includes constructor: modifiers, name, return type, parameter types, and annotation of the method. The section of the method also contains compiled code of a method as a sequence of bytecode instructions. All types' names are be fully qualified. The compiled class also contains a constant pool section that holds all constants that appear in the class. ASM takes care of the constant pool during the manipulation. Method and type descriptions in the class file are different from a source file. Descriptors of primitive types are single characters are in Table 1.

Arrays are represented as a square bracket before the type descriptor: [I – int[]. Objects are described with L followed by full names that include package and Class name and semicolon: Ljava/lang/String; - String. Method descriptors start with left parenthesis: (Iljava/lang/String;)I - int method(int i, String s). ASM API is based on the ClassVisitor Abstract class.

Each method of ClassVisitor Abstract class visits one section of the class. Methods of ClassVisitor class should be called in a specific order. First should be called visit(), after that visitSource() or visitOuterClass(), then any number in any order visitAnnotation() or visitAttribute(), visitInnerClass(), visitField() and visitMethod(), and terminated by a single call to visitEnd(). ASM API has two more core components to manipulate or generate class files: ClassReader – parses class and calls visit methods of ClassVisitor instance; ClassWriter – directly writes class in a byte form, the class can be obtained calling toByteArray() method.

The decision to use event-based ASM API in bytecode generation was made. Tree-based representation is not needed as it will reconstruct a redundant syntax tree. The tree is already being constructed by Lib Clang. After receiving arguments from the JNI, static methods implement ASM API to generate bytecode. At the end toByteArray() method is called to get .class file. Currently, there is a possibility to translate method signature; structures; structure fields of type short, int, double; unions;

union fields of type short, int, double.

Table 1 Qualified type names in JVM

Z	boolean
C	char
B	byte
S	short
I	int
F	float
J	long
D	double

The problem is that the C library can contain several .c files that are dependent on each other. But the resulting .class files don't have to follow the same structure. It is also hard to decide how to translate functions to methods. Java can represent methods as static members. They will not be bound to instances. Also the problem is to decide how to represent unions in Java. Java does not allow direct memory manipulations. Each type in Java has a predefined size and can not be changed. Solutions to the described problems were made are described below.

To represent the structure properly, the resulting code will be split into several parts. Structures and unions of C code will be translated to separate class files with the structure name. Class files representing structure or union will contain public instance fields and a default constructor. Instance fields were generated from fields of structure or union used in the C library. These class files are placed in a separate package. The methods of C code will be placed in separate folders. The folder name will be taken from the header file. All methods of header files will be presented as static members. To represent unions in Java – Javolution API [Jea20] can be used. Javolution API can manipulate memory on a bit level.

All the decisions made for this compiler are ambiguous. The development was done from 2021 to 2023. Project was dropped due to its unclearness and complexity. The code was moved from trunk to separate branch and abandoned.

6. Design with ANTLR4

The second implementation consists of C2Java project that uses ANTLR4 and ASM libraries to generate Java bytecode from C source file (Figure 7). The design can be seen from the diagram below.

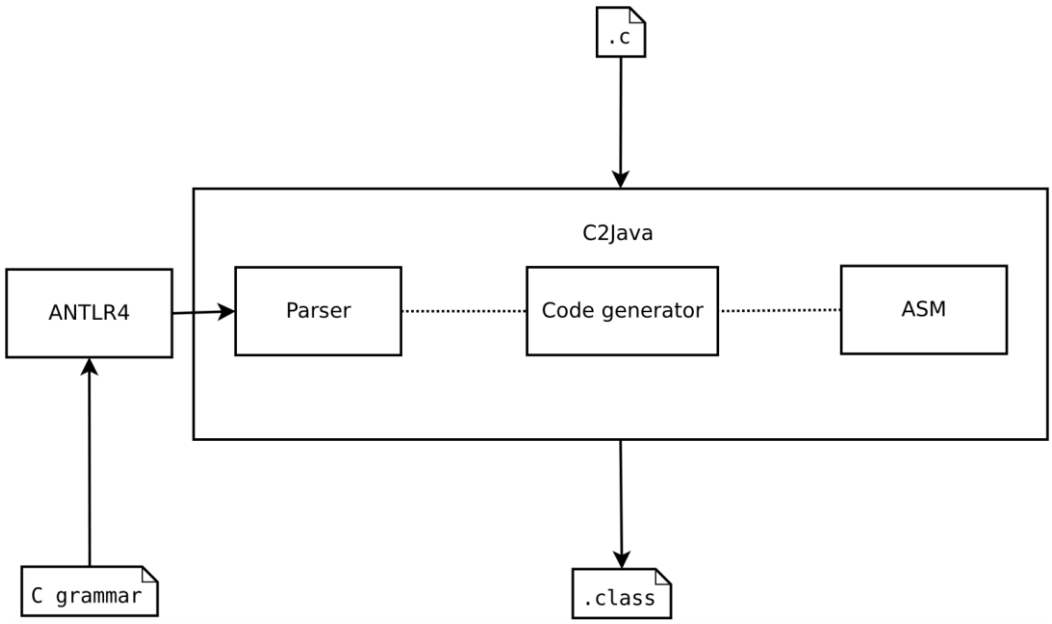

Figure 7 Design of project with ANTLR4.

In order to generate bytecode using this method, we must utilize the ANTLR4 parser generator in conjunction with a grammar for the C language. The process begins by providing the C grammar [Ter13a] to ANTLR4, which generates Listener and Visitor classes. These classes are then utilized to traverse the ParseTree that is generated from the .c file. During this traversal, functions of the ASM library are called in order to produce bytecode for the specific rules outlined in the C grammar. This process allows us to effectively generate bytecode for a C language library by utilizing the ANTLR4 parser generator and the C grammar.

6.1. Implementation

The second implementation of a project differs from the first one by replacement that was found for clang++ compiler. The replacement is ANTLR4. ANTLR4 is a parser generator that is commonly used to develop language parsers and interpreters. It is the most recent version of ANTLR4 and has been designed to be more efficient, more flexible, and easier to use than its predecessor. To generate a parser using ANTLR4, a grammar describing the structure and syntax of a language must be provided as input. The generated parser can then be used to parse and process input in that language. The parser can be employed to construct compilers, interpreters, translators, and other language processing tools.

The grammar required for ANTLR4 to create a parser defines the syntax of the language and includes the rules for constructing valid sentences in the language. It is written using a specialized syntax called ANTLR4 grammar syntax, which consists of a series of rules that define the structure of the language. Each rule has a name and a definition specifying the elements that can be used to build valid sentences in the language.

In ANTLR4, a listener is a class that implements a set of predefined callback methods. These methods are triggered by the parser as it processes input and allow the listener to be notified of events that occur during the parse. Listeners listen for events and take some action when they receive them. For example, a listener might be used to print the tokens encountered during the parse or to construct an abstract syntax tree (AST) from the parse. To use a listener, you must create a subclass of the BaseListener class and override the callback methods of interest. You can then instantiate your listener class and pass it to the parser. The parser will invoke the appropriate callback methods on the listener as it processes input.

A visitor is a class that visits the nodes of an AST and performs some action on each node. It is similar to a listener, but it operates on the AST rather than the input stream. This visitor is generated by ANTL4. To use a visitor, you must create a subclass of the AbstractParseTreeVisitor class and override the methods of interest. You can then create an instance of your visitor class and call its visit method, passing in the root node of the AST. The visitor will traverse the AST, calling the appropriate methods on each node as it goes. Both listeners and visitors are useful for customizing the output of the parser. They enable you to write code tailored to the structure of the language being parsed, rather than having to write generic code that handles all possible input.

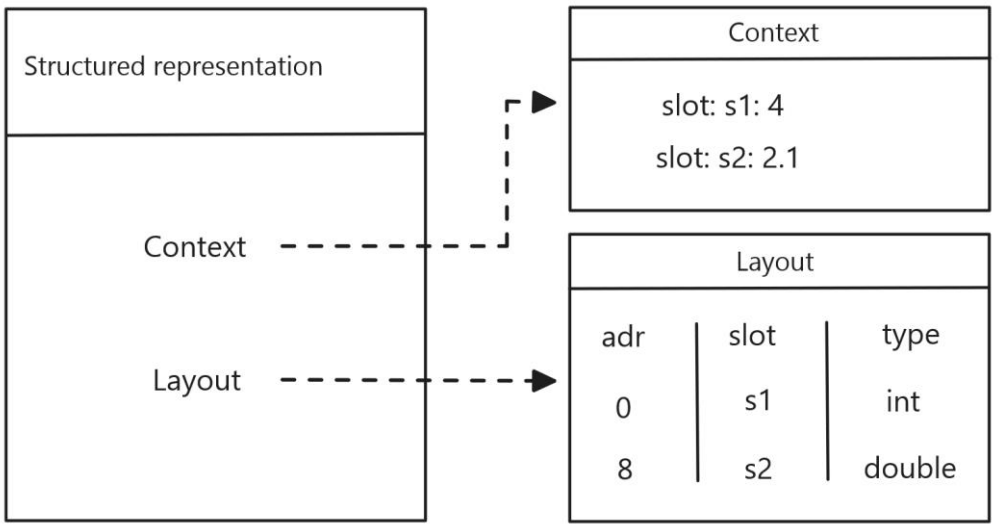

*Figure 8 Structured representation of pointer (image was adapted from [GRC*15])*

The pivotal aspect of the C language resides in its utilization of pointers. To reconcile the absence of direct pointer manipulation in Java, the approach advocated by Grimmer[GRC*15] entails the utilization of structured objects for emulating pointer behavior. The structural object encompasses both content and layout components. The content denotes an entity that retains information pertaining to all values assigned to its constituent members, termed slots, within a given context. Each slot is capable of accommodating primitive values, another structural object, or an address value. The layout component, conversely, serves to record the offset, type, and slot designation of each member within the context. It facilitates the correlation of offsets with respective slots and specifies the type of the stored member. Figure 8 illustrates the depiction of a pointer in the form of a structured object.

TDD is a disciplined approach to software development in which developers write tests for small pieces of functionality before writing the code to implement those features. This process helps to ensure that the code meets all requirements and functions as intended. It also makes it easier to maintain the code, as the tests act as a safeguard against unintended changes to existing functionality. To follow the TDD process, a developer first writes a test for a small unit of code, then writes the code to make the test pass, and finally refactors the code to meet desired standards of design and performance. By iterating through this cycle repeatedly, developers can create high-quality code that has been thoroughly tested and validated, leading to faster development times and reduced risk of defects. This approach can also accelerate development times by allowing for incremental testing and modularization of code. An example of a test class implemented can be seen below.

7. Design with ANTLR4 and emulated memory

The third implementation consists of ANTLR4, ASM and the CMemoryImplementation class - memory simulator class to generate Java bytecode. This class was created for C memory simulation inside JVM.

7.1. Implementation

The compiler translates C source programs into executable Java bytecode adopting a modular design based on the ANTLR parsing framework and the ASM bytecode generation library. The process begins in the C2JavaWithHeap class, which orchestrates the entire compilation pipeline. A C source file is first read and tokenized using a custom ANTLR lexer (CLexer) and then parsed into an abstract syntax tree by CParser. Semantic processing is carried out through a set of custom listeners (CStructListener, CNameListener, and CFunctionListener) that walk the parse tree and extract relevant metadata while producing bytecode. Class generation is handled using ASM's ClassWriter and ClassVisitor constructs. A Java class is dynamically created with public access and a default constructor. Struct declarations from C are processed first and stored in a scoped symbol table (SymTabNested) via the CStructListener. Function signatures, return types, and parameters are identified and recorded by the CNameListener, which also constructs type descriptors. Finally, the CFunctionListener translates function bodies into JVM bytecode, referencing previously collected metadata.

A core challenge in translating C to Java is reconciling their distinct memory models. Java does not allow direct manipulation of memory addresses, nor does it permit storing raw object references in primitive-type arrays such as byte[]. To address this, the compiler uses an emulated memory system implemented in the CMemoryImplementation class. This system includes a static array of object references (heap[]), which acts as a translation layer from C-style integer addresses to Java arrays and objects. Each element in the heap array serves as a memory block that can store raw data or act as a placeholder for C constructs such as arrays or structs. The heap[1] slot is reserved for emulating the C stack and supports functions that require addressable automatic storage. The design mandates that, wherever possible, C variables be implemented directly using the Java method stack for efficiency. This includes local scalars, arrays, and even automatic structs, provided their addresses are never taken (i.e., no pointer to them is used). However, if a C function requires the address of a local variable, the compiler switches to using heap[1] as an emulated C stack. In this case, a virtual register—referred to as the C Stack Pointer (CSP) or C Frame Pointer (CFP)—is introduced as a runtime variable. The CSP tracks the top of the emulated stack. Functions that utilize this memory model must increment the CSP upon entry and decrement it upon exit. Additionally, to preserve stack integrity, such functions must catch and

rethrow any exceptions while ensuring the CSP is correctly restored in the catch block. Memory allocation is achieved through the malloc function, which returns a 64-bit long pointer combining heap index and byte offset. If the heap is full, it is resized dynamically. Memory is released via the free function, which reclaims the corresponding slot and updates the free list. Struct sizes can be computed using the sizeof function, which uses Java reflection to examine the field layout of Java class equivalents of C structures. The sizeof function can be removed in next versions of compiler because CStructListener saves size in symbol table.

Results and conclusions

Key Results

1. C-to-Java Bytecode Compiler Prototype. A basic compiler was developed. Compiler translates C source code into Java bytecode using ANTLR4 for syntax parsing and the ASM library for bytecode generation. The prototype supports small set of core C features. Compiler generates executable .class files which can be run on JVM.[1]
2. Simulated C Memory Model on JVM. A custom Java class, called CMemoryImplementation was created. Class simulates C memory allocation and deallocation. It supports both C stack memory and C heap memory behavior. This component provides an abstraction layer that mimics direct memory access in C using managed pointers as longs.
3. Partial Feature Implementation. The compiler currently supports a limited subset of C constructs. They include: primitive types, arrays, pointers, functions, control flow structures (if, return, goto). Also compiler can generate bytecode for basic arithmetic and logical operations.

Conclusions

1. Feasibility Confirmed for Partial Translation. The work shows that translating a subset of C to Java bytecode is technically feasible. The essential C constructs can be simulated in Java environment.
2. Practical Lessons on JVM-Based Compilation. The process of building the compiler. Deep understanding c too bytecode-level compilation. The complexities of simulating low-level C semantics such as pointer arithmetic.
3. Prototype Limitations and Future Direction. Compiler serves as a steppingstone toward a more complete tool.

Caution for Integration Claims. The thesis does not support full integration of the COD CIF parser and does not demonstrate improved parsing performance. These remain goals for future research once the compiler achieves a more complete feature set.

[1] svn://saulius.grazulis.lt/c2java

References and sources

[BAd09] Brian Alliet & Adam Megacz. (2009) Complete Translation of Unsafe Native Code to Safe Bytecode. Available from: http://www.megacz.com/berkeley/research/papers/nestedvm.ivme04.pdf [Accessed 6 Feb 2020]

[BBB*16] Bernstein, H.J., Bollinger, J. C., Brown, I. D., Gražulis, S., Hester, J. R., McMahon, B., Spadaccini, N., Westbrook, J. D. and Westrip, S. P. (2016). J. Appl. Cryst. 49 277-284. Specification of the Crystallographic Information File format, version 2.0

[BLC02] Eric Bruneton, Romain Lenglet, Thierry Coupaye, ASM: A code manipulation tool to implement adaptable systems (2002). In Adaptable and extensible component systems. Available from: https://www.researchgate.net/publication/228556492_ASM_A_code_manipulation_tool_to_implement_adaptable_systems [Accessed 6 Feb 2020]

[Bob97] Bob Hanson. Java Universal Molecular Browser for Objects (1997). Available from: https://jmol.sourceforge.net/. [Accessed 26 May 2025]

[Cor09] Corey Coogan. (2009) Visitor Pattern: A Real World Example. Available from:

https://coreycoogan.wordpress.com/2009/06/16/visitor-pattern-real-world-example/ [Accessed 26 May 2025]

[CPS19] CPS311 - COMPUTER ORGANIZATION. A Brief Introduction to the MIPS Architecture (2019). Available from: http://www.cs.gordon.edu/courses/cps311/handouts-2019/MIPS%20ISA.pdf [Accessed 7 Jun 2021]

[CYS*03] Christoph Steinbeck, Yongquan Han, Stefan Kuhn, Oliver Horlacher, Edgar Luttmann, and Egon Willighagen. The Chemistry Development Kit (CDK): An Open-Source Java Library for Chemo-and Bioinformatics. Journal of Chemical Information and Computer Sciences 2003, 43, 2, 493-500 doi:10.1021/ci025584y.

[DAH99] Dahm M., Byte Code Engineering. Proceedings JIT'99, Springer, 1999. DOI: https://doi.org/10.1007/978-3-642-60247-4

[Dav14] Dave Hanson. (2014) LCC. Available from: https://github.com/drh/lcc [Accessed 26 May 2025]

[Daw19] Dawei Hucs. (2019) C2j-Compiler. Available from: https://github.com/dejavudwh/C2j-Compiler [Accessed 26 May 2025]

[FWB*06] P. M. D. Fitzgerald, J. D. Westbrook, P. E. Bourne, B. McMahon, K. D. Watenpaugh and H. M. Berman. International Tables for Crystallography (2006). Vol. G, ch. 3.6, pp. 144-198 doi:10.1107/97809553602060000738

[GCC*20] Geoffrey R Hutchison, Chris Morley, Craig James, Chris Swain, Hans De Winter, Tim Vandermeersch, Noel M O'BoyleOpen Babel Documentation (2020). Available from: https://readthedocs.org/projects/open-babel/downloads/pdf/latest/. [Accessed 6 Feb 2020]

[GCD*09] Grazulis, S., Chateigner, D., Downs, R. T., Yokochi, A. T., Quiros, M., Lutterotti, L., Manakova, E., Butkus, J., Moeck, P. & Le Bail, A. (2009) "Crystallography Open Database – an open-access collection of crystal structures". J. Appl. Cryst. 42, 726-729. doi: 10.1107/S0021889809016690

[GDM*12] Gražulis, S., Daškevič, A., Merkys, A., Chateigner, D., Lutterotti, L., Quirós, M., Serebryanaya, N. R., Moeck, P., Downs, R. T. & LeBail, A. (2012) "Crystallography Open Database (COD): an open-access collection of crystal structures and platform for world-wide collaboration". Nucleic Acids Research 40, D420-D427. doi: 10.1093/nar/gkr900

[GRC*15] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Hanspeter Mössenböck. Memory-safe Execution of C on a Java VM 2015

[GMR*08] H. Grgić, B. Mihaljević, A. Radovan. Rochester Institute of Technology Croatia, Zagreb, Croatia. Comparison of Garbage Collectors in Java Programming Language, 2008. DOI:10.23919/MIPRO.2018.8400277

[GRM*2014] Grimmer, Matthias, Rigger, Manuel, Schatz, Roland, Stadler, Lukas , Mössenböck, Hanspeter. (2014). TruffleC: dynamic execution of C on a Java virtual machine. 10.1145/2647508.2647528.

[HAB91] Hall, S. R. Allen, F. H. Brown, I. D. The crystallographic information file (CIF): a new standard archive file for crystallography (1991). DOI: 10.1107/S010876739101067X.

[Her08] Herbert J Bernstein. VCIF2 : extended CIF validation software (2008). Journal of Applied Crystallography 41(4):808-810. DOI: 10.1107/S002188980801385X.

[Hes06] Hestera J. R. A validating CIF parser: PyCIFRW 2006 Journal of Applied Crystallography 39(4):621-625. DOI: 10.1107/S0021889806015627.

[HWS*06] S. R. Hall, J. D. Westbrook, N. Spadaccini, I. D. Brown, H. J. Bernstein, B. McMahon Specification of the Crystallographic Information File (CIF) (2006). ISBN: 978-1-4020-3138-0. pp 20-21.

[Jea20] Jean-Marie Dautelle (2020) Javolution. Available from: https://github.com/javolution/javolution [Accessed 19 Jan 2022]

[Joe18] Joel Wahl. OpenChemLib. (2018) Available from: https://github.com/Actelion/openchemlib/ [Accessed 26 May 2025]

[Ken02] Kent Beck, Test Driven Development: By Example(2002). ISBN: 9780321146533

[Kli13] Manuel Klimek. European LLVM Conference (2013).

Available from: https://llvm.org/devmtg/2013-04/klimek-slides.pdf [Accessed 7 Jun 2021].

[KRi88] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall: Englewood Cliffs, NJ, 1988. ISBN 0-13-110370-9. pp 127-129.

[KVo17] Petr Kudriavtsev, Vladimir Voskresensky, Clank: Java-port of C/C++ Frontend (2017). Available from: https://llvm.org/devmtg/2017-03//assets/slides/clank_java_port_of_c_cxx_compiler_frontend.pdf [Accessed 7 Jun 2021].

[Mar02] Martin Johannes. (2002) A C to Java migration Environment. In: A Dissertation Submitted in Partial Fulfillment of the Requirements for Degree of Doctor of Philosophy in the Department of Computer Science, pp. 2-7. Available from: https://dspace.library.uvic.ca/bitstream/handle/1828/10202/Martin_Johannes_PhD_2002.pdf?sequence=1&isAllowed=y [Accessed 6 Feb 2020]

[MDP*01] Marian Bubak, Dawid Kurzyniec, Piotr Luszczek, Vaidy S. Sunderam: Creating Java to Native Code Interfaces with Janet. Sci. Program. 9(1): 39-50 (2001) doi:10.1155/2001/582127

[Mic16] Michael Kerrisk. Linux manual page. (2016) Overview of standard C libraries on Linux. Available from: https://man7.org/linux/man-pages/man7/libc.7.html [Accessed 26 Feb 2025]

[Moh20] Mohan Embar. (2020) Mips2Java. Available from: http://www.xwt.org/mips2java/ [Accessed 26 May 2020]

[MPM*15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, Nathaniel Nystrom. Use at Your Own Risk: The Java Unsafe API in the Wild. OOPSLA 2015: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and ApplicationsOctober 2015 Pages 695–710 https://doi.org/10.1145/2814270.2814313

[MVB*16] Merkys, A., Vaitkus, A., Butkus, J., Okulič-Kazarinas, M., Kairys, V. & Gražulis, S. (2016) "COD::CIF::Parser: an error-correcting CIF parser for the Perl language". Journal of Applied Crystallography 49. doi: 10.1107/S1600576715022396

[Nal18] Nalin Adhikari: Java Code Engine for Online Code Compilation (2018). doi: 10.13140/RG.2.2.27241.19046

[Ora20] Oracle. (2020) The class File Format. Available from: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html [Accessed 26 May 2025]

[Pou89] Pountain, Dick (1989). Configuring parallel programs, Part 1: The Occam Transpiler, now under development, will make writing software for parallel processing easier. Vol. 14, pp. 349–352. ISSN 0360-5280.

[RGW16] M. Rigger, M. Grimmer, C.Wimmer. Bringing low-level languages to the JVM: efficient

execution of LLVM IR on Truffle (2016). 6-15. 10.1145/2998415.2998416.

[SEi19] Roland Schatz, Josef Eisl. Sulong: An experience report of using the "other end" of LLVM in GraalVM (2019). EuroLLVM'19, April 8-9, 2019, Brussels, Belgium.

[Ter10] Terence Parr: Language Implementation Patterns. ISBN-13: 978-1-934356-45-6. pp 31-34.

[Ter13a] Terence Parr. C 2011 grammar built from the C11 Spec Available from: https://github.com/antlr/grammars-v4/blob/master/c/C.g4 [Accessed: 26 May 2025].

[Ter13b] Terence Parr, The Definitive ANTLR 4 Reference (2013). ISBN: 978-1-934356-99-9. pp 1-17.

[TRi96] History of Programming Languages-II ed. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. ACM Press (New York) and Addison-Wesley (Reading, Mass), 1996; ISBN 0-201-89502-1.

[WHI07] White A., Available from: Serp http://serp.sourceforge.net [Accessed 7 Jun 2021].

Appendix 1. Clang AST

Clang AST is a part of the LLVM project.

```
TranslationUnitDecl 0x67c318 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x67cbb0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x67c8b0 '__int128'
|-TypedefDecl 0x67cc20 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x67c8d0 'unsigned __int128'
|-TypedefDecl 0x67cf28 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x67cd00 'struct __NSConstantString_tag'
|   `-Record 0x67cc78 '__NSConstantString_tag'
|-TypedefDecl 0x67cfc0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x67cf80 'char *'
|   `-BuiltinType 0x67c3b0 'char'
|-TypedefDecl 0x67d2b8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x67d260 'struct __va_list_tag [1]' 1
|   `-RecordType 0x67d0a0 'struct __va_list_tag'
|     `-Record 0x67d018 '__va_list_tag'
`-FunctionDecl 0x6dbf90 <hello.c:1:1, line:8:1> line:1:5 main 'int (void)'
  `-CompoundStmt 0x6dc338 <col:16, line:8:1>
    |-DeclStmt 0x6dc140 <line:3:5, col:14>
    | `-VarDecl 0x6dc0b8 <col:5, col:13> col:9 used x 'int' cinit
    |   `-IntegerLiteral 0x6dc120 <col:13> 'int' 5
    |-DeclStmt 0x6dc1f8 <line:4:5, col:14>
    | `-VarDecl 0x6dc170 <col:5, col:13> col:9 used y 'int' cinit
    |   `-IntegerLiteral 0x6dc1d8 <col:13> 'int' 6
    `-DeclStmt 0x6dc320 <line:6:5, col:18>
      `-VarDecl 0x6dc228 <col:5, col:17> col:9 c 'int' cinit
        `-BinaryOperator 0x6dc300 <col:13, col:17> 'int' '+'
          |-ImplicitCastExpr 0x6dc2d0 <col:13> 'int' <LValueToRValue>
          | `-DeclRefExpr 0x6dc290 <col:13> 'int' lvalue Var 0x6dc0b8 'x' 'int'
          `-ImplicitCastExpr 0x6dc2e8 <col:17> 'int' <LValueToRValue>
            `-DeclRefExpr 0x6dc2b0 <col:17> 'int' lvalue Var 0x6dc170 'y' 'int'
```

Appendix 2. ClangAST generation and traversal using a cursor

```
if ( argc < 2 ){
        cout << ".c file is missing\n";
        return 1;
}

// Command line arguments required for parsing the TU
constexpr const char *ARGUMENTS[] = {};

// Create an index with excludeDeclsFromPCH = 1, displayDiagnostics = 0
CXIndex index = clang_createIndex( 1, 0 );

// Speed up parsing
CXTranslationUnit translationUnit = clang_parseTranslationUnit(
        index, argv[1], ARGUMENTS, extent<decltype(ARGUMENTS)>::value,
        nullptr, 0, CXTranslationUnit_None );

// Visit all the nodes in the AST
CXCursor cursor = clang_getTranslationUnitCursor(translationUnit);
clang_visitChildren( cursor, visitor, 0 );

// Release memory
clang_disposeTranslationUnit( translationUnit );
clang_disposeIndex( index );
```

AST traversal starts in the visit function.

```
CXChildVisitResult visitor( CXCursor cursor, CXCursor parent, CXClientData clientData ){

        return CXChildVisit_Recurse;
}
```

This method always returns CXChildVisit_Recursive. It indicates that all nodes should be visited. The result of a function can also be CXChildVisit_Continue and CXChildVisit_Break. CXChildVisit_Continue will indicate that the cursor should not visit child nodes. CXChildVisit_Break will indicate that the cursor should stop traversal. Traversal is done by the use of visitor patterns [Cor09]. Visitors allow adding a new virtual function to the class without modifying it. The first parameter of the function is a cursor's current position in a tree. The second parameter is a parent node in the tree. The third parameter allows passing additional data for each traversal.

To generate struct as a class following data is needed: struct name. So the code that finds and sends struct name to JNI is pretty simple.

```
CXString cursorSpelling = clang_getCursorSpelling( cursor );
string structName = clang_getCString( cursorSpelling );

jmethodID mid2 = env->GetStaticMethodID( cls2, "generateStructAsClass", "([Ljava/lang/String;)V" );     // find method

if( mid2 == nullptr ){
        cerr << "ERROR: method not found !" << endl;
}

jobjectArray arr = env->NewObjectArray( 1, env->FindClass( "java/lang/String" ), env->NewStringUTF( "str" ));
env->SetObjectArrayElement( arr, 0, env->NewStringUTF( structName.c_str() ));

env->CallStaticVoidMethod( cls2, mid2, arr );                          // call method
```

As seen from the code, JNI calls a static method that exists on the JVM side.

```java
/* This method will create class with constructor
* args contain only one element - struct name
* provided by C++ while calling via JNI */
public static void generateStructAsClass(String[] args) {
        System.out.println("Struct passed: " + args[0]);
        String packageName = "generated/struct/";
        String className = args[0];

        //Create class writer to write class in bytecode format
        //ClassWriter.COMPUTE_FRAMES to automatically set maxs and frames
        ClassWriter cwStruct = new ClassWriter(ClassWriter.COMPUTE_FRAMES);

        cwStruct.visit(V9, ACC_PUBLIC, packageName + className, null,
                "java/lang/Object", null);

        //Create method named <init> for default constructor
        MethodVisitor constructorMethod = cwStruct.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
        constructorMethod.visitCode();
        constructorMethod.visitVarInsn(ALOAD, 0);
        constructorMethod.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V", false);
        constructorMethod.visitInsn(RETURN);
        constructorMethod.visitMaxs(1, 1);
        constructorMethod.visitEnd();

        // Indicate that generation of constructor is done and get byte array
        cwStruct.visitEnd();
        byte[] bytesOfClassToWrite = cwStruct.toByteArray();

        checkClassPackage(packageName);
        writeClassToPackage(packageName, className, bytesOfClassToWrite);
}
```

Variables are currently hard coded. The method parameter is an array of String objects and it does not return any value. The generation of default constructors is a must.

The next part is to populate the struct with fields. This is done by calling another method that visits generated class file and populates it. To populate struct following data needed: type of field, name of field, and parent of field. The code that finds and passes that data can be seen below.

```
if( kind == CXCursorKind::CXCursor_FieldDecl ){

        CXCursor parentCursor = clang_getCursorLexicalParent( cursor );
        CXString parentCursorSpelling = clang_getCursorSpelling( parentCursor );
        string parentName = clang_getCString( parentCursorSpelling );

        CXCursorKind parentCursorKind = clang_getCursorKind( parentCursor );
        if( parentCursorKind == CXCursorKind::CXCursor_StructDecl ){

                jmethodID mid2 = env->GetStaticMethodID( cls2, "populateStructAsClass", "([Ljava/lang/String;)V" );     // find method

                if( mid2 == nullptr ){
                        cerr << "ERROR: method not found !" << endl;
                }

                cout << parentName << ":" << argumentType << ":" << structMemberName  << endl;

                jobjectArray arr = env->NewObjectArray( 3, env->FindClass( "java/lang/String" ), env->NewStringUTF( "str" ));
                env->SetObjectArrayElement( arr, 0, env->NewStringUTF( parentName.c_str() ));
                env->SetObjectArrayElement( arr, 1, env->NewStringUTF( to_string( argumentType ).c_str() ));
                env->SetObjectArrayElement( arr, 2, env->NewStringUTF( structMemberName.c_str() ));

                env->CallStaticVoidMethod( cls2, mid2, arr );                           // call method
        } else if( parentCursorKind == CXCursorKind::CXCursor_UnionDecl

        }
}
```

The method on the Java side accepts a String object array of three elements.

```
/* This method will create instance fields inside the struct class
* Fields will be taken one by one from C code and passes via JNI to this method
* args contain: [0] - class name, [1] - type of field, [2] - name of field*/
public static void populateStructAsClass(String[] args) throws IOException {

        String packageName = "generated/struct/";
        String className = args[0];
        String typeOfField = args[1];
        String nameOfField = args[2];

        //System.out.println(className + ":" + typeOfField + ":" + nameOfField);

        byte[] bytesOfClassToChange = readClassFromPackage(packageName, className);

        ClassReader crStructFields = new ClassReader(bytesOfClassToChange);
        ClassWriter cwStructFields = new ClassWriter(crStructFields, 0);
        ClassVisitor cvStructFields = new ClassVisitor(ASM4, cwStructFields) { };
        crStructFields.accept(cvStructFields, 0);

                //System.out.println("Teeeeeeest " + typeOfField);
                generateField(cvStructFields, className, ACC_PUBLIC, nameOfField,
                        TypeConverter.get(typeOfField).get().toString(), null, null);


        // Indicate that generation of field is done and get byte array
        cvStructFields.visitEnd();
        cwStructFields.visitEnd();
        bytesOfClassToChange = cwStructFields.toByteArray();

        checkClassPackage(packageName);
        writeClassToPackage(packageName, className, bytesOfClassToChange);
}
```

The first element is the name of a class, the second is an instance field type and the last is an instance field name. With the current version of a project, it is also possible to generate method signatures according to C functions.

```
if( kind == CXCursorKind::CXCursor_FunctionDecl ){

        jmethodID mid2 = env->GetStaticMethodID( cls2, "generateMethod", "([Ljava/lang/String;)V" );    // find method

        if( mid2 == nullptr ){
                cerr << "ERROR: method not found !" << endl;
        }

        unsigned returnType = clang_getCursorResultType( cursor ).kind;

        CXString cursorSpelling = clang_getCursorSpelling( cursor );
        string methodName = clang_getCString( cursorSpelling );

        unsigned numberOfArguments = clang_Cursor_getNumArguments( cursor );
        //spaces in method call
        jobjectArray arr = env->NewObjectArray( numberOfArguments + 2, env->FindClass( "java/lang/String" ), env->NewStringUTF( "str" ) );
        env->SetObjectArrayElement( arr, 0, env->NewStringUTF( methodName.c_str() ));

        for( unsigned i = 1, j = 0; i <= numberOfArguments; i++, j++ ){

                CXCursor argumentCursor = clang_Cursor_getArgument( cursor, j );
                unsigned argumentType = clang_getCursorType( argumentCursor ).kind;
                env->SetObjectArrayElement( arr, i, env->NewStringUTF( to_string(argumentType).c_str() ));

        }

        env->SetObjectArrayElement( arr, numberOfArguments + 1, env->NewStringUTF( to_string(returnType).c_str() ));    //arr name

        env->CallStaticVoidMethod( cls2, mid2, arr );                          // call method
        cout << endl;
}
```

The cursor iterates through the function declaration and according to number of parameters calls the proper Java static method.

```
String packageName = "generated/methods/";
String className = "MethodsFromFoo";
String methodName = args[0];

byte[] bytesOfClassToChange = readClassFromPackage(packageName, className);

if (bytesOfClassToChange == null){
        createEmptyClassWithinPackage(packageName, className);
        writeEmptyConstructor(packageName, className);
        bytesOfClassToChange = readClassFromPackage(packageName, className);
}

ClassReader crMethods = new ClassReader(bytesOfClassToChange);
ClassWriter cwMethods = new ClassWriter(crMethods, ClassWriter.COMPUTE_FRAMES);
ClassVisitor cvMethods = new ClassVisitor(ASM4, cwMethods) { };
crMethods.accept(cvMethods, 0);

StringBuilder sb = new StringBuilder();
sb.append("(");

for (int i = 1; i < args.length; i++) {

        if (i == (args.length - 1)) {
                sb.append(")");
        }
        sb.append(TypeConverter.get(args[i]).get());
}

generateMethodSignature(cvMethods, methodName, sb.toString(), null, null);

// Indicate that generation of methods is done and get byte array
cvMethods.visitEnd();
```

The implementation of Javolution API to the moment of thesis submition is not implemented.