International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Introduction to Compilers

Neha Pathapati[1], Niharika W. M.[2], Lakshmishree .C[3]

[1, 2, 3] M.S.Ramaiah Institute of Technology, MSR Nagar, MSRIT Post, Bangalore – 560 054, India

Abstract: ***A compiler translates the source language code into a target language code. This translation takes place through a number of phases. So, we can structure the compiler as collection or combination of many phases or passes, where each pass performs a single task. This paper outlines the various phases of the compiler and the tools which are used to construct a lexer and a parser. The actions that take place in each and every phase of the compiler are illustrated using an example. The main intention of this paper is to provide an insight into the basic knowledge of a compiler, as well as provide a springboard for a more detailed study on this topic.***

Keywords: lexer, parser, lexeme, token

1. Introduction

A programming language is a formal constructed language that is designed to communicate instructions to a machine, particularly a computer. The world as we know it is dependent on programming languages, as all the software running on all computers is written in some programing language. Computer programs are formulated in one of the programming language among numerous choices available and specify classes of computing processes. However, computers interpret sequences of particular instructions and not the program texts. Therefore, the program should be translated into an apposite instruction sequence before it can be processed by a computer. It is possible for this translation to be automated, which implies that it itself can be formulated as a program. The translation program is called a compiler.

A compiler is a special program that translates a program written in a high-level programming language (source code), suitable for human programmers into an equivalent low-level machine language (object code) that can be executed by the computer's processor. Thus, the primary reason for the conversion of the source code is to generate an executable program, into which the input can be fed to obtain the required output.

The term compilation defines the conversion of an algorithm that is expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language.

The most significant functionality of the compiler lies with its ability to apprise the programmer of the errors that it has detected in the source program during the process of translation. Figure.1. illustrates the methodology of a compiler [4].

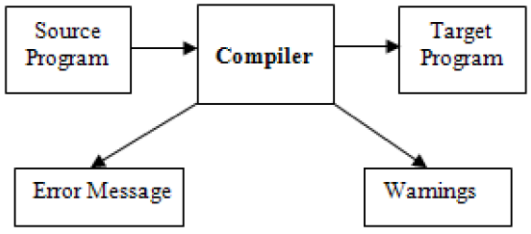

Figure 1: Methodology of a Compiler

Every language is said to have a syntactic and a semantic aspect. The syntax is responsible for prescribing which texts are grammatically accurate and the semantics specifies how to derivethe meaning from a text that is syntactically correct. A concise definition of the syntax and semantics of the source language and the target language is essential, that is, the source language and the target language should be formal. The compiler performs a reliable translation, that is, the machine language statements generated by the compiler should have the same semantic meaning as that of the source language statements. Thus, a compiler is Semantic Preserving [1].

2. Phases of a Compiler

The functioning of the compiler is organized into a series of six phases. Each phase accepts as input the intermediate form of the program that has been generated in the preceeding phase. Thus, the subsequent phases of the compiler operate on lower-code representations. The key phases include of a compiler are -
(a) Lexical Analysis
(b) Syntax Analysis
(c) Semantic Analysis
(d) Intermediate Code Generation
(e) Code Optimization
(f) Target Code Generation [2].
The phases of a compiler may be broadly categorized into the following two parts:

Analysis Part – The Front End
In the analysis part, the source program is fragmented into constituent pieces and a grammatical structure is imposed on each of them, which is then used to generate an intermediate representation of the input source program. All the syntactic and the semantic shortfalls are reported to the programmer for the appropriate corrective action to be taken. A data structure, the symbol table in which each identifier in a program's source code is associated with information relating to its declaration or appearance in the source program, such as its type, scope level and sometimes its location in the input source program is constructed and maintained. The symbol table and the intermediate representation is passed on to the Synthesis part. The analysis part can be mapped into the following phases of the compiler:
(a) Lexical analysis

(b) Syntax analysis
(c) Semantic analysis[2].

Synthesis Part – The Back End

The synthesis part takes in as input the intermediate representation and the information in the symbol table. An equivalent target program is produced. The synthesis part can be mapped to the following phases:
(a) Code Generation Phase
(b) Code Optimization Phase[2].

Each phases of the compiler which interacts a symbol table and an error handler. Hence, this is called as the analysis/synthesis model of compilation [3]. Figure.2. illustrates the phases of the compiler [4].

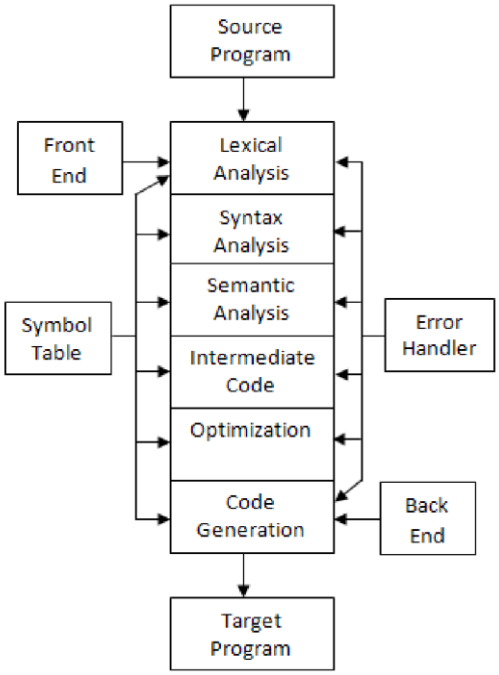

Figure 2: Phases of a Compiler

2.1 Lexical Analysis

The first phase of the compiler is termed as scanning or lexical analysis. A lexical analyzer, also known as the lexer, is a pattern recognition engine which reads a string of individual characters as its input and groups them into meaningful sequences called lexemes. For each lexeme, a token of the form,
<token-name, attribute –value>
is generated [2]. Here, the token-name refers to an abstract symbol to be used in the next phase – Syntax Analysis phase and the attribute-value is a pointer to the entry in the symbol table.
Additionally, the lexer also performs the following tasks:
(a) It discards comments and eliminates the characters (usually white-space, newlines, comments, etc) separating the tokens.
(b) Provides the tokens generated as input to the next phase of the compiler – parser or syntax analyzer.
(c) Facilitates the parser in reporting errors detected in the source program by keeping a track of the line number of the source program.
(d) The symbol table is constructed, which is useful for Semantic Analysis and Code Generation.

The primary purpose of the Lexical Analysis phase is to make the subsequent phase easier [4]. Alternatively, a lexical analyser can also be termed as a linear analyser, as it scans the input character-by-character from left to right. Some notable definitions related to this phase are :

Lex
A set of buffered input routines and constructs. It is responsible for translating regular expression into lexical analyser [4].

Token
A basic, indivisible lexical unit or language elements. 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 refer to the input symbols that the syntax analyser will process[4][2].

Lexeme
They are original string (character sequence) comprised (matched) by an instance of the token [4].

2.2 Syntax Analysis

Syntax analysis is the second phase of the compiler and is also known as parsing. The tokens generated during the lexical analysis phase of the compiler is the input of this phase which are used to create the tree like structure of the token list. In the typical structure of syntax tree, the interior node represents an operation and the children of the node represent the arguments of the operation [7].

What is important in constructing a syntax tree is to determine how the leaves are combined to form the tree structure and how the interior node is labelled. In addition to this, the parser should identify invalid texts and reject them by reporting the detected syntax errors [8].

2.2.1 Designing a Parser
To design a parser we must first define a grammar to be processed, supplement it with connection points and choose a parsing algorithm. The augmented grammar should be converted to a form that suits the chosen algorithm. After this, the actual construction of a parser can be done manually. Thus the process of parser design is a grammar design, in which we derive a grammar satisfying the rules of a particular parsing algorithm [8].

The parser accepts sequence of symbols from the symbolic table, attaches the symbols to the existing syntactic structure and outputs the modified structure. If any syntactic errors exist, the parser invokes error handler to report errors and aid recover [8]. Figure.3 shows parser information flow.

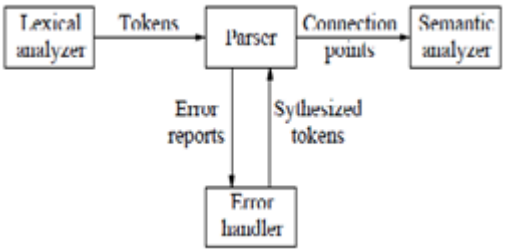

Figure 3: Parser Information Flow

The parser module provides the parse program which invokes lexical analyser's next symbol operation for each symbol. It reports each connection point by invoking an appropriate operation of other modules. This invocation is called parser action [8].

Parser looks ahead the symbols to control parsing. It reads the next symbol to supply the consecutive symbols in sequence after accepting each symbol. Using LL or LR techniques, we can determine the syntactical correctness of the program. Parser actions that enter declarations into the table or directly generate code must be introduced by the programmer. Parser actions include the symbol encoding from lexical analyser as a parameter to connection points(also called symbol connections) [8].
Selection of parsing algorithm depends on economic and implementation considerations. The algorithm should work under all circumstances. LL algorithms are best suited if no parser generator is available because the conditions are easier to verify by hand. Alternatively LR algorithms apply to large class of grammar because the production applicable is decided after reduction [8].

2.3 Semantic Analyser

Semantic analyser checks the following from the parse tree entries:
(a) Data type of first operand (num1)
(b) Data type of second operand (num2)
(c) Checks if + operator is unary or binary
(d) Checks the number of operands supplied to the operator depending on its type (unary or binary)
(e) Data type of identifier sum
(f) Checks if the data type of both LHS and RHS match in the assignment statement[10]

2.3.1 Type Checking

The process of checking that each operation satisfies the type system of the language is called type checking. That is, all operands in an expression should be of appropriate type and number. The rules for such operations are defined in other parts of code like function prototypes and sometimes they are part of language definition. Type checking can be done during compilation or execution or both. If each and every type error is recognized during compilation, then the language is considered to be strongly typed [9].

Type checking is classified into two types, static and dynamic.
The type check during compilation is called static. This type checker takes information from the declarations and stores in a master symbol table. The type involved in the operations is checked against the entries in the symbol table. A language that involves only static type checking cannot be considered as strongly typed because of the drawbacks of static checking. During compilation, many type errors might get away from the notice of type checker. For example, consider an expression a*b where a and b are assigned values outside the range of int type or computing the ratio between two integers which may result in divide by zero [9].

Implementing the type information for data location during run time is called dynamic type checking. A variable of a type will include the value as well as a tag indicating the type. An operation is performed only after checking the type tags and if the type of the operands is compatible. To maintain the type integrity, the runtime type system takes over during execution of program. Dynamic type checking detects errors which are not identified at the compile time [9].

Some compilers like C, C++ provide implicit type conversion or coercion. For example, when addition is carried out between an integer and floating point values, the integer value is implicitly promoted to float type. If at any time a type error occurs in C, a compiled code is inserted to fix the error. If no conversion is possible then type error occurs [9].

2.4 Intermediate code Generation

Intermediate representation of the source program is created in this phase. Intermediate code is the structure on which translators both built-in and user created operates [12].
Properties of intermediate code:
(a) It should be simple and well defined.
(b) It should have unambiguous semantic structure.
(c) It should be independent of source language and machine language.
(d) Translation from source to target code must be easy [12].

This representation must be easy to generate and translate into the target program. Of many forms of representation that exist today, the most common form is three-address code(TAC) which does not commit to a particular architecture and is more like general assemble language. TAC is a set of simple instructions having at most three operands [13].

2.5 Code Optimization

In this phase, intermediate language representation is transformed into functionally enhanced form that results to an improved target code. Usually "improved" has manifold interpretations based on the end objectives desired of the target code, such as a faster code, a more compact code, a code with a small memory-footprint or a code that consumes less power [8][9].

To generalize, an optimizer aims to improve the time and space requirements of the code. Several ways exist in which code can be optimised, most are uneconomical in terms of time, as well as space to implement. To list out a few:
(a) removing redundant identifiers,
(b) removing the unreachable sections of code,
(c) identifying common subexpressions,
(d) unfolding loops, and

(e) eliminating procedures [9].

There occurs a massive variation to the extent of which different compilers implement this phase. For instance, "optimizing compilers" spend a considerable amount of time in this phase [8].

2.6 Code Generation

In the final phase of the compiler, the transformed intermediate language is translated into the target language, usually the native machine language of the system (object code). The calling of resource and storage decisions, such as deciding which variables to store into registers and memory and also the selection and scheduling of the appropriate machine instructions along with their associated addressing modes is the primary purpose of this phase [9].

2.7 Symbol Table Management

A data structure that stores all the identifiers (i.e. names of variables, procedures etc.) of a source program along with the attributes of each identifier is called a symbol table. The typical attributes for a variable include:
(a) its type,
(b) the amount of memory it occupies, and
(c) its scope.
(d) For procedures and functions, the typical attributes are:
(e) the number and type of each argument (if any),
(f) the method of passing each argument, and
(g) the type of value returned (if any).

The primary purpose of this symbol table is to facilitate accelerated and uniform access to identifier attributes throughout the compilation process. The updation of the symbol table is generally done during the lexical analysis and/or syntax analysis phases [9].

3. An Illustrative Example

Consider the input:

$$d = (c + f) * (c + f) \quad (1)$$

Now, let us have an overview of the working of the different phases of the compiler with respect to the above input.

3.1 Lexical Analyzer

The lexical analyzer (or lexer) makes an entry into the symbol table for all the variables declared in the input program.
The tokens generated will be as follows:
<d,id,1><=,operator>< (,open parenthesis ><c,id,2><+,add operator><f,id,3><),close parenthesis><*,multiplication operator>< (,open parenthesis><c,id,2><+,add operator><f,id,3><),close parenthesis>
Here, id refers to identifier and the values 1, 2, and 3 refer to the entry of the respective identifiers in the symbol table.

3.2 SyntaxAnalyzer (Parser)

In this phase, the input is checked with respect to a grammar G.
Consider grammar G as follows:
E->variable=E
variable->identifier
E->E*E
E-> (E)
E->E+E
E->number
E->identifier
The parse tree generated would be as follows, as shown in Figure.4.

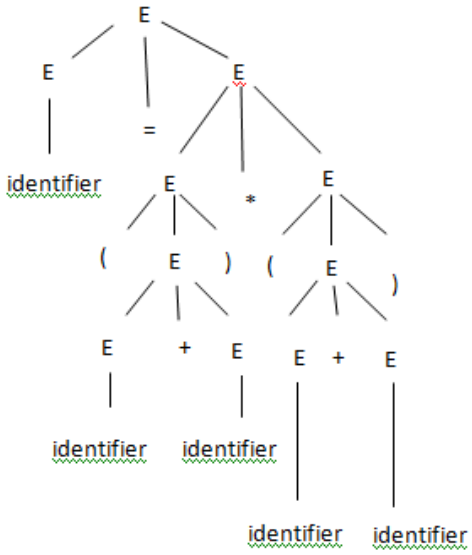

Figure 4: Parse tree

3.3 Semantic Analyzer

In this step, the meaning of the expression or statement is checked. Suppose, for the above input the variables were declared as:
floatc,f;
int d;
Then, type conversion or coercion takes place. That is, the result of (c+f)*(c+f) is computed, which is a float, which is later converted into an integer.

3.4 Intermediate Code Generator

The compiler generates temporary variables to store the intermediate result.
t1:=c+f
t2:=c+f
t3:=t1*t2
d:=t3

3.5 Code Optimizer

The optimization of code takes place as follows:
t1:=c+f
t2:=t1*t1
d:=t2

3.6 Code Generator

The compiler generates an assembly language code. This assembly language code is then converted into machine language by the assembler.

```
Load R1,c
Load R2,f
Add R1,R2
Store t1,R1
Mul R1,R1
Store t3,R1
Store d,R1
```

4. Lex and Yacc

4.1 Lex

Lex is a tool that is used for constructing a lexical analyzer. The main function of a lexical analyzer is to divide the input into tokens, that is, it performs tokenization or in other words, it generates tokens. The output produced or generated by the lexical analyzer form the input for the next phase of the compiler, which is the parser. These tokens are usually processed by a tool known as yacc (to be discussed later).

In a lex program, the user defines a set of patterns, which is compared with the given input. The set of specifications given to a Lex is known as Lex specification. If a match is found, the lex program uses the C code provided by the user. Lex itself doesn't produce an executable program; instead it translates the lex specification into a file containing a C routine called yylex(). Your program then calls yylex() to run the lexer. The set of patterns defined by the user is also known as regular expression. Lex uses a number of special characters to define a pattern. These expressions are matched by first determining the longest matching expression [6].

We write an input file, known as lex.l which clearly describes the lexical analyzer to be generated. The Lex compiler transforms this input file into a C program named lex.yy.c. This file is then compiled by the C compiler and is stored in a.out. The C compiler produces an output which consists of a lexical analyzer that takes a stream of characters as an input and then produces a stream of tokens [2].

4.1.1 Regular Expressions

A regular expression is a pattern description using a "meta" language, a language that you use to describe particular patterns of interest. The characters used in this meta language are part of the standard ASCII character set used in UNIX and MS-DOS.

Some of the characters that are used in regular expression are:
. –Used to match any single character except "\n"
[] – This denotes a character class. Characters are written inside []. Character classes matches one of the characters written inside [] the range of characters present inside []. For example, writing [A-D] is the same as writing [ABCD] where either A or B or C or D will be matched. If the character class is followed by a *, it indicates that one or more of the characters will be matched.
{} –This indicates how many times the pattern preceeded by this is allowed to match when containing one or two numbers. For example, Z {1, 4} matches one to four occurrences of the letter Z.
+ - This matches one or more occurrences of a particular symbol or a character class. For example,[A-D]+ matches "A","AA","BCA", etc. but not an empty string.
? - Matches zero or one occurrence of the preceding regular expression.
/ - Matches the preceding regular expression but only if being followed by the following regular expression. For example:

0/1

matchesthe "0" in the string "01" but would not match anything in the strings "0" or "02".

4.1.2 Example of Lex Specification for Decimal Numbers

```
%%
[\n\t ] ;

-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
printf("number\n"); }

. ECHO;
%%
main()
{
 yylex();
}
```

The above lex program ignores whitespace and echoes any characters it doesn't recognize as parts of a number to the output [5][11].

4.1.3 Conflict Resolution in Lex

When there are several prefixes of the given input string that match with two or more patterns which are defined, then the following points have to be taken into consideration:

a) Always prefer the longer prefix to the shorter one.
b) If the longest possible prefix matches two or more patterns, give preference to the pattern which is listed first in the Lex program [2].

4.2 Yacc

Lex recognizes or identifies regular expressions, whereas Yacc recognizes grammars. The tokens generated by the Lex are grouped together logically by Yacc. Yacc uses the grammar specified by the user and writes a parser which in turn recognizes "sentences" present in a grammar.
Grammars are a series of rules that is used by the parser to recognize syntactically valid input. For example,

$$stmt \rightarrow name = expr$$

$$expr \rightarrow number + number \mid number - number \quad (2)$$

In this grammar, stmt is the start symbol.The vertical bar, "|", implies that there are two possibilities for the same symbol, i.e. an expression can be number + number or number − number. The symbol present on the left of the arrow (→) is known as the left-hand side of the rule (abbreviated LHS), and the symbols to the right are the right-hand side (usually abbreviated RHS) [5].

Symbols that actually appear in the input and which are returned by the lexer are known as terminal symbols or tokens, while the symbols that appear on the left-hand side of some rule are non-terminal. Terminal and non-terminal

symbols must be different; it is an error to write a rule with a token on the left side [5].

The working of a yacc parser begins by looking for rules that might match the tokens seen so far. When yacc processes a parser, it creates a set of states each of which reflects a possible position in one or more partially parsed rules. As the parser reads tokens, each time it reads a token that doesn't complete a rule it pushes the token on an internal stack and switches to a new state reflecting the token it just read. This action is called a shift. When it has found all the symbols that constitute the right-hand side of a rule, it pops the right-hand side symbols off the stack, pushes the left-hand side symbol onto the stack, and switches to a new state reflecting the new symbol on the stack. This action is called a reduction, since it usually reduces the number of items on the stack. Whenever yacc reduces a rule, it executes user code associated with the rule.

It is also possible to write grammars that the yacc cannot handle. It cannot deal with ambiguous grammars, which is a context-free grammar for which there exists a string that can have more than one leftmost derivation or rightmost derivation.

4.2.1 Error Reporting

Error reporting should give as much information about the error as possible. The yacc error only indicates that a syntax error exists and parsing should be stopped. The different errors can be classified as:

a) General syntactic errors (e.g., a line that has no meaning)
b) A non-terminated string
c) The wrong type of string (quoted instead of unquoted or vice versa)
d) A premature end-of-file
e) Use of duplicate names

Error correction is not the responsibility of yacc alone, but also the lex, where fundamental errors are better detected [5].

4.3 Lexer Parser Communication

When a lex scanner and a yacc parser are used together, the parser is the higher level routine. It calls the lexeryylex()whenever it needs a token from the input. The lexer then scans through the input to recognize tokens. On finding a token of interest to the parser, it returns to the parser, returning the token's code.Not all tokens are of any importance to the parser. In many programming languages the parser doesn't give importance to comments and whitespace,etc. For such token which are not of importance, the lexer continues on to the next token without bothering the parser [6].

Semantic Analysis

This phase deals with checking the meaning of statements parsed in the parse tree to know if they form a meaningful set of instructions in a programming language. A program is said to be semantically correct, all the variables, functions, classes, expressions must be properly defined and must respect the type system. The Semantic Analyzer thus performs name and type resolution that is, it weeds out the possible ambiguities and errors in the program [9].

In languages where identifiers have to be declared before they are used, as new declarations are encountered, the semantic analyzer identifies and records the type of the identifier. The following checks are performed in the phase. As the examination of the program proceeds, the semantic analyzer checks if the type of identifiers is respected in operations encountered. For example, the type of right hand side expression in an assignment statement should match the type of left hand side identifier. The type and number of arguments in a function call should match the parameters of the function. Other examples include unique identifier declaration which prevents two global identifiers from having the same name [9].

5. Conclusion

This paper outlines the elemental aspects of the most popularly used translator in today's computing world, a compiler. The working principle of a compiler and its sequence of phases in translating a source program into a target program are described in-depth. Furthermore, the operation of the compiler tools – lex and yacc, is also illustrated. The primary purpose of this report is to provide an explanatory and detailed introduction to the fundamental concepts of compilers, which is of ultimate significance to any undergraduate student.

6. Acknowledgement

We would like express our sincere gratitude to Dr.K.G.Srinivasan, the Head of Computer Science Department of M.S.Ramaiah Institute of Technology, for giving us the opportunity and the much needed encouragement to venture into writing a technical paper. We are also very obliged and thankful to Ms.Parkavi A, ourhonourable faculty member and project guide for providing valuable guidance and feedback throughout the process of writing this paper. Their assistance is unmatched.

References

[1] D. Vermeir, An introduction to compilers – DRAFT, Dept. of Computer Science, VrijUniversiteit Brussel, February 2009.(book style)
[2] Alfred V. Aho, Monica S.Lam, Ravi Sethi, et al, Compilers Principles, Techniques and Tools, 2nd ed, Addision, 2006. (book style)
[3] Dr. Matt Poole 2002, edited by Mr. Christopher Whyley, 2nd Semester 2006/2007, Compilers, Department of Computer Science, University of Wales Swansea,www.compsci.swan.ac.uk/~cschris/compilers (general internet site)
[4] Biswajit Bhowmik1, Abhishek Kumar, Abhishek Kumar Jha, et al, "A New Approach of Complier Design in Context of Lexical Analyzer and Parser Generation for NextGen Languages", International Journal of Computer Applications (0975 – 8887), Volume 6– No.11, September 2010. (journal style)
[5] https://www.safaribooksonline.com/library/view/lex-yacc/9781565920002/ch01.html (General Internet site)

[6] http://dinosaur.compilertools.net/lex/(General Internet site)
[7] http://www.ijetae.com/files/Volume4Issue3/IJETAE_0314_87.pdf. (General Internet site)
[8] http://symbolaris.com/course/Compilers/waitegoos.pdf. (General Internet site)
[9] http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/180%20Semantic%20Analysis.pdf. (General Internet site)
[10] http://www.c4learn.com/c-programming/semantic-analysis/. (General Internet site)
[11] lex&yacc, 2nd Edition,By Doug Brown, John Levine, Tony Mason, O'Reilly Media, October 1992. (book style)
[12] RESEARCH PAPER ON PHASES OF COMPILER, Charu Arora, Chetna Arora, Monika Jaitwal,2014,INTERNATONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY
[13] http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/020%20CS143%20Course%20Overview.pdf. (General Internet site)

Author Profile

Neha Pathapatiis pursuing 3rd year B.E in Computer Science at M.S.Ramaiah Institute of Technology.

Niharika W.Mis pursuing 3rd year B.E in Computer Science at M.S.Ramaiah Institute of Technology.

Lakshmishree Cis pursuing 3rd year B.E in Computer Science at M.S.Ramaiah Institute of Technology.