



International Journal of Emerging Technologies in Computational and Applied Sciences (IJETCAS)

www.iasir.net

JAVA TO C COMPILER (J2CC)

Nair Saarika Bhasi, J. Shylaja, Arun.V, T.S.Niveda

Department of Information Technology,

Amrita School of Engineering, Amrita Viswa Vidyapeetham University, Amrita Nagar post, Ettimadai,
Coimbatore-641112, Tamil Nadu, India.

Abstract: To design a flexible compiler to convert a java program to a C program and compile it with the aim of reducing the time complexity. The main objective of the project is to combine the robustness of Java and speed factor of C. Though the implementation seems to be from object oriented programming language to procedural language, the project concentrates in reducing the execution time of the program and utilizing the rich libraries of Java. This project can be used as a interface for many real time projects that are time-dependent.

Keywords: Java, Compiler, OOPS, procedural , C, C++

I. INTRODUCTION

C is an imperative (procedural) language. It was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support. C is therefore useful for many applications that had formerly been coded in assembly language, such as in- system programming. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant and portably written C program can be compiled for a very wide variety of computer platforms and operating systems with few changes to its source code. The language has become available on a very wide range of platforms, from embedded microcontrollers to supercomputers.

Java is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to byte code (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture. With the advent of Java, most of the development activities in the recent times have experienced a shift from C based languages to Java based languages. While this is advantageous in a way, it has its disadvantage too. As programs written in Java language runs on a virtual machine, it runs somewhat slowly compared to other programs. A rather worse problem is that the programs do not always work correctly even if they are written correctly, because a Java virtual machine may be written incorrectly. It is difficult to write a perfect program, especially if it is as big as a virtual machine, so that programs written in Java language tend to suffer from slightly different problems depending on the platforms on which the Java virtual machines run. In C, it's easier to work with hardware directly. This is the reason why embedded C is still widely in use than embedded JAVA. C programs can definitely be faster than Java Programs since Java is partially interpreted.

On the other hand, Java is a strong, platform independent language with powerful features such as garbage collection, error handling and the powerful library framework. Developing in Java provides us with various options and scope of improvement. The deployment in Java is what seems to be less advantageous compared to deployment in C. This study aims to provide a solution to this particular problem. The objective of the study is to devise a method that can be used to practice developing in Java and deploying the program in C. This allows us to harness the rich features of the Java programming language while developing and at the same time, eliminate the need for a JVM to deploy the code. This way, the advantages of both the languages are brought forward, eliminating their drawbacks. The proposed plan is to develop a Java-to-C translator which performs the task of syntactic conversion of the Java language to C language.

II. ANALYSIS OF JAVA AND C

The first step of the approach starts with understanding and comparison between the both the languages. On addressing the way of compilation of two programming language and according to [1], compared to C oriented languages, Java has an overhead in terms of execution time and memory that it uses. To prove this fact we undertook an analysis over execution time and memory of C and Java programs from an online programming

competition forum from [3]. The execution time and memory values that are recorded for a particular problem from [4] were plotted in a graph. The result showed that Java is costlier in aspect of memory and speed and the result will be in agreement with solutions in Java and C to all kinds of problems.

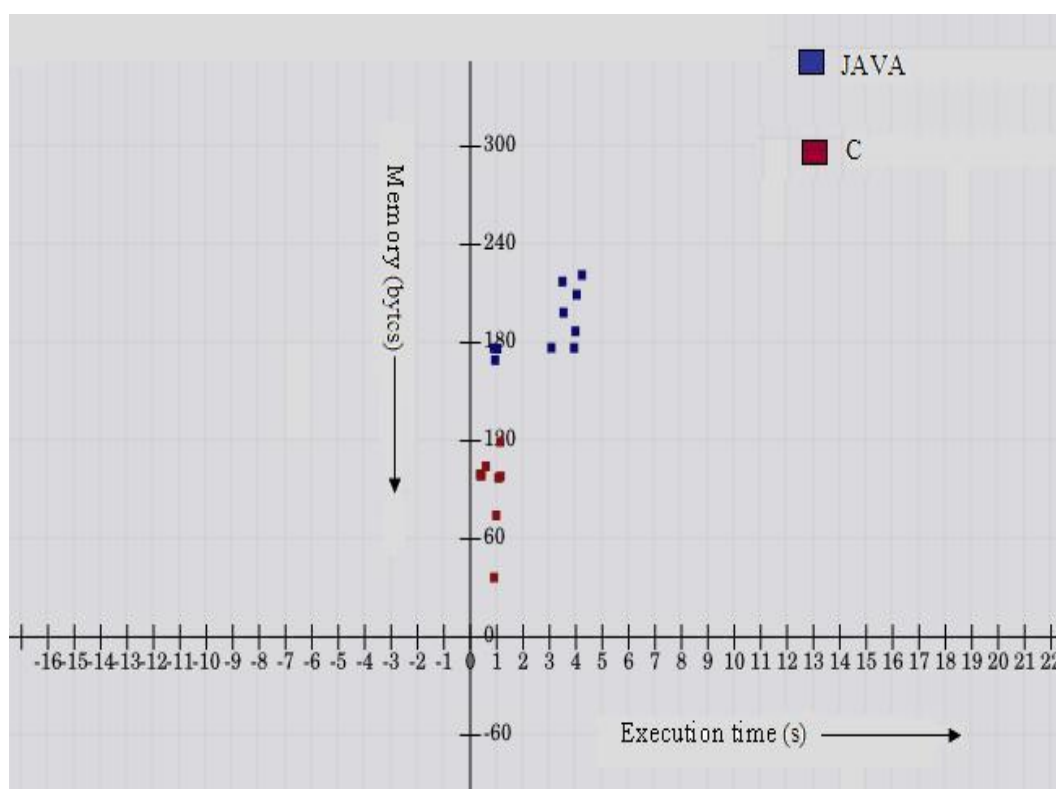


Figure:1.1 Analysis on Java and C

II. ARCHITECTURAL DIAGRAM

The first step for constructing the J2CC is to analyze and understand the work flow and design the Architectural diagram. Figure 1.2 shows the Architectural diagram of J2CC. The first phase is Lexical Analysis (to emit tokens), second phase is syntax analysis (writing CUP grammar for the Java specification), third phase is semantic analysis(writing Abstract syntax definition) and finally Target code generation(output i.e. a C program). The detail description of each phase will be described in the section III

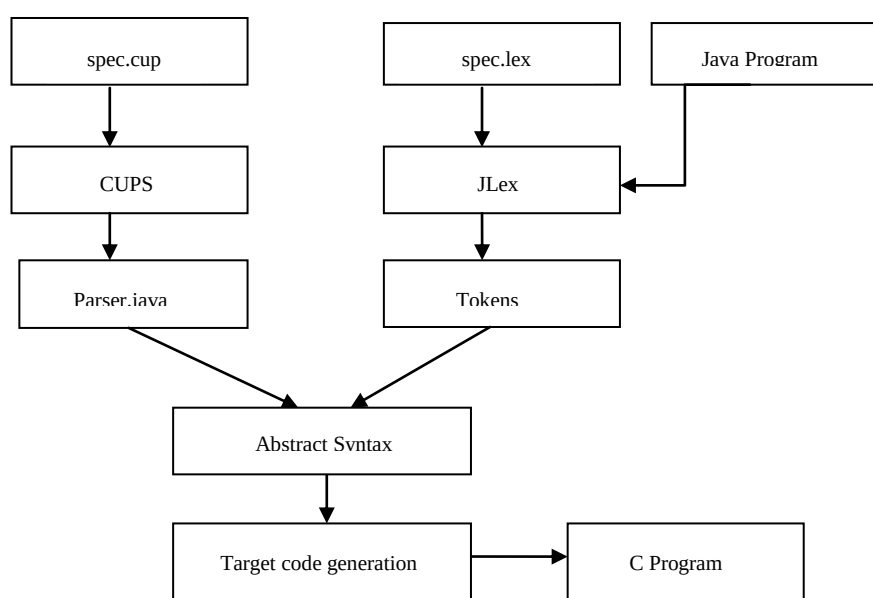


Figure1.2 Architectural diagram

The work flow starts with writing `spec.lex`, considering all the data types, operators and writing regular expression of Java here a input is a Java program and output will be a stream of tokens used in the input program. Then writing `spec.cup`, here we write cup specification for the entire Java language after identifying the syntax. The `parser.java` is the output of the Cup specification i.e. Syntax analysis phase and tokens are output of Lexical analysis. Then we write Abstract syntax for the grammar. The abstract classes are written based on the usability of the language specification in C. According to [6], the current embedded implementations of Java impose tight constraints on functionality and require significant storage space. It demonstrates a Java-to-C compilation strategy that enables the use of Java on a wide range of embedded platforms by removing many of the constraints on functionality and code size and is also applicable to embedded systems without a JVM. The generated executables were found to be over 25 times smaller than those generated by a Java-to-native code compiler. The method aims to eliminate the use of a JVM by converting a Java code to a C code which is then compiled by a C compiler. In this method, all Java library classes are compiled into a library and the required ones are loaded in the runtime. The code is analyzed and unnecessary classes, methods and fields are discarded. This leads to generation of highly optimized C code.

III. TOOLS

Lexical analyzer reads input characters and produces a sequence of tokens as output (`nexttoken()`). A program or function that performs lexical analysis is called a **lexical analyzer**. A lexer is generally combined with a parser, which together analyze the syntax of programming languages. A lexer is itself a kind of parser – the (context-free) syntax of the language is divided into two pieces: the lexical syntax (word structure), which is processed by the lexer; and the phrase structure, which is processed by the (phrase-level) parser. The lexical syntax is usually a regular language, whose atoms are individual characters, while the phrase syntax is usually a context-free language, whose atoms are words (tokens produced by the lexer). A lexical analyzer breaks an input stream of characters into tokens. Lex is a lexical analyzer generator for the UNIX operating system, targeted to the C programming language. Lex takes a specially-formatted specification file containing the details of a lexical analyzer. This tool then creates a C source file for the lexer.

The JLex utility is based upon the Lex lexical analyzer generator model. JLex takes a specification file then creates a Java source file for the corresponding lexical analyzer. Format of JLex specification has three parts, first is User code-It have the Java source code to be copied into the generated Java source file. It contains utility classes or return type classes you need. Second is JLex directive-have the macro definition and third is Regular Expression-These rules specify how to divide up the input into tokens. Each rule includes an optional state list, a regular expression, and an associated action.

The format can be like

User Code

%%

JLex Directives

%%

Regular Expression rules

The J2CC compiler does the parsing action with the help of LALR type parser. This section gives a brief idea of LALR parsers and working of the parser. LALR parsers are based on a finite-state-automata concept, from which they derive their speed. The data structure used by an LALR parser is a Stack with pushdown automaton (PDA). A deterministic PDA is a deterministic-finite automaton (DFA) that uses a stack for a memory, indicating which states the parser has passed through to arrive at the current state. Because of the stack, a PDA can recognize grammars that would be impossible with a DFA; for example, a PDA can determine whether an expression has any unmatched parentheses, whereas an automaton with no stack would require an infinite number of states due to unlimited nesting of parentheses.

LALR parsers are driven by a parser table in a finite-state machine (FSM) format. An FSM is very difficult for humans to construct and therefore an LALR parser generator is used to create the parser table automatically from a grammar in Back-Naur Form which defines the syntax of the computer language the parser will process. The parser table is often generated in source code format in a computer language (such as C++ or Java). When the parser (with parser table) is compiled and/or executed, it will recognize text files written in the language defined by the BNF grammar. LALR parsers are generated from LALR grammars, which are capable of defining a larger class of languages than SLR grammars, but not as large a class as LR grammars. Real computer languages can often be expressed as LALR(1) grammars, and in cases where a LALR(1) grammar is insufficient, usually an LALR(2) grammar is adequate. If the parser generator handles only LALR(1) grammars, then the LALR parser will have to interface with some hand-written code when it encounters the special LALR(2) situation in the input language.

As in [2], the Cups does the syntax analysis of the grammar. A parser specification for CUPs consists of a set grammar rules. The CUP-generated parser implements semantic values by keeping a stack of them parallel to

the state stack. Where each symbol would be on a simple parsing stack, now there is a semantic value. When the parser performs a reduction, it must execute a Java – language semantic action; it satisfies each reference to the right-hand-side semantic value by a reference to one of top k elements of the stack (for a rule with k right-hand-side symbols). When the parser pops the top k elements from the symbol stack and pushes a non terminal symbol, it also pops k from the semantic value stack and pushes the value obtained by executing the Java semantic action code. The J2CC compiler's Parsing phase with the CUPS (LALR) parser firstly adds the CUPS (Common Utility Parser) package following the lexical analysis. The lexicons are made to emit from *spec.lex*. Then a separate *spec.cup* file is written such a way that the file gets the lexicons emitted from *spec.lex* in the form of command line argument. With the input file specified (i.e. Input java file for which the transition of language happens) is accepted by separate Java file. With the Java specifications written in the *spec.cup* whole project is built and made to generate the parse. The generation of the parser is accepted only if the *sym.java* and *parser.java* is created in the current working directory of the project. The *sym.java* file contains the enumeration of all terminal symbols specified in the project. The *parser.java* file turns the grammar into separate functions, Each function checks for the syntax and the semantic(This is a point where abstract syntax classes are called for the next phase).Now running the Java file that is written for accepting the Java file(which is used for transition) gives a parsing successful message if the constructs are maintained. Errors are flashed by CUPS if the constructs are not followed by the Java program. Hence the Error handling happens in the Parsing Phase of the project

IV. TRANSLATION SYNTAX DEFINITION

The Java classes are written for the grammar and also semantic action codes are included in the CUP specification. Wherever the generated parser reduces by rule, it will execute the corresponding semantic action fragment. The semantic actions are written only for the production that is being used in C. Now consider some examples, the production modifier which have list of modifiers like public, protected, private...etc, since the target code (C) does not have definition for modifiers in the language specification, the production modifier in the CUP specification does not have semantic action for it. Similarly consider the production package declaration, since package statements are not used C, the semantic action code for production package declaration is also not defined and the class declaration in Java is mapped to Struct in C. So the semantic actions are written after the comparative study on Java and C. An Abstract syntax does semantic check of the grammar.

For Example:

The import statement in cup specification is

```
import_declaration ::=
single_type_import_declaration
|
    type_import_on_demand_declaration
|
    static_single_type_import_declaration
|
    static_type_import_on_demand_declaration
;
single_type_import_declaration ::=
IMPORT name SEMICOLON
;
static_single_type_import_declaration ::=
IMPORT STATIC name SEMICOLON
;
type_import_on_demand_declaration ::=
IMPORT name DOT MULT SEMICOLON
;
static_type_import_on_demand_declaration ::=
IMPORT STATIC name DOT MULT SEMICOLON
;
```

Here, *IMPORT STATIC name SEMICOLON* and *IMPORT name SEMICOLON* is mapped as

"#include<" + name + ">";and

IMPORT name DOT MULT SEMICOLON and *IMPORT STATIC name DOT MULT SEMICOLON* as return
"#include<" + name + ".*.h>";

V. TARGET CODE GENERATION

Since the target is an algorithmic language, the syntax can directly be mapped to the target syntax. Our approach works not only for syntactically compatible language pairs but to the other category as well, where the source and target are syntactically non compatible. The generation can be carefully controlled for the most basic syntax elements, so as to get the perfect translation to the target language. The higher syntax structures such as classes and blocks simply delegates their translation to its constituent syntax elements eventually reaching the basic

constructs such as keywords, operators, identifiers and their ordering in the target.

This kind of explosion can even deal with nested and semi nested type syntax constructs such the virtual method overriding of java. The methods of certain classes can be individually overridden for each of its objects, its sub objects and so on. Although it is difficult to capture far relation between syntax constructs, we will be able to cater to all CFG based constructs. This is adequate for our task at hand for all programming languages, since they will never have more complex constructs.

A few cases that we had to deal with the generation include templates and generics in Java and C++ pair. Consider the template class construct in C++. This has an explicit 'template' key word and followed by a list of arguments both type and value, enclosed in angle brackets ($\langle \rangle$), e.g. `template<class T> class A{....}`. Where as in Java the same generic class is constructed with simple placeholders in angle brackets, e.g. `class A<T> {....}`. Although the Java to CPP translation is predominantly lossy in nature, this is an exceptional case where we had to insert the template key word and rearrange the later, to a more complex form.

VI. CONCLUSION AND FUTURE WORKS

As stated, J2CC compiler is flexible and is efficient. Since the grammar specification and the abstract syntax classes are written for the entire Java specification, the compiler can be generated for other languages too like pearl, Pascal, C++...etc. We just want to change representation in the specification for the desired target code language. It takes the time complexity for the LALR grammar. Our future works are Error handling, Garbage collection, Templates.

REFERENCES

- [1] Andrew W.Appel,"Modern Compiler Implementation in Java"Revised Ed. Cambridge University Press India Pvt.Ltd, New Delhi, 2010.
- [2] CUPS [online].Available: <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
- [3] CodeChef [online].Available:www.codechef.com
- [4] CodeChef [online].Available:www.codechef.com/problems/H1
- [5] Dejan Jelovic."Why Java Will Always Be Slower than C++"unpublished.
- [6] Ankush Varma and Shuvra S. Bhattacharyya, "Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems", Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems.
- [7] Anders Nilsson, Torbjorn Ekman and Klas Nilsson, "Real Java for Real Time – Gain and Pain", CASES 2002, October 8–11, 2002, Grenoble, France.
- [8] Robert C. Martin."Java C++ A critical comparison. "unpublished.