COMPILER DESIGN

A MINI PROJECT REPORT SUBMITTED BY:

Nidhi Rai
4NM18CS103

Nishmitha K Suvarna
4NM18CS104

UNDER THE GUIDANCE OF:

Mrs. Anusha Anchan
Assistant Professor Gd II
Department of Computer science and Engineering

In partial fulfillment of the requirement for the award of the Degree of
BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING
From

Visvesvaraya Technological University, Belagaum

NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India
(ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC

☎: 08258 - 281039 - 281263, Fax: 08258 - 281265

Department of Computer Science and Engineering
B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

N.M.A.M. INSTITUTION OF TECHNOLOGY

(An Autonomous Institution affiliated to VTU, Belagavi) (NBA Accredited, ISO 9001:2008 Certified)
Nitte-574110, Karkala, Udupi District, Karnataka, India

Department of Computer Science and Engineering

CERTIFICATE

Certified that the Mini Project work entitled

COMPILER DESIGN

Is a Bonafide work carried out by

Nidhi Rai: 4NM18CS103
Nishmitha K Suvarna: 4NM18CS104

In partial fulfillment of requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering prescribed by Visvesvaraya Technology University, Belgaum during the year 2021-2022

It is certified that all corrections/suggestions indicated for Internal Assessment have been Incorporated in the report

The mini project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree

Name & Signature of Guide(s)
Mrs. Anusha Anchan
Assistant Professor, Gd II
Department of CSE

Name & Signature of HOD
Dr Jyothi Shetty
Head of the Department
Department of CSE

ACKNOWLEDGMENT

We believe that our project will be complete only after we thank the people who have contributed to make this project successful.

First and foremost, our sincere thanks to our beloved principal, **Dr Niranjan N Chiplunkar** for giving us an opportunity to carry out our project work at our college and providing us with all the needed facilities.

We express our deep sense of gratitude and indebtedness to our guide **Mrs. Anusha Anchan**, Assistant Professor GD II, Department of Computer Science and Engineering, for her inspiring guidance, constant encouragement, support and suggestion for improvement during the course of our project.

We sincerely thank **Dr Jyothi Shetty**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyantaya Memorial Institute of Technology, Nitte.

We also thank all those who have supported us throughout the entire duration of our project.

Finally, we thank the staff members of the Department of Computer Science and Engineering and all our friends for their honest opinions and suggestions throughout the course of our project.

Nidhi Rai(4NM18CS103)
Nishmitha K Suvarna (4NM18CS104)

TABLE OF CONTENTS

1. ABSTRACT

The computer is an integral tool in our lives because it turns application that solve many real-life problems. Computer programmers write programs to perform various tasks. The high-level programming languages currently used can only be understood by human beings, but not by the computer. It requires a compiler to convert this high-level language to a language that can be understood by the machine.

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers.
This report contains the details of how one can develop the simple compiler for given language using Lex (Lexical Analyzer Generator) and YACC (Yet Another Compiler). Lex tool helps write programs whose control flow is directed by instances of regular expressions in the input stream.

Lex tool source is the table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. On the other hand, YACC tool receives input of the user grammar. Starting from this grammar it generates the C source code for the parser. YACC invokes Lex to scan the source code and uses the tokens returned by Lex to build a syntax tree. With the help of YACC and Lex tool one can write their own compiler.

2. INTRODUCTION

COMPILER

A compiler is a software that takes a program written in a high-level language and translates it into an equivalent program in a target language. Most specifically a compiler takes a computer program and translates it into an object program. Some other tools associated with the compiler are responsible for making an object program into executable form.

Source program – It is normally written in a high-level programming language. It contains a set of rules, symbols and special words used to construct a computer program.

Target program – It is normally the equivalent program in machine code. It contains the binary representation of the instructions that the hardware of computer can perform.

Error Message – A message issued by the compiler due to detection of syntax errors in the source program.

Compilation is a large process. It is often broken into stages. Many phases of the compiler try and optimize by translating one form into a better (more efficient) form. Most of compiling is about “pattern matching” languages and tools that support pattern matching, are very useful. An efficient compiler must preserve semantics of the source program and it should create an efficient version of the target language.

PHASES OF COMPILERS

Typically, a compiler includes several functional parts. For example, a conventional compiler may include a lexical analyser that looks at the source program and identifies successive “tokens” in the source program. A conventional compiler also includes a parser or syntactical analyser, which takes as an input a grammar defining the language being compiled and a series of actions associated with the grammar.

The syntactical analyser builds a “parse tree” for the statements in the source program in accordance with the grammar productions and actions. For each statement in the input source program, the syntactical analyser generates a parse tree of the source input in a recursive, “bottom-up” manner in accordance with relevant productions and actions in the grammar. Generation of the parse tree allows the syntactical analyser to determine whether the parts of the source program comply with the grammar. If not, the syntactical analyser generates an error

CLASSIFICATION OF COMPILER PHASES

There are two major parts of a compiler phases: Analysis and Synthesis.
In analysis phase, an intermediate representation is created from the given source program that contains:

- Lexical Analyser
- Syntax Analyser
- Semantic Analyser

In synthesis phase, the equivalent target program is created from this intermediate representation. This contains:

- Intermediate code Generator
- Code Optimisation
- Code Generation

1. LEXICAL ANALYZER

Lexical analyser takes the source program as an input and produces a string of tokens or lexemes. Lexical Analyzer reads the source program character by character and returns the tokens of the source program. The process of generation and returning the tokens is called lexical analysis. Representation of lexemes in the form of tokens as:

2. SYNTAX ANALYSER

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program. In other words, a Syntax Analyzer takes output of lexical analyser (list of tokens) and produces a parse tree. A syntax analyser is also called as a parser. The parser checks if the expression made by the tokens is syntactically correct.

3. **SEMANTIC ANALYSER**

   Semantic analyser takes the output of syntax analyser. Semantic analyser checks a source program for semantic consistency with the language definition. It also gathers type information for use in intermediate-code generation.

4. **INTERMEDIATE CODE GENERATION**

   After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language.

5. **CODE OPTIMISER**

   The code optimizer takes the code produced by the intermediate code generator. The code optimizer reduces the code (if the code is not already optimized) without changing the meaning of the code. The optimization of code is in terms of time and space.

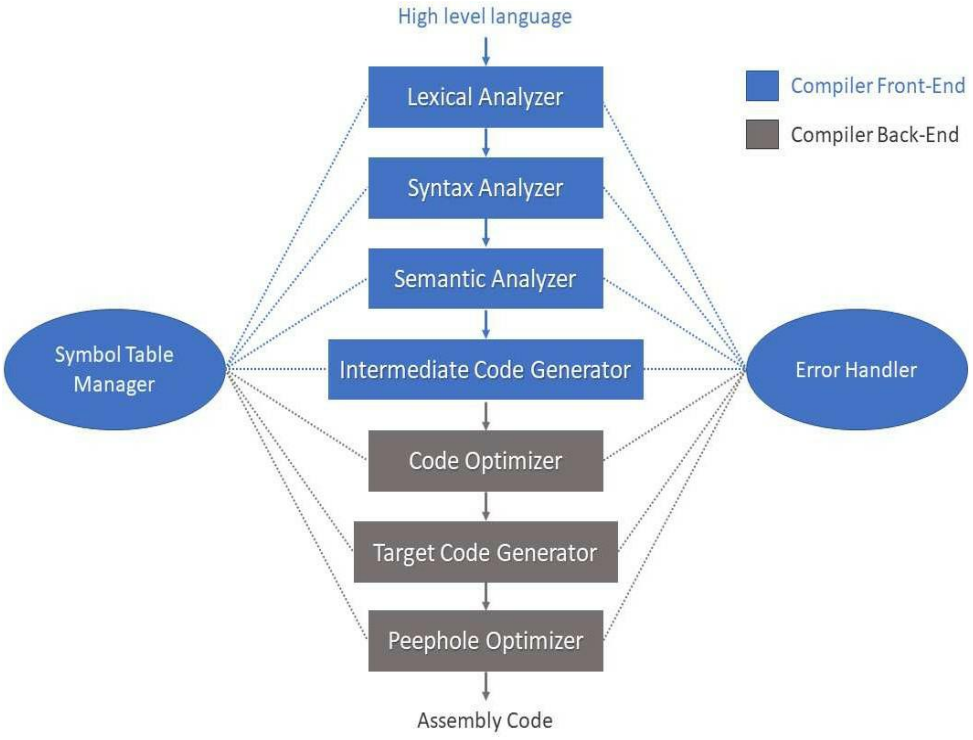
6. **CODE GENERATION**

   This produces the target language in a specific architecture. The target program is normally is an object file containing the machine codes. Memory locations are selected for each of the variables used by the program.

SYMBOL TABLE

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Phases of Compiler



3. IMPLEMENTATION

PROBLEM STATEMENT

```
int main()
begin
        int count=1;
        while(n>1)
                count=count+1;
                n=n/2;
        end while
return count
end
```

LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase.

Token:

Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are,
1) Identifiers
2) keywords
3) operators
4) special symbols
5) constants

Pattern:

A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Lex Program:

```
%{
#include "y.tab.h"
%}
%%
"int" {return INT;}
"float" {return FLOAT;}
"char" {return CHAR;}
"void" {return VOID;}
"main" {return MAIN;}
"begin" {return BEG;}
"end" {return END;}
"(" {return OPENP;}
")" {return CLOSEP;}
"return" {return RETURN;}
"while" {return WHILE;}
[a-zA-Z][a-zA-Z0-9]* {return ID;}
[0-9]+ {return NUM;}
[ ] {return SP;}
[;] {return SC;}
[ ]* {return SPS;}
[\t]* {return TAB;}
[\n] {return NL;}
[,] {return CM;}
">="|"<="|"<"|">"|"=="|"!=" {return RELOP;}
[+\-=*/] {return OPER;}
. {return yytext[0];}
%%
int yywrap(){
        return 1;
}
```

Yacc Program:

```
%{
#include<stdio.h>
%}
%token MAIN OPENP CLOSEP BEG END INT VOID CHAR FLOAT ID NUM SC NL SP CM TAB SPS WHILE RELOP OPER RETURN
%%
stmt: type SP MAIN OPENP CLOSEP NL space BEG NL space stmts space RETURN SP vals NL space END {printf("\nValid
 string\n"); return 1;};
space: SPS space|
       TAB space|
       SP space|;
vals: ID|
      NUM;
stmts: type SP space varlist SC NL stmts|
       while_stmt NL stmts|
       expr SC NL stmts|
       ;
varlist: ID space|
         expr space|
         expr space CM|
         ID space CM;
type: INT|
      VOID|
      FLOAT|
      CHAR;
while_stmt: WHILE OPENP space vals space RELOP space vals space CLOSEP NL space stmts space END SP WHILE;
expr: expr space OPER space expr|
      vals;
%%
int yyerror(char* msg){
        printf("\nInvalid string\n");
        return 0;
}
void main(){
        printf("Enter the string:\n");
        yyparse();
}
```

Output:

```
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ gedit project.l
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ gedit project.y
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ lex project.l
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ yacc -d project.y
project.y: warning: 21 shift/reduce conflicts [-Wconflicts-sr]
project.y: warning: 3 reduce/reduce conflicts [-Wconflicts-rr]
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ cc y.tab.c lex.yy.c
y.tab.c: In function ‘yyparse’:
y.tab.c:1298:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
 1298 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1431:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
 1431 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ ./a.out
Enter the string:
int main()
begin
int count=1;
while(n>1)
count=count+1;
n=n/2;
end while
return count
end

Valid string
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ ./a.out
Enter the string:
void main()
end

Invalid string
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$
```

SYNTAX ANALYSIS

In our compiler model, the parser obtains a string of tokens from the lexical analyser, as shown in the figure below, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing

Parser:

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer. The parser obtains a string of tokens from the lexical analyser and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

Code Screen Shots

input.txt

```
int main()
begin
        int count=1;
        while(n>1)
                count=count+1;
                n=n/2;
        end while
return count
end
```

ourgrammer.txt

```
S->im()Ba
A->ic=1;B
B->w(n>1)C
C->c=c+1;D
D->n=n/2;E
E->ewF
F->rce
```