Modern Compiler Design: An approach to make Compiler Design a Significant Study for Students

Namit Bhati
Assistant Professor, JNU Jaipur
namit.gbu@gmail.com

----------ABSTRACT----------

Compiler Design is a common subject of most modern Computer Science undergraduate curriculum. However, compiler design has become a highly specialized topic, and it is not clear that a significant number of Computer Science students will find themselves designing compilers professionally. This paper is a thorough introduction to compiler design, focusing on more low-level and systems aspects rather than high-level questions such as polymorphic type inference or separate compilation. You will be building several complete end-to-end compilers for successively more complex languages. Designing the content of compiler design courses to emphasize this broad applicability can make them more relevant to students.

Keywords: Lex, Yacc, Lexeme, Semantically, Lexical, Syntactically, Intermediate codes, yy.c

1. Introduction

There are many examples of such translators—discussed later in this paper-that fall outside the traditional model of compilers; a lot of them don't involve programming languages at all. In each of these cases, however, the translation process has roughly the same structure: an input string is decomposed into tokens; the token sequence is grouped into "phrases" whose structure is specified by (something akin to) a context-free grammar; and these phrases are finally mapped to the output sequence in a manner determined by their structure and the context in which they occur. Many of the issues that arise, including the ways in which the input can be organized into tokens and phrases and the ways in which such phrases can be represented and manipulated, are very similar across all of these examples. Focusing on these commonalities makes it possible to present many traditional compiler techniques, e.g., buffer management for lexical analysis, parsing techniques for context-free languages, and attribute evaluation and propagation in parse trees, in a much more general setting that emphasizes their relevance to a significantly wider range of applications. It also shows how compiler development tools such as lex and yacc can be applied for many translation problems that students do not typically see as compilation problems.

1.1 Why Study Compilers?

Everything that computers do is the result of some program, and all of the millions of programs in the world are written in one of the many thousands of programming languages that have been developed over the last 60 years. Designing and implementing a programming language turns out to be difficult; some of the best minds in computer science have thought about the problems involved and contributed beautiful and deep results [1,2]. Learning something about compilers will show you the interplay of theory and practice in computer science, especially how powerful general ideas combined with engineering insight can lead to practical solutions to very hard problems. Knowing how a compiler works will also make you a better programmer and increase your ability to learn new programming languages quickly.

2. Compiler Design: Overview

You will also understand some specific components of compiler technology, such as lexical analysis, grammars and parsing, type-checking, intermediate representations, static analysis, common optimizations, instruction selection, register allocation, code generation, and runtime organization. The knowledge gained should be broad enough that if you are confronted with the task of contributing to the implementation of a real compiler in the field, you should be able to do so confidently and quickly [3]. For many of you, this will be the first time you have to write, maintain, and evolve a complex piece of software. You will have to program for correctness, while keeping an eye on efficiency, both for the compiler itself and for the code it generates. Because you will have to rewrite the compiler from lab to lab, and also because you will be collaborating with a partner, you will have to pay close attention to issues of modularity and interfaces. Developing these software engineering and system building skills are an important goal of this class, although we will rarely talk about them explicitly.

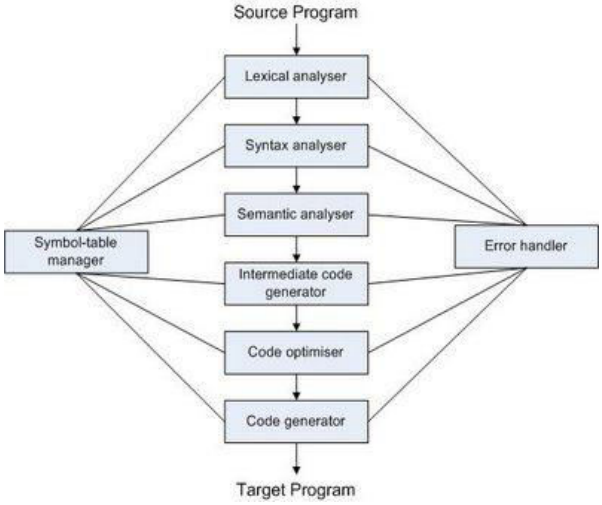

Fig: 2.1 structure of a compiler [1]

3. Phases of a Compiler

The execution of a compiler conceptually consists of four phases:

1. **Lexical Analysis**
2. **Syntax Analysis or Parsing**
3. **Semantic Analysis**
4. **Code Generation**
5. **Code Optimization**

In this section discusses each such phase with regard to how its ideas, concepts, and techniques can be useful in translation problems outside the realm of traditional compilation.

3.1 Lexical Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well [4]. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding these interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNext Token command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

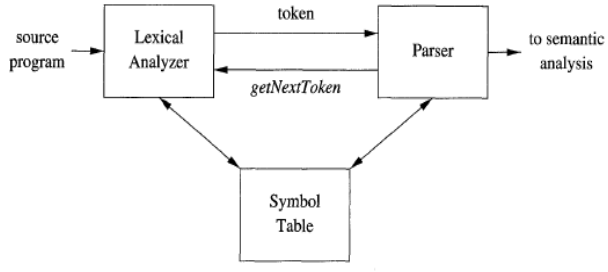

Fig: 3.1 overview of lexical Analysis [5]

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.

One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

TOKENS, PATTERNS, AND LEXEMES:

When discussing lexical analysis, we use three related but distinct terms:

Token: A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name [4, 5].

Pattern: A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings [4, 5].

Lexeme: A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

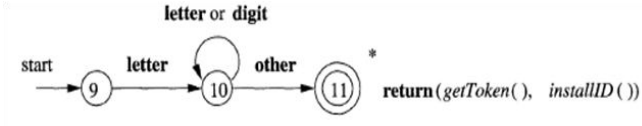

Fig: 3.2 a transition diagram for id's and keywords

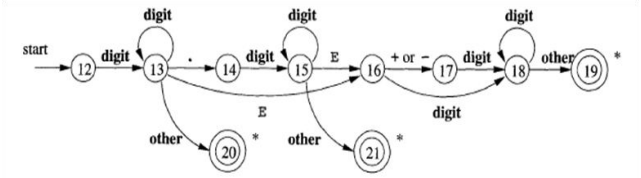

Fig: 3.3 the transition diagram for a token number is given

- If a dot is seen we have an optional fraction.
- State 14 is entered and we look for one or more additional digits.
- State 15 is used for this purpose.
- If we see an E, we have an optional exponent, states 16 through 19 are used to recognize the exponent value.
- In the state 15, if we see anything other than E or digit, then 21 is the end of the accepting state.

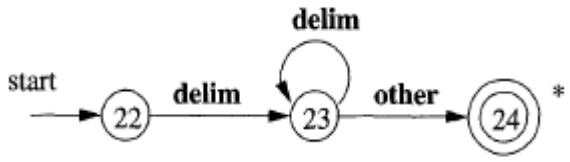

Fig: 3.4 a transition diagram for whitespace is given below

In the diagram we look for one or more whitespace characters represented by delim in the diagram –typically these characters would be blank, tab, newline.

Use of Lex: An input file lex1 is written in the lex language and describes the lexical analyzer to be generated. The Lex compiler transforms lex1 to a c program in a file that is always named lex.yy.c

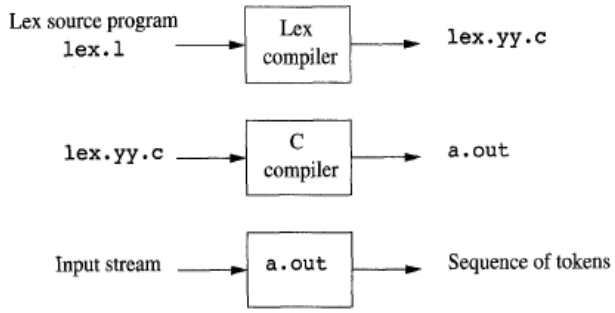

Fig: 3.5 Use of Lex

LEX PROGRAM FOR TOKEN:

```
%{
/* definitions of manifest constants
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim [ \t\nl
ws (delim)+
letter [A-Za-z]
digit [o-9]
id {letter} {(letter) | {digit})*
number {digit)+ (\ . {digit}+)? (E [+-] ?{digit}+)?

%%
{ws} (/* no action and no return */)
if      {return(IF) ; }
then  {return(THEN) ; }
else  {return(ELSE) ; }
{id}            {yylval = (int) installID(); return(ID);}
{number}     {yylval = (int) installNum() ;
                return(NUMBER) ; }
"<"             {yylval = LT; return(REL0P); }
"<="            {yylval = LE; return(REL0P); }
"="             {yylval = EQ ; return(REL0P); }
"<>"            {yylval = NE; return(REL0P);}
">"             {yylval = GT; return(REL0P);}
">="            {yylval = GE; return(REL0P);}
 %%
```

```
int installID0 {/* function to install the lexeme, whose
                   first character is pointed to by yytext,
                   and whose length is yyleng, into the
                   symbol table and return a pointer
                   thereto */
}
int installNum() {/* similar to installID, but puts
numerical constants into a separate table */

}
```

3.2 SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree [1].

Advantages of grammar for syntactic specification :

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

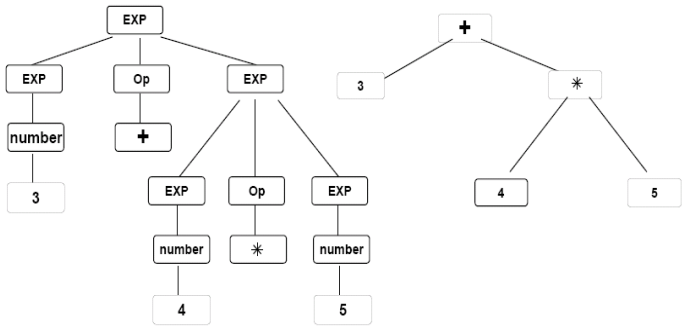

Fig: 3.6 Example for Syntax analysis

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program[7,8]. It also recovers from commonly occurring errors so that it can continue processing its input.

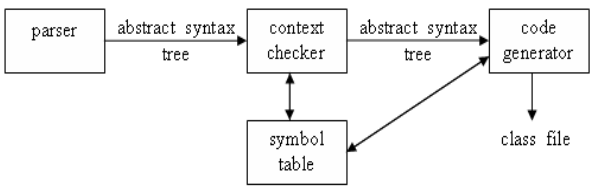

Fig: 3.7 Parser

Functions of the parser:
1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues Of parser:
1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling:
Programs can contain errors at many different levels. For example:
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Example:
Given grammar G : E → E+E | E*E | (E) | - E | id
Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

Strings that appear in leftmost derivation are called left sentinel forms. Strings that appear in rightmost derivation are called right sentinel forms.

3.3 Semantic Analysis
Semantic analysis refers to the computation and propagation of information that is not part of the context-free syntax of the language. In a compiler, this might refer to the type or scope of a variable. A common way of handling such information is using "attribute grammars," which associate properties ("attributes") with grammar symbols and specify rules, called semantic rules, for computing their values. These rules in effect specify the flow of information between different points in the parse tree for a program. Not surprisingly, information has to be propagated along the parse tree for many other translation problems as well [9,10]. An example that we discuss in class involves displaying HTML documents in a browser. The input in this case is an HTML document, with tags such as and that affect the way specific characters are displayed, as well as the amount of space taken by a group of characters (a boldface character is typically wider than one that is not). The output is the sequence of characters being displayed in the browser window. Among the problems to be addressed is the determination of when the line being displayed is "long enough," making it necessary to emit a line break character [1, 2, 3].

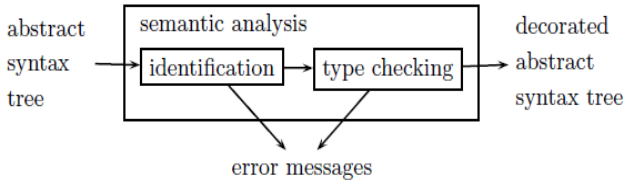

Fig: 3.8 Semantic analysis

This makes it necessary to figure out how to compute and propagate semantic information about the font in use at any particular point in the text as well as the line length in the display window up to that point. Compiler courses traditionally treat optimization in terms of code transformations that make the program run faster[6,7,8]. A more general view is that optimization aims to reduce the "cost" of the generated code for some cost measure of interest.

Traditionally, the cost measure most often used has been execution time; however, even within mainstream compiler research, other measures of cost have recently been gaining credence: these include code size (for limited-memory processors, e.g., in embedded and mobile systems) and energy usage (e.g., for battery-operated portable computers). When we generalize to other translation problems, it may still make sense to consider the "cost" of a representation. As an example, the graph drawing tool dot [1] takes a textual specification of a graph as input and produces a pictorial representation of the graph, e.g., as a JPEG or PostScript file, as output. Since a picture with many edges crossing one another is harder to understand than one with fewer edge crossings, dot tries to "optimize" the pictorial representation it produces by changing the layouts of vertices and edges so as to reduce the number of edge crossings [11,12]. Conceptually, this is exactly analogous to the optimization phase of a compiler.

3.4 Intermediate code generator
In Intermediate code generation we use syntax directed methods to translate the source program into an intermediate form programming language constructs such as declarations, assignments and flow-of-control statements.

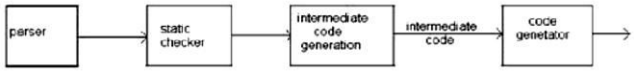

Fig: 3.9 Role of intermediate code generator

Advantages of Using an Intermediate Language
1. Build a compiler for a new machine by attaching a new code generator to an existing front-end.
2. Reuse intermediate code optimizers in compilers for different languages and different machines.

3. The terms "intermediate code", "intermediate language", and "intermediate representation" are all used interchangeably.

3.5 CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

ISSUES IN THE DESIGN OF A CODE GENERATOR

Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation[12].

Intermediate representation can be:

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free. Target program: The output of the code generator is the target program. The output may be:

1. Absolute machine language - It can be placed in a fixed memory location and can be executed immediately. front end code optimizer code generator symbol table
2. Reloadable machine language - It allows subprograms to be compiled separately.
3. Assembly language - Code generation is made easier.

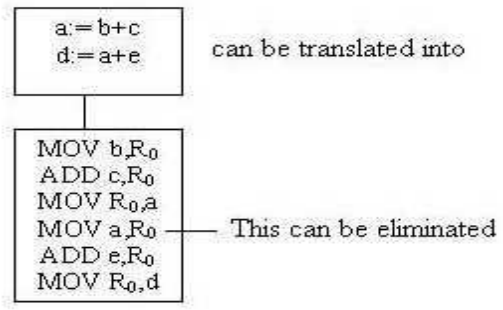

Fig: 3.10 an example for code generator

The following issues arise during the code generation phase:

a. Input to code generator
b. Target program
c. Memory management
d. Instruction selection
e. Register allocation

3.6 CODE OPTIMIZATION

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers. Optimizations are classified into two categories[13]. They are

1. Machine independent optimizations:
2. Machine dependant optimizations:

Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig.

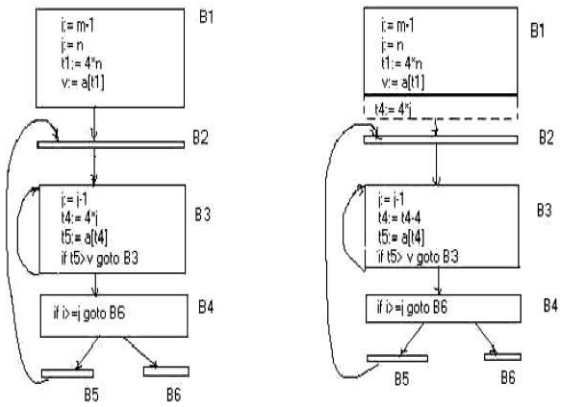

Fig: 3.11 an example for code optimization

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

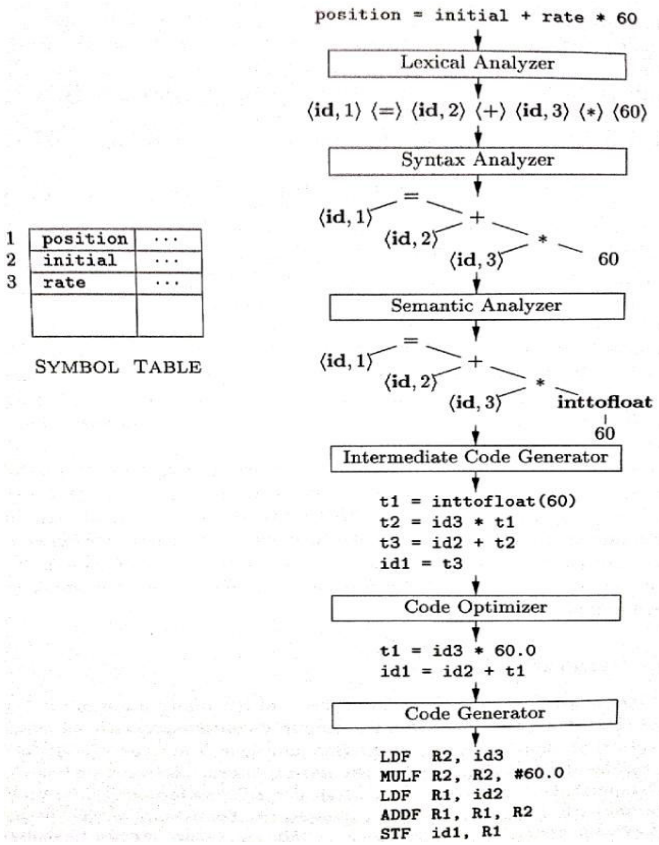

Fig: 4 example for phases of compiler

5. Conclusions

Compiler design courses typically focus narrowly on the translation of high-level programming languages into low-level assembly or machine code. Given that the majority of computer science students are unlikely you be involved in compiler design as a day-to-day professional activity, this limits the relevance of such courses to the students' eventual careers. However, it is possible to generalize the traditional view and consider the problem of translating from a source language to a target language, where both the source and target languages are defined broadly, e.g., need not even be programming languages. Such a generalized view includes many translation problems, e.g., document formatting or graph drawing that are not traditionally viewed as "compiler problems." Viewing such translation problems in this way allows us to identify and understand essential underlying commonalities of the translation process.

This has several benefits, among them that the use of tools such as lex and yacc to generate the front end of a translator reduces development time, and that by relying on well understood techniques and avoiding ad hoc approaches to the lexical analysis and parsing problems, reliability is enhanced.

Overall, therefore, students benefit from having a deeper understanding of a variety of translation problems; being able to apply techniques and tools developed for compilers to other translation problems; and thereby being able to produce better code more quickly.

References:

[1] Compilers: Principles, Techniques, and Tools Hardcover – Import, 31 Aug 2006 by Alfred V. Aho (Author), Monica S. Lam (Author), Ravi Sethi (Author), Jeffrey D. Ullman (Author)

[2] Andrew W. Appel. Modern Compiler Implementation in ML. Cambridge University Press, Cambridge, England, 1998.

[3] E. Koutsofios and S. C. North, "Drawing graphs with dot", AT&T Bell Laboratories, Murray Hill, NJ, 1993.

[4] Michael Matz, Jan Hubi˘cka, Andreas Jaeger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement. Available at http://www.x86-64.org/documentation/abi.pdf, May 2009. Draft 0.99.

[5] L. Lamport, LaTeX: A Document Preparation System,User's guide and Reference Manual. Addison-Wesley,1994.

[6] L. Maranget, "HeVeA User Documentationversion 1.06-7", INRIA, France, May 2001.http://para.inria.fr/˜maranget/hevea/doc/index.html

[7] J. S. Plank, "Jgraph – A Filter for Plotting Graphs in PostScript", Conference Proceedings, Usenix Winter 1993 Technical Conference, January 1993, pp. 63–68.

[8] R. W. Quong, "Ltoh: a customizable La- TeX to HTML converter", April 2000. http://www.best.com/˜quong/ltoh.

[9] Eyerman, S., Eeckhout, L., Karkhanis, T., Smith, J.E.: A performance counter architecture for computing accurate CPI components. In: ASPLOS. (2006) 175–184

[10] Eyerman, S., Smith, J.E., Eeckhout, L.: Characterizing the branch misprediction penalty. In: ISPASS. (2006) 48–58

[11] Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011.

[12] Principles of compiler design, 2nd ed, Nandini Prasad, Elsevier

[13] http://www.nptel.iitm.ac.in/downloads/106108052/

[14] Compiler construction, Principles and Practice, Kenneth C Louden, CENGAGE

[15] Implementations of Compiler, A new approach to Compilers including the algebraic methods, Yunlinsu, SPRINGER