Teaching Compiler Development to Undergraduates Using a Template Based Approach

Md. Zahurul Islam and Mumit Khan
Department of Computer Science and Engineering, BRAC University, Dhaka, Bangladesh.
zahurul@bracuniversity.ac.bd, mumit@bracuniversity.ac.bd.

Abstract

Compiler Design remains one of the most dreaded courses in any undergraduate Computer Science curriculum, due in part to the complexity and the breadth of the material covered in a typical 14-15 week semester time frame. The situation is further complicated by the fact that most undergraduates have never implemented a large enough software package that is needed for a working compiler, and to do so in such a short time span is a challenge indeed. This necessitates changes in the way we teach compilers, and specifically in ways we set up the project for the Compiler Design course at the undergraduate level. We describe a template based method for teaching compiler design and implementation to the undergraduates, where the students fill in the blanks in a set of templates for each phase of the compiler, starting from the lexical scanner to the code generator. Compilers for new languages can be implemented by modifying only the parts necessary to implement the syntax and the semantics of the language, leaving much of the remaining environment as is. The students not only learn how to design the various phases of the compiler, but also learn the software design and engineering techniques for implementing large software systems. In this paper, we describe a compiler teaching methodology that implements a full working compiler for an imperative C-like programming language with backend code generators for MIPS, Java Virtual Machine (JVM) and Microsoft's .NET Common Language Runtime (CLR).

Keywords: code generator, compiler design, lexical analysis, optimization, semantic analysis, syntax, analysis.

I. INTRODUCTION

A typical undergraduate is likely to experience large and complex software design and implementation challenge for the first time in the course on Compiler Design. This core course in the undergraduate Computer Science curriculum is dreaded by the undergraduate students for both its depth and breadth of the material covered, from theoretical computer science to the very practical issue of how to design large modular software in a very short time frame of a semester or even a quarter [1]. The lessons learned however is highly dependent on the teaching methodology used as well as on the software tools used to implement the compiler during the course. Our experience shows that the student is more likely to absorb and retain the knowledge of both the theoretical underpinnings of programming languages and the practical software engineering methodologies used to implement a compiler for these languages if they are taught using an incremental template based method, in which the students "fill in" the various templates used to implement the various phases in the compiler instead of starting from scratch. In this paper, we describe our experience in developing a pedagogical software platform for teaching compiler development to the undergraduate students using a subset of the C programming language, which generates code for three different architectures – MIPS, JVM and CLR. The instructor is of course free to choose any other suitable language – a scaled-down version of a common programming language such as Java or C for example – and create the templates needed to develop a compiler for that language. Our approach is quite traditional in that we make extensive use of automated tools such as scanner and parser generators to develop a compiler for a simple but usable language, which reduces the programming load without oversimplifying the underlying complexity of compiler design [2-5].

There are a set of initial decisions that the instructor must make before the project is underway:

1. A suitable language to develop the compiler for. We have chosen a subset of the C programming language modeled after the C- language described in [6]. We could have just as easily chosen Appel's MiniJava [5] or Aiken's COOL [2], which are both designed for classroom compiler development projects. Whatever language is chosen, it is imperative that the abstract syntax tree (AST) is well-defined before the beginning of semantic analysis, especially if the *Visitor Pattern* is used to implement the later phases.
2. The implementation language to develop the compiler in. We have chosen Java, although we have developed the framework in both Java and C++, so both are likely candidates. In fact, the instructor may give the students a choice, within reason, of implementation languages.
3. Automated tools. We have chosen JLex [7] and CUP [8] as the scanner and parser generator respectively (Lex and YACC when using C++). It is important to note that there are a large number of choices available, each with its own set of constraints.
4. The phases of a compiler to develop. A typical choice may be to limit the optimization phase to simple local optimizations, or to eliminate the phase altogether. If optimization is included, a suitable intermediate representation (IR) is very important. We have chosen LLVM, which is an SSA-based, low-level, strongly-typed IR designed to support efficient global optimization and high-level analysis [9].

5. The target platform. It is useful to pick a reasonable target to generate code for, especially one for which debugging and simulator tools are readily available. Our typical choices include MIPS, JVM and CLR [10-12].
6. Implementing the project. The course staff must implement the project fully before its use in the course. That is the only way to ensure that the project is appropriate for the course. This also allows the course staff to create the testing infrastructure that the students will use.

In the following sections, we provide an overview of the C- language, and then proceed onto the various phases of the compiler front- and back-end.

II. OVERVIEW OF THE C- LANGUAGE

The C- programming language is basically a small tractable subset of C without pointers and aggregates [5]. We have extended the language to include new data types such as *bool* and *string*. A C- program to compute the greatest common divisor (GCD) of two numbers is shown in Fig. 1. We provide an overview of the syntax and semantics of the language below.

```
// Program to compute GCD using
// Euclid's algorithm.
int gcd (int u, int v)
{
  if (v == 0) return u;
  else return gcd (v,u-u/v*v);
}
void main (void)
{
  int x; int y;
  write_int(gcd(x,y));
}
```

Fig. 1 C- program to compute GCD using Euclid's algorithm.

A. Lexical Convention

Table I shows the keywords, operators, and the special symbols in the C- language.

Table I C- language lexical items

Keywords	int, bool, string, void, true, false, if, else, while, return.
Operators	+ - * / < <= > >= == != = - (unary) !
Special symbols	; , ( ) [ ] { } //
Other tokens	ID, INT_LITERAL, STR_LITERAL

An Identifier in C- language consists of a letter, followed by zero or more letters, digits, or underscores and these are case sensitive. An INT_LITERAL is an integer literal (a digit followed by zero or more digits). A STR_LITERAL is character string literal, surrounded by double quotes. White space consists of blanks, new lines, and tabs. White space must separate IDs, INT_LITERALs, STR_LITERALs and keywords. Comments start with //, and extend to the end of the line. The following regular expressions describe the ID, INT_LITERAL, and STR_LITERAL tokens:

letter = [a-zA-Z]
digit = [0-9]
ID = letter (letter | digit | '_')*
INT_LITERAL = digit digit* = digit+
STR_LITERAL = '"' [^"]* '"'

B. Grammar

A C- program contains list of declaration, statements, and expression. Each function declaration has a return type, name (identifier) list of parameter and compound statement. A compound statement may have some local declaration and list of statement. The C- language support various types of statements such as compound statement, assignment statement, selection statement, iteration statement, call statement and return statement. The grammar of C- language is given in Table II

Table II C- language grammar

1. program → declaration-list
2. declaration-list →declaration-list declaration | ε
3. declaration → var-declaration | fun-declaration
4. var-declaration → type-specifier ID ;
   | type-specifier ID [INT_LITERAL] ;
5. type-specifier → int | bool | string | void
6. fun-declaration → type-specifier ID (params) compound-stmt
7. params → param-list | void | ε
8. param-list → param-list , param | param
9. param → type-specifier ID | type-specifier ID []
10. compound-stmt → { local-declarations statement-list }
11. local-declarations → local-declarations var-declaration | ε
12. statement-list → statement-list statement | ε
13. statement → compound- stmt | assign-stmt
    | selection-stmt | iteration-stmt | call-stmt
    | return-stmt
14. selection-stmt → if (expression) statement
    | if (expression) statement else statement
15. iteration-stmt → while (expression) statement
16. return-stmt → return ; | return expression ;
17. call-stmt → call ;
18. assign-stmt → var = expression ;

```
19. var → ID | ID [ expression ]
20. expression → expression mulop additive-expression
        | expression addop additive-expression
        | expression relop additive-expression
        | ( expression )
        | var
        | call
        | INT_LITERAL
        | STR_LITERAL
        | true
        | false
21. relop → <= | < | > | >= | == | !=
22. addop → + | -
23. mulop → * | /
24. call → ID ( args )
25. args → arg-list | ε
26. arg-list → arg-list , expression | expression
```

A sample C- program that calculates greatest common divisor is shown in Fig. 1.

III. THE COMPILER FRONT-END PHASES

The first phase of the compiler is the lexical analysis phase, which reads the input program to be compiled, and breaks it up into a sequenced of tokens. These tokens are in turn used by the next phase of the compiler, the syntax analyzer, to check for grammaticality of the input program. We use JLex [7] to automatically create a C- language scanner given its token definition. It is instructive to study the NFA and DFA tables produced by the automated scanner. The students are provided with a template that describes some of the tokens in the language, and are required to fill in the rest to create a complete scanner. The test input programs are provided to check for conformance. The students should learn to write hand-crafted scanners, but that can be limited to a highly simplified language.

The next phase of the compiler reads in the sequence of tokens returned by the scanner, and performs a syntactic check of the input program against the specified grammar. The output of this phase is an abstract syntax tree (AST) of the C- language program. We use Construction of Useful Parsers (CUP) [8] to create the parser for the C- language. The students are advised to study the LALR states and transitions produced by the generated parser. The critical design decisions in this phase are to (a) create the grammar description suitable for the parser algorithm, and to (b) create the AST type hierarchy that describes the language to be compiled in reasonable detail. We have in the past chosen to split these two decisions and have implemented these in two sub phases. The design of the AST type hierarchy is critical in the later phases, and must be designed with forethought. Fig. 2 shows the AST type hierarchy used in the most recent course on compiler development that used the C- language described here. Note that while there is no one "correct" AST type hierarchy, there are many "wrong" ones, and the instructor needs to spend some time detailing the various design decisions that go into designing such a hierarchy so that the students gain some insight into the process. It is also important to design a stable AST type hierarchy right at the beginning if one should choose to use the *Visitor Pattern* [13] to implement the later phases of the compiler. Our later phases of the compiler are all designed using the Visitor Pattern, greatly simplifying the implementation. For the first sub phase, the instructor sets up the skeleton subset of the language grammar, and the students must complete the grammar description and create a complete parser that emits error messages if there are syntax errors. The second sub phase is more complex, as it requires the building of the AST given the AST type hierarchy. The template CUP description includes examples of how some of the top-level AST nodes are created, as well as some of the leaf nodes (e.g., *IdAST*, *IntLiteralAST*, etc), and the students fill in the rest after studying the AST type hierarchy in detail. One of the useful techniques to verify the correctness of the AST creation is to visualize the AST as a graph. Our students create a *DotPrinter* visitor to output the AST in the GraphViz dot format [14], which can then be visualized using the dotty program. Another useful assignment is to write a *PrettyPrinter* visitor, which traverses the AST to create a formatted version of the original source code.

Once the AST is created by the syntax analysis phase, the semantic phase proceeds to check the static semantics of the input program. The two main tasks for the semantic analysis phase are to (a) create the name-to-object bindings for the input program, checking for declaration errors, and to (b) check the static type correctness. Both of these are implemented using the Visitor Pattern.

The name-checker visitor traverses the AST and creates name-to-object bindings using a symbol table, appropriately scoped. The design of the symbol table is an important design decision in industrial strength compilers, but for the purposes of the project can be a simple Java *HashMap* that maintains the binding for each scope. The students should however be made aware of the various techniques used to create and maintain a symbol table. The visitor that implements this function annotates the AST with binding information that is later used by the semantic type-checker visitor to check for type correctness in the input program. The template sets up the *NameChecker* visitor that traverses the AST and checks the names using a symbol table, which is local to the visitor. The visitor annotates the AST with the name binding information, and discards the symbol table.

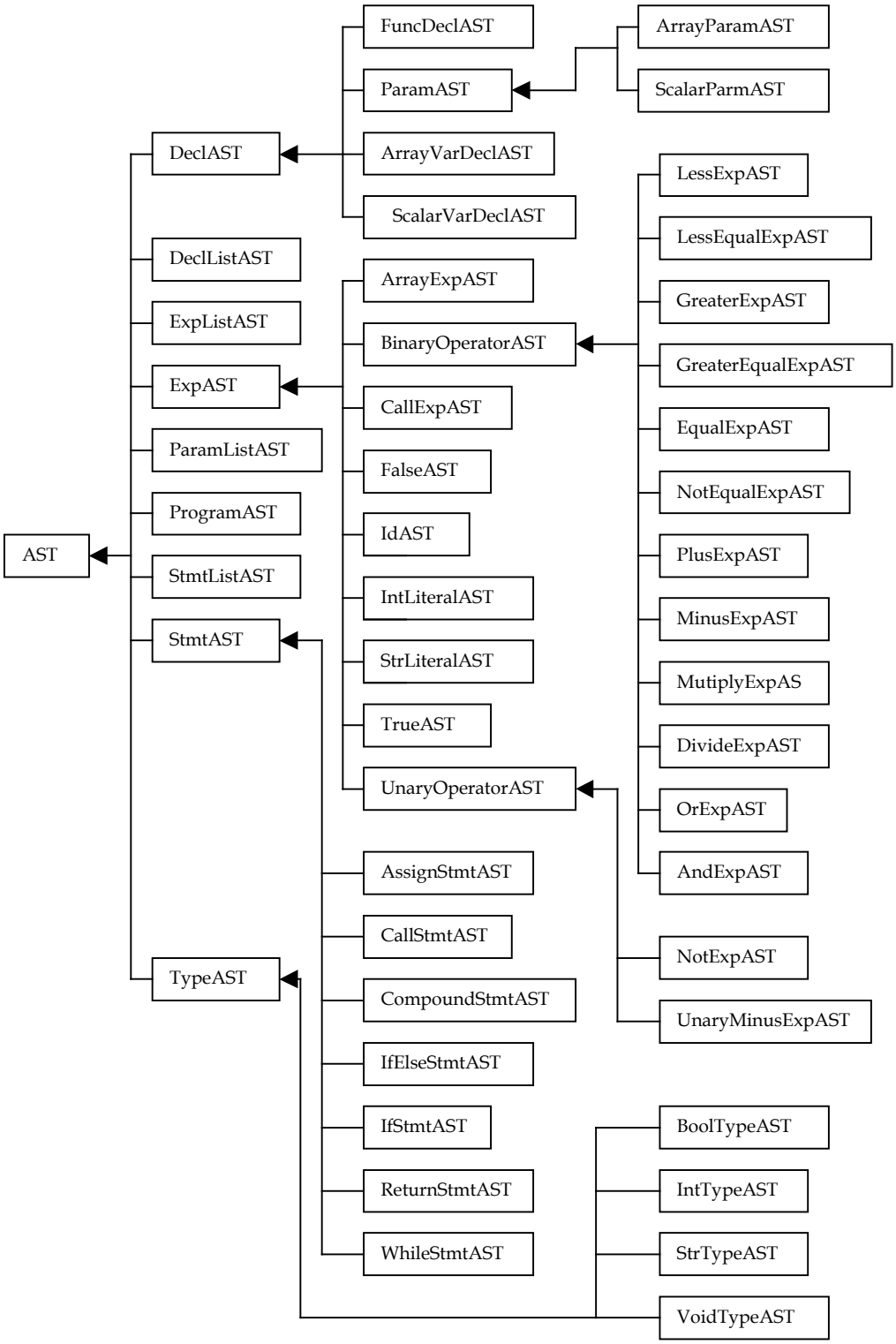

Figure 2 The C- language AST type hierarchy

The type-checker visitor further annotates the AST by adding type information for all the identifiers in the input program, while checking for type correctness. While type checking is a non-trivial task for most programming languages, the static type semantics of the C- language is simple enough that this is quite reasonable to implement in a short time. The following describes the type semantics for a function application in C- for example:

E → E1 (E2) {E.type := if E2.type = s and
E1.type = S → t then t
Else type_error}

This rule says that in an expression formed by applying E1 to E2, the type of E1 must be a function s → t from the type s of E2 to some range type t. The type of E1 (E2) is t. To check of a C- program we also have to

check the type of expressions and statements. Since some C- language constructs like statements do not have values, the special type *non_type* is assigned to those. If an error is detected within a statement, the type assigned is *type_error*. The C- type system is described by a Java or C++ class hierarchy rooted at *Type*. C-language static type semantics does not have subtyping, and so the implementation does not have to worry about type unification, greatly simplifying the design and implementation. The template sets up the *TypeChecker* visitor that traverses the AST and checks type correctness in the input program, using the name binding annotations in the AST. The output of this visitor is to annotate the AST further with type information, so that each node has a type.

IV. CODE GENERATION

The code generation phase in the compiler back-end takes as input a higher-level intermediate representation such as AST or some other IR such as LLVM IR of the source program and produces as output an equivalent target program. The target program code must be correct and high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently. The C-Compiler generates codes for the multiple targets machine such as Java Virtual Machine (JVM), Microsoft Common Language Runtime (CLR), and MIPS architecture. The MIPS code generator is used as the baseline code generator as it is trivial to map the C- AST to MIPS instruction using a stack-machine. To generate code for the JVM we translate the C- program into an intermediate language called Oolong [15]. Similarly, to generate code for the CLR we translate the C- program into an intermediate language called Microsoft Intermediate Language (MSIL) [16]. The templates that are created for this phase are one visitor per target, with some of the code generation primitives fill in to get the students started.

A. MIPS Code Generation

As mentioned earlier, we use MIPS as the baseline target platform, because of its orthogonal instruction set and the availability of the SPIM simulator that can be used to test the target code on a variety of platforms [17]. The MIPS baseline code generator generates code for a pure stack machine, using eight registers shown in Table III. The calling convention and the activation frame layout change from semester to semester to illustrate design trade-offs. Only a few optimizations are performed, such as eliminating consecutive push and pop operations involving the same register and memory location tuple.

Table III MIPS registers used in code generation

Register	Explanation
$fp	Frame pointer
$sp	Stack pointer
$ra	Return address register
$ v0	Function/expression result
$v1	Function/expression result
$a0	Function argument
$t0	Temporary register
$t1	Temporary register

B. Oolong Code Generation for JVM

Our JVM code generator produces an intermediate representation called Oolong [15] directly from the AST. Oolong is an assembly language for the Java virtual machine, based on the Jasmin language by Jon Meyer [18]. The Oolong representation is functionally equivalent to the Java binary *class* format, but far easier to read and write. The JVM is divided into four conceptual data spaces these are: Class area, Java Stack, Heap, and Native Method Stack. Since our code generators all generate code for a stack machine, our primary focus is the Java Stack. Each time a method is invoked, a new data space called a stack frame is created. Collectively, the stack frames are called the Java stack. The stack frame on top of the stack is called the active stack frame. Each stack frame has an operand stack, an array of local variables, and a pointer to the currently executing instruction. This instruction pointer is called the program counter (PC). The program counter points into the method area, and points at the current instruction.

The JVM was designed with Java in mind, with support for classes. We generate a class with a name based on the C- file name and all the C- code is translated to Oolong code under that class. To perform any operation, we push the operands onto the stack then perform the operation. To generate the byte-code from the Oolong code, we assemble the Oolong code by the Oolong assembler, which is freely available on the web. It should be noted that some of the Oolong instructions are data type dependent.

C. MSIL CODE GENERATION FOR CLR

The .NET Common Language Runtime (CLR) [12] is designed to be a language-neutral architecture. The CLR differs from the JVM in this one respect. There are many similarities as well. CLR consists of a typed, stack-based intermediate language (IL), an Execution Engine (EE) that executes IL and provides a variety of runtime services (storage management, debugging, profiling, security, etc.), and a set of shared libraries (.NET Frameworks) [16]. The CLR has been successfully targeted by a variety of source languages, including C#, Visual Basic, C++, Eiffel, Cobol, Standard ML, Mercury, Scheme and Haskell.

To generate Byte-code for CLR we use Microsoft In-

termediate Language (MSIL) as the intermediate language. There is an MSIL intermediate language assembler available with the Microsoft .NET framework. The Intermediate Language assembler assembles the MSIL code to portable executable code, which is the Byte-code for the Common Language Runtime. The CLR has an evaluation stack, an array of local variable, and an array of incoming arguments. The evaluation stack is like the JVM operand stack. To perform any operation we need to push the operands onto the evaluation stack then perform the operation. The array of local variable holds the local variables of a function. To use the local variable array in CLR we need to first allocate the space. The array of incoming arguments contains the function parameter when a function is invoked. The MSIL instruction does not depend on the data type like Oolong instruction.

V. CODE IMPROVEMENT

Code improvement is often referred to as optimization, though it seldom makes anything optimal in any absolute sense. It is an optional phase of compilation whose goal is to transform a program into a new version that computes the same result more efficiently, more quickly or using less memory or both (see [19] for an extensive discussion on code improvement). Some improvements are machine independent, and can be performed as transformations on the intermediate form such as AST or low-level IR; other improvements require an understanding of target machines, and must be performed as transformations on the target program. Thus code improvement often appears as two additional phases of compilation, one immediately after semantic analysis and intermediate code generation, the other immediately after target code generation.

We have limited the code improvement to machine independent local optimizations only. Target dependent and Global optimizations, while somewhat covered in the theory lessons, are simply too complex to handle in a single semester course on compiler development. We have chosen to use the SSA-based LLVM IR, which makes it ideal for most of the trivial local optimization tasks. The students are required to first create the Basic Blocks (BB), followed by the Control Flow Graph (CFG), and then apply some of the simple local optimizations such as constant folding, copy propagation, common sub-expression elimination, peephole optimization, etc, all within a single basic block. There are a set of templates for the code improvement phase, the first of which deals with creation of the CFGs. Once these are created, the students have to write a *CFGDotPrinter* to create the graph that can be used to visualize the CFG using GraphViz [14]. The other templates set up the basic optimization tasks, allowing students to incrementally add various code improvements by studying some of the existing ones. Fig. 3 shows an excerpt of the graph for the *IfElseStmtAST* node, created from the GCD program in Fig. 1.

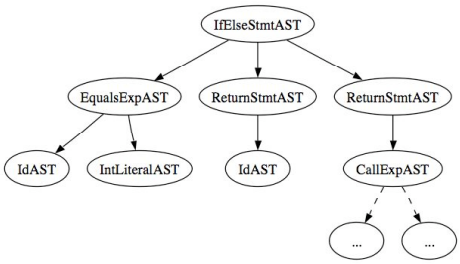

Figure 3 AST graph visualization using GraphViz

VI. CONCLUSION

A good course in compiler development is not only beneficial to the student's understanding of theoretical computer science, it also greatly enhances a student's grasp of the practical area of software engineering. A compiler project is likely to be the most complex software engineering task many students complete in an undergraduate program. Our experience shows that a large factor in how much the students learn from this course depends on how the course project is structured. Aiken points out the importance of structuring the project carefully, and implementing the entire project by the course staff before its use in the course, "because a full implementation is the only reliable way to ensure that the project is self-consistent, complete, and tractable" [2]. A template based method, where the students create a fully working compiler from a set of templates created by the instructor, and using well-documented automated tools to create scanners and parsers, has been shown to be very effective. It also helps if the students manage to create a compiler that emits runnable code, providing a sense of great accomplishment.

REFERENCES

[1] ACM Computing Curricula 2001 - Appendix A: CS Body of Knowledge. http://www.computer.org/education/cc2001/final/pl.htm

[2] Alexander Aiken. *Cool: a portable project for teaching compiler construction*. ACM SIGPLAN Notices, Vol. 31, No. 1, July 1996.

[3] Bill Appelbe. *Teaching Compiler Development*. Proceedings of the tenth SIGCSE technical symposium on Computer science education, pp. 23-27, January 1979.

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers Principles, Techniques, and tools*. Addison-Wesley, Boston, 1986.

[5] A. W. Appel. *Modern Compiler Implementation in Java*. Second edition, Cambridge University Press, 2002.

[6] Kenneth C. Louden. *Compiler Construction Principles and Practice*. PWS Publishing Company, 1997.

[7] JLex: A Lexical Analyzer Generator for Java™. www.cs.princeton.edu/~appel/modern/java/JLex/

[8] CUP Parser Generator for Java™. www.cs.princeton.edu/~appel/modern/java/CUP/

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

[10] John L. Hennessy, David A. Pattterson. *Computer Organization and Design*. Hardcourt (India) Private Limited & Morgan Kaufmann, 1999.

[11] Frank Yellin, Tim Lindholm. *The Java™ Virtual Machine Specification*. Sun Microsystems, Inc., 1999.

[12] Eric Meijer, John Gough. *Technical Overview of the Common Language Runtime*. MIT Press, 2001.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, October 1994.

[14] Emden R. Gansner and Stephen C. North. *An open graph visualization system and its applications to software engineering*. Software – Practice and Experience, Vol. 30, No. 11, pp. 1203-1233, 2000.

[15] Joshua Engel. *Programming for Java™ Virtual Machine*. Addison Wesley, 1999.

[16] Andrew Kennedy, Don Syme. *Design and Implementation of Generic for the .NET Common Language Runtime*. Microsoft Corporation, 2000.

[17] SPIM: A MIPS32 Simulator. http://www.cs.wisc.edu/~larus/spim.html

[18] Jasmin: A Java assembler for the Java Virtual machine. http://jasmin.sourceforge.net/

[19] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.