

# Compiler Construction Project Report

Aashish Kumar Reddy  
Prathamesh Chavan  
Jinisha Shah

# Table of Contents

I. Introduction.....	3
II. Parser.....	6
III. Lexical Analyzer.....	9
IV. Code Generation.....	12
V. Error Checking.....	14
VI. Screen Shots.....	15
VII. What we learnt from Project?.....	19
VIII. Contribution.....	19

## **INTRODUCTION**

This project deals with the Designing of the Compiler. In this Project we design a compiler wherein we convert a code written in CoreTL3 to C. Then we check the C code to see that it produces the correct results when written in C.

This Project consists of 4 phases. They are:

- a) Lexical Analyses Phase or Scanning.
- b) Syntax Analysis Phase or Parsing.
- c) Code Generation Phase
- d) Error Checking Phase

A translator is a program, which converts programs written in a source language into an equivalent program in an object language. The source language is a high-level programming language and the object language is a machine language of an actual computer. From the realistic view, a translator defines the semantics of the programming language, it transforms operations specified by the syntax into operations of the computational model to some real or virtual machine. Context-free grammars are used in the construction of language translators. Since the translation is guided by the syntax of the source language, the translation is said to be syntax-directed.

A compiler is a program, which translates a source program written in a high level language to another computer language (target language). A Compiler consists of generally of the following phases, they are:

- Lexical phase groups characters into lexical units or tokens. The input to the scanner is a character stream. The output is a stream of tokens. Using Regular expressions we define the tokens recognized by a scanner. The scanner is implemented as a finite state machine. Lex and Flex are tools for generating scanners.
- The parser or syntax phase group's tokens into syntactical units. The output of this phase is a parse tree representation of the program. Context-free grammars define the program structure recognized by a parser. The parser is implemented as pushdown automata. Yacc and Bison are tools for generating parsers.
- The semantic analysis phase analyzes the parse tree for context-sensitive information often called the static semantics. The output is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program. This phase is combined with the parser. During this phase, information regarding variables and objects is stored in a symbol table.
- The optimizer applies semantics and conserves transformations to the annotated parse tree to simplify the structure of tree and facilitate generation of more efficient code.
- Code generator converts the simplified parse tree into object code using rules, which denote the semantics of the source language. The code generator is integrated with the parser.

In this project we make go through all the phases and use them to build a compiler. In this project we construct a compiler for a programming language called TL3. The context free grammar for this is:

$\langle \text{program} \rangle ::= \text{PROGRAM } \langle \text{declarations} \rangle \text{ BEGIN } \langle \text{statementSequence} \rangle \text{ END}$

$\langle \text{declarations} \rangle ::= \text{VAR ident AS } \langle \text{type} \rangle \text{ SC } \langle \text{declarations} \rangle$   
|  $\epsilon$

$\langle \text{type} \rangle ::= \text{INT} \mid \text{BOOL}$

$\langle \text{statementSequence} \rangle ::= \langle \text{statement} \rangle \text{ SC } \langle \text{statementSequence} \rangle$   
|  $\epsilon$

$\langle \text{statement} \rangle ::= \langle \text{assignment} \rangle$   
|  $\langle \text{ifStatement} \rangle$   
|  $\langle \text{whileStatement} \rangle$   
|  $\langle \text{writeInt} \rangle$

$\langle \text{assignment} \rangle ::= \text{ident ASGN } \langle \text{expression} \rangle$   
|  $\text{ident ASGN READINT}$

$\langle \text{ifStatement} \rangle ::= \text{IF } \langle \text{expression} \rangle \text{ THEN } \langle \text{statementSequence} \rangle \langle \text{elseClause} \rangle \text{ END}$

$\langle \text{elseClause} \rangle ::= \text{ELSE } \langle \text{statementSequence} \rangle$   
|  $\epsilon$

$\langle \text{whileStatement} \rangle ::= \text{WHILE } \langle \text{expression} \rangle \text{ DO } \langle \text{statementSequence} \rangle \text{ END}$

$\langle \text{writeInt} \rangle ::= \text{WRITEINT } \langle \text{expression} \rangle$

$\langle \text{expression} \rangle ::= \langle \text{simpleExpression} \rangle$   
|  $\langle \text{simpleExpression} \rangle \text{ OP4 } \langle \text{simpleExpression} \rangle$

$\langle \text{simpleExpression} \rangle ::= \langle \text{term} \rangle \text{ OP3 } \langle \text{term} \rangle$   
|  $\langle \text{term} \rangle$

$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \text{ OP2 } \langle \text{factor} \rangle$   
|  $\langle \text{factor} \rangle$

$\langle \text{factor} \rangle ::= \text{ident}$   
|  $\text{num}$   
|  $\text{boollit}$   
|  $\text{LP } \langle \text{expression} \rangle \text{ RP}$

where the non-terminal symbols are given in all lower case, the terminal symbols are given in all caps or as literal symbols. The start symbol is “program”. While the grammar uses uppercase to highlight terminal symbols, they are to be implemented in lowercase.

## **THE PARSER**

A parser is a program, which determines input is syntactically valid, and determines the structure. Parsers may be hand written or may be automatically generated by a parser generator from descriptions of valid syntactical structures. The descriptions are in the form of a context-free grammar.

Yacc is a program, which takes a context-free grammar and constructs a C program which will parse input according to the grammar rules. A input file for Yacc is of the form:

C and parser declarations

%%

Grammar rules and actions

%%

C subroutines

The first section of the Yacc file consists of a list of tokens that are expected by the parser and the specification of the start symbol of the grammar.

```
%union {
    int ival;
    char *bool;
    char *str1;
    struct nodeType *a_tree;
}
%start program
%token PROGRAM START END VAR AS SC READINT WRITEINT IF THEN ELSE
WHILE DO LP RP INT BOOL ASGN
%token<ival> num OP2 OP3 OP4
%token<ival> boollit
%token<str1> ident
%type <a_tree> assignment factor expression ifStatement statementSequence elseClause
whileStatement simpleExpression term statement program declarations type writeInt readInt
```

The second section of the Yacc file consists of the context-free grammar for the language. Productions are separated by semicolons, the '::<=' symbol of the BNF is replaced with ':', the empty production is left empty, non-terminals are written in all lower case, and the multicharacter terminal symbols in all uppercase. The context free-grammar is modified to include calls to the install and context checking functions. \$n is a variable internal to Yacc which refers to the semantic record corresponding the nth symbol on the right hand side of a production.

```

program : PROGRAM declarations START statementSequence END { $$ =
create_node(PROGRAM, 2, $2, $4); } ex($$); }
;
declarations : VAR ident AS type SC declarations { $$ = create_node(AS, 3, variable("var"), $4,
$6); }
| { $$ = NULL; }
;
type : INT { $$ = str("int"); }
| BOOL { $$ = str("bool"); }
;
statementSequence : statement SC statementSequence { $$ = create_node(SC, 2, $1, $3); }
| { $$ = NULL; }
;
statement : assignment { $$ = $1; }
| ifStatement { $$ = $1; }
| whileStatement { $$ = $1; }
| writeInt { $$ = $1; }
;
assignment : ident ASGN expression { $$ = create_node(ASGN, 2, variable("var"), $3); }
| ident ASGN READINT { int x; scanf("%d", &x);
$$ = create_node(ASGN, 2, variable("var"), NULL); }
;
ifStatement : IF expression THEN statementSequence elseClause END { $$ =
create_node(IF, 3, $2, $4, $5); }
;
elseClause : ELSE statementSequence { $$ = create_node(ELSE, 1, $2); }
| { $$ = NULL; }
;
whileStatement : WHILE expression DO statementSequence END { $$ = create_node(WHILE,
2, $2, $4); }
;
writeInt : WRITEINT expression { $$ = create_node(WRITEINT, 1, $2); }
;
expression : simpleExpression { $$ = $1; }
| simpleExpression OP4 simpleExpression { $$ = create_node($2, 2, $1, $3); }
;
simpleExpression : term OP3 term { $$ = create_node($2, 2, $1, $3); }
| term { $$ = $1; }
;
term : factor OP2 factor { $$ = create_node($2, 2, $1, $3); }
| factor { $$ = $1; }
;
factor : ident { $$ = variable("var"); }
| num { $$ = literal($1); }
| boollit { $$ = literal($1); }
| LP expression RP { $$ = $2; }

```

```
    ;  
%%
```

The third section of the Yacc file consists of C code. There must be a main() routine which calls the function yyparse(). The function yyparse() is the driver routine for the parser. There must also be the function yyerror() which is used to report on errors during the parse. Simple examples of the function main() and yyerror() are:

```
void yyerror(char *msg){  
    fprintf(stderr,"%s : on line %d\n",msg,yylineno);  
    exit(1);  
}  
  
int main(void){  
    yyparse();  
    printf("*****PARSING COMPLETED*****\n\n");  
}
```

The parser, doesn't generate output however, the parse tree is constructed implicitly during the parse. As the parser executes, it builds an internal representation of the structure of the program. The internal representations based on the right hand side of the production rules. When a right hand side is recognized, it is reduced to the corresponding left hand side. Parsing is complete when the entire program has been reduced to the start symbol of the grammar.

We Compile the Yacc file with the command `bison -d parser.y` which causes the generation of two files `parser.tab.h` and `parser.tab.c`. The `parser.tab.h` contains the list of tokens is included in the file which defines the scanner. The file `parser.tab.c` defines the C function `yyparse()` which is the parser. Yacc is distributed with the Unix operating system while Bison is a product of the Free Software Foundation, Inc.



## **LEXICAL ANALYZER**

A lexical analyzer is a program which recognizes patterns in text. The Lexical analyzer recognizes tokens defined in the regular expressions. Lex is a lexical analyzer generator. The input to Lex is a file containing tokens defined using regular expressions. Lex produces an entire scanner module that can be compiled and linked to other compiler modules. A input file for Lex is of the form:

Lex generates a file containing the function `yylex()` which returns an integer denoting the token recognized.

C and scanner declarations

%%

Token definitions and actions

%%

C subroutines

The first section of the Lex file contains the C declaration to include the file `parse.tab.h` produced by Yacc/Bison which contains the definitions of the the multi-character tokens. The first section also contains Lex definitions used in the regular expressions.

```
% {
#include<stdio.h>
#include<string.h>
#include "parse.tab.h"
void yyerror(char *);

% }
```

The second section of the Lex file gives the regular expressions for each token to be recognized and a corresponding action. Strings of one or more digits are recognized as an integer and thus the value `INT` is returned to the parser. The reserved words of the language are strings of lower case letters (upper-case maybe used but must be treated differently). Blanks, tabs and newlines are ignored. All other single character symbols are returned as themselves (the scanner places all input in the string `yytext`).

```
"\n"      yylineno++;
[ \r\n\t]* ;
"("      return LP;
")"      return RP;
":="     {
                yyval.str1 = malloc(strlen(yytext));
                strncpy(yyval.str1, yytext, strlen(yytext));
                return ASGN;
        }
```

```

";"          return SC;

"*" |
"div" |
"mod"       {
    yylval.str1 = malloc(strlen(yytext));
    strncpy(yylval.str1, yytext, strlen(yytext));
    return OP2;
}

"+" |
"-"       {
    yylval.str1 = malloc(strlen(yytext));
    strncpy(yylval.str1, yytext, strlen(yytext));
    return OP3;
}

"=" |
"!=" |
"<" |
">" |
"<=" |
">="   {
    yylval.str1 = malloc(strlen(yytext));
    strncpy(yylval.str1, yytext, strlen(yytext));
    return OP4;
}

if          return IF;
then       return THEN;
else       return ELSE;
begin     return START;
end       return END;
while     return WHILE;
do        return DO;
program   return PROGRAM;
var       return VAR;
as        return AS;
int       return INT;
bool      return BOOL;
writeInt  return WRITEINT;
readInt   return READINT;

[1-9][0-9]*|0 {
    yylval.ival = atoi(yytext);
    return num;
}

false|true {
    yylval.bool = malloc(strlen(yytext));

```

```

        strncpy(yylval.bool, yytext, strlen(yytext));
        return boollit;
    }
[A-Z][A-z0-9]*    {
        yylval.str1 = malloc(strlen(yytext));
        strncpy(yylval.str1, yytext, strlen(yytext));
        return ident;
    }
.                yyerror("INVALID CHARACTER");

```

The values associated with the tokens are the integer values that the scanner returns to the parser upon recognizing the token.

The third section of the file is empty in this example but may contain C code associated with the actions. Compiling the Lex file with the command `flex file.lex` results in the production of the file `lex.yy.c` which defines the C function `yylex()`. On each invocation, the function `yylex()` scans the input file and returns the next token.

## **CODE GENERATION**

Lex and Yacc files can be extended to handle the context sensitive information. For example, suppose we want to require that, in TL3, we require that variables are declared before being referenced. The parser must be able to compare variable references with the variable declarations. One way to accomplish this is to construct a list of the variables during the parse of the declaration section and then check variable references against the those on the list. Such a list is called a symbol table. Symbol tables may be implemented using lists, trees, and hash-tables.

### **SYMBOL TABLE**

A symbol table contains the environmental information concerning the attributes of various programming language constructs. In particular, the type and scope information for each variable. The symbol table will be developed as a module to be included in the yacc/bison file. The symbol table declarations are:

```
struct SymbTab
{
    char label[10];
    int addr;
    struct SymbTab *next;
};
struct SymbTab *first,*last;

int Insert(char lab[])
{
    int n,j;
    n=Search(lab);
    if(n==0)
    {
        struct SymbTab *p;
        p=malloc(sizeof(struct SymbTab));
        strcpy(p->label,lab);
        p->addr=q;

        q++;
        p->next=NULL;
        if(size==0)
        {
            first=p;
            last=p;
        }
        else
        {
            last->next=p;
            last=p;
        }
    }
}
```

```

        }
        size++;
        j=1;
        return j;
    }
}

int Search(char lab[])
{
    int i,flag=0;
    struct SymbTab *p;
    p=first;
    for(i=0;i<size;i++)
    {
        if(strcmp(p->label,lab)==0)
            flag=1;
        p=p->next;
    }
    return flag;
}

```

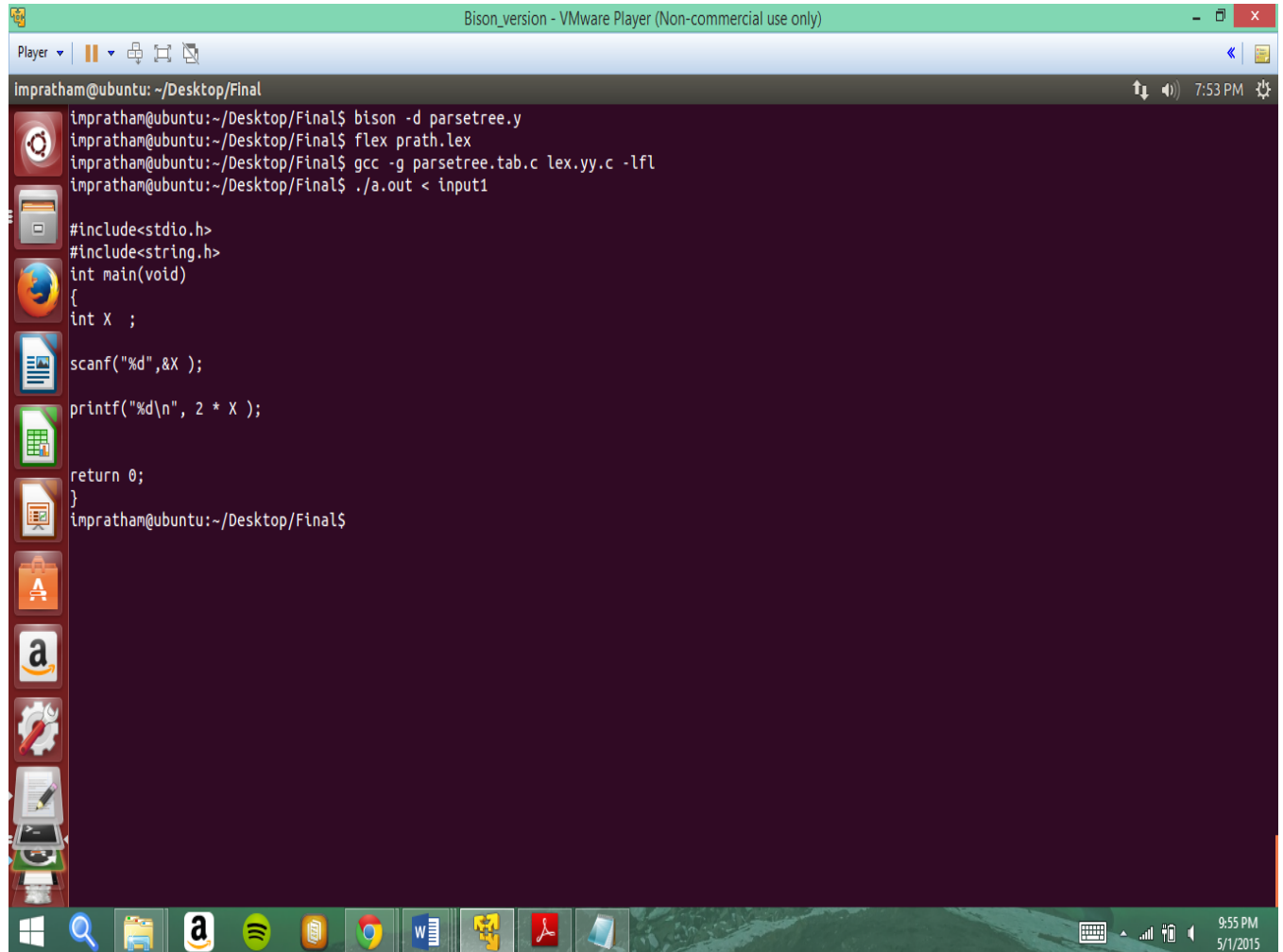
## **ERROR CHECKING**

Before handling the error, we must ensure that we can detect them. Error Checking is done to verify that whether the program generates error on giving wrong input. Good Compiler is the one who catches the maximum number of errors in least time. A compiler performs Syntactic Analysis of the source program in order to check whether the program is valid in a given programming language.

For example

- If OP2, OP3, OP4 gets a value which is double or Boolean, it will display an error as it can only take integer values. Similar, if the result generated by OP2 OP3 is not an integer, then an error will be displayed as the final result of that should be an integer, whereas OP4 should generate a Boolean result.
- Error will be displayed if a variable type is undeclared.
- Any particular variable can be declared only once, defining the variable more than once will give an error.
- The left hand side of assignment must be a variable and its right hand side should be an expression of the variable's type
- Error will be displayed if say float or Boolean value is assigned to the result of readInt, because the result of readInt should be an integer.
- Also, in the same manner, writeInt will give error if its evaluating float or Boolean.
- The literals "true" and "false" should have Boolean values otherwise the code will generate an error.

# SCREEN SHOTS



Bison\_version - VMware Player (Non-commercial use only)

Player

impratham@ubuntu: ~/Desktop/Final 7:53 PM

```
impratham@ubuntu:~/Desktop/Final$ bison -d parsetree.y
impratham@ubuntu:~/Desktop/Final$ flex prath.lex
impratham@ubuntu:~/Desktop/Final$ gcc -g parsetree.tab.c lex.yy.c -lfl
impratham@ubuntu:~/Desktop/Final$ ./a.out < input1

#include<stdio.h>
#include<string.h>
int main(void)
{
    int X ;

    scanf("%d",&X );

    printf("%d\n", 2 * X );

    return 0;
}
impratham@ubuntu:~/Desktop/Final$
```

9:55 PM 5/1/2015

Bison\_version - VMware Player (Non-commercial use only)

```
Player | [Icons] [7:50 PM]

Impratham@ubuntu: ~/Desktop/Final
Impratham@ubuntu:~/Desktop/Final$ bison -d parsetree.y
Impratham@ubuntu:~/Desktop/Final$ flex prath.lex
Impratham@ubuntu:~/Desktop/Final$ gcc -g parsetree.tab.c lex.yy.c -lfl
Impratham@ubuntu:~/Desktop/Final$ ./a.out < input2

#include<stdio.h>
#include<string.h>
int main(void)
{
    int X ;
    int Y ;

    scanf("%d",&X );
    Y = 5 - X ;

    printf("%d\n",X - Y );

    return 0;
}
Impratham@ubuntu:~/Desktop/Final$
```

[Taskbar: Windows, Search, Firefox, Amazon, Spotify, Docker, Chrome, Word, VS Code, PDF Reader, File Explorer]

[System Tray: 9:52 PM 5/1/2015]

Bison\_version - VMware Player (Non-commercial use only)

```
Player | [Icons] [7:51 PM]

Impratham@ubuntu: ~/Desktop/Final
Impratham@ubuntu:~/Desktop/Final$ bison -d parsetree.y
Impratham@ubuntu:~/Desktop/Final$ flex prath.lex
Impratham@ubuntu:~/Desktop/Final$ gcc -g parsetree.tab.c lex.yy.c -lfl
Impratham@ubuntu:~/Desktop/Final$ ./a.out < input3
syntax error : on line 15
Impratham@ubuntu:~/Desktop/Final$
```

[Taskbar: Windows, Search, Firefox, Amazon, Spotify, Docker, Chrome, Word, VS Code, PDF Reader, File Explorer]

[System Tray: 9:53 PM 5/1/2015]



Bison\_version - VMware Player (Non-commercial use only)

```
Player
Impratham@ubuntu: ~/Desktop/Final
Impratham@ubuntu:~/Desktop/Final$ bison -d parsetree.y
Impratham@ubuntu:~/Desktop/Final$ flex prath.lex
Impratham@ubuntu:~/Desktop/Final$ gcc -g parsetree.tab.c lex.yy.c -lfl
Impratham@ubuntu:~/Desktop/Final$ ./a.out < input3

#include<stdio.h>
#include<string.h>
int main(void)
{
    int N ;
    int SQRT ;

    scanf("%d",&N );
    SQRT = 0 ;

    while(SQRT * SQRT <= N )
    {
        SQRT =SQRT + 1 ;
    }
    SQRT =SQRT - 1 ;
    printf("%d\n",SQRT );

    return 0;
}
Impratham@ubuntu:~/Desktop/Final$
```

9:56 PM 5/1/2015

Bison\_version - VMware Player (Non-commercial use only)

```
Player
Impratham@ubuntu: ~/Desktop/Final
Impratham@ubuntu:~/Desktop/Final$ bison -d parsetree.y
Impratham@ubuntu:~/Desktop/Final$ flex prath.lex
Impratham@ubuntu:~/Desktop/Final$ gcc -g parsetree.tab.c lex.yy.c -lfl
Impratham@ubuntu:~/Desktop/Final$ ./a.out < Fibo

#include<stdio.h>
#include<string.h>
int main(void)
{
    int N1 ;
    int N2 ;
    int NEXT ;
    int MAX ;

    scanf("%d",&MAX );
    N1 = 0 ;
    N2 = 1 ;

    printf("%d\n",N1 );

    while(N2 < MAX )
    {
        printf("%d\n",N2 );
        NEXT =N1 + N2 ;
        N1 =N2 ;
        N2 =NEXT ;
    }

    return 0;
}
Impratham@ubuntu:~/Desktop/Final$
```

DG860A12 9:58 PM Internet access 5/1/2015

Bison\_version - VMware Player (Non-commercial use only)

Player

Impratham@ubuntu: ~/Desktop/Final

```
Impratham@ubuntu:~/Desktop/Final$ bison -d parsetree.y
Impratham@ubuntu:~/Desktop/Final$ flex prath.lex
Impratham@ubuntu:~/Desktop/Final$ gcc -g parsetree.tab.c lex.yy.c -lfl
Impratham@ubuntu:~/Desktop/Final$ ./a.out < input4
```

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    int N ;
    int I ;

    scanf("%d",&N );
    I = 2 ;

    while(I <= N )
    {
        while(N % I == 0 )
        {
            N =N / I ;
            printf("%d\n",I );
        }
        I =I + 1 ;
    }

    return 0;
}
```

Impratham@ubuntu:~/Desktop/Final\$

10:00 PM  
5/1/2015

## **What we learnt from the Project?**

This project gave an insight into some of the complicated tools. It helped us learn and use new tools such as YAC/FLEX.

Also, the project was divided in phases, which made it easier for us to understand each and every part of project description and also made it much simpler to implement. The project also gave some really difficult challenges wherein we had to optimize our code multiple times in order to get the accurate result.

This project first started with very basic use of FLEX and then went into the depth of using it. Since these were new technology, some examples during the lecture also contributed towards understanding it. At the end of this project, each member in the team has a complete overview of how YAC/FLEX can be used.

## **CONTRIBUTION**

The project was a team effort. Each member had equal participation during the course of the project.

Aashish Kumar Reddy performed token generation using LEX tool, Symbol table and Error Handling.

Jinisha Shah was involved in implementation of Parse tree from the giving BNF using YACC.

Code Generation from the designed parse tree is done by Prathamesh Chavan.

However, everyone had put their equal efforts and discussed each and every part of the project carefully.