NES's

GANGAMAI COLLEGE ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

Laboratory Manual

Class: B.E. Computer **Semester: VII**

Subject: Compile Design Lab - I

Academic Year: 2023-24

Institute Vision

- Empowering first generation engineers to excel in technical education based on human values

Institute Mission

- To import affordable and quality education in order to meet needs of industry and to achieve excellence in teaching learning process.
- To achieve excellence in application oriented research inselected area of Technology to contribute to thedevelopment of the region.
- To collaborate with industries to promote innovation capabilities of budding engineers.
- To develop responsible citizens to awareness and acceptance of ethical values.
- To build a support system of all stakeholders to develop the institute.

DEPARTMENT OF COMPUTER ENGINEERING

Vision

To emerge as the leading Computer Engineering department forinclusive development of students.

Mission

To provide student-centered conducive environment for preparingknowledgeable, competent and value-added computer engineers.

A Laboratory Manual

For

Compile Design – Lab

For the Bachelor of Engineering in the Computer Engineering

BE Computer

Semester – VII

2023-2024

Name: …………………………………………………………………………………………………..

Roll No:…………………………………………………Batch:…………………………………..

PRN NO:…………………………………………………………………………………………

Nagaon Education Society's
GANGAMAI COLLEGE OF ENGINEERING,
NAGAON, DHULE-05
Computer Department

CERTIFICATE

This is to certify that

Mr./Mr.____

Having Roll No. __

Of VII Semester for the course Bachelor of Computer Engineering

Of the institute **Gangamai College of Engineering, Nagaon, Dhule**,has

completed the term work satisfactorily of the subject

Engineering Compile Design - Lab

for the academic year 2023 – 2024

as prescribed in the curriculum

Date:_________________ **PRN No:**______________________

Place: Nagaon, Dhule Exam Seat No: _______________

Subject Teacher **HOD** **Principal**

Seal of the Institute

PRACTICAL-COURSE OUTCOMES

COURSE OUTCOMES(CO$_{S)}$

1. Demonstrate LEX and YACC tools.
2. Design Lexical Analyzer.
3. Design Syntax Analyzer.
4. Design Code Optimization.
5. Design Code Generator

Expt No.	Name of Experiment	Page No.	Starting Date	Ending Date	Remark
1.	Implement a lexical analyzer for a subset of C using LEX Implementation should support Error Handling.				
2.	To Implement a lexical analyzer of identification of numbers.				
3.	To Implement a Calculator using LEX and YACC.				
4.	Implementation of Context Free Grammar.				
5.	Implementation of Code Generator.				
6.	Implementation of Deterministic Finite Automaton.				

DEPARTMENT OF COMPUTER ENGINEERING

Objectives

1. Demonstrate LEX and YACC tools.
2. Design Lexical Analyzer.
3. Design Syntax Analyzer.
4. Design Code Optimization.
5. Design Code Generator

Gangamai College of Engineering, Nagaon, Dhule
Department of Computer Engineering

Name: **Date of Performance: _ _ /_ _/20_ _**
Class: B.E. Computer **Date of Completion: _ _ /_ _/20_ _**
Div: Batch: _ _ _ _ _

Roll No:

Subject : Compiler Design lab **Sign. of Teacher with Date**

Experiment No. 1

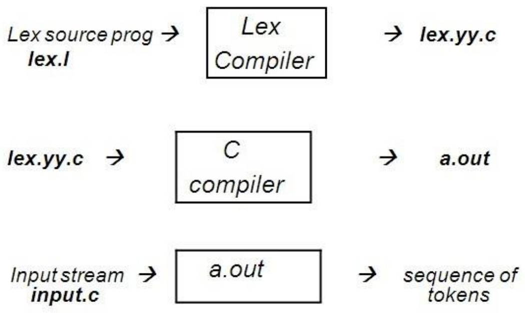
Aim: Implement a lexical analyzer for a subset of C using LEX Implementation should support Error Handling.

1. Objective: To able implement a lexical analyzer for a subset of C using LEX Implementation.

2. Background:
As the first phase of 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. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. The lexical analyzer not only identifies the lexemes but also pre-processes the source text like removing comments, white spaces, etc.
Lexical analyzers are divided into a cascade of two processes:
Scanning - It consists of simple processes that do not require the tokenization of the input such as deletion of comments, compaction of consecutive white space characters into one.
Lexical Analysis- This is the more complex portion where the scanner produces sequence of tokens as output.
A Token is pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing the kind of lexical unit, eg., a particular keyword or an identifier.
A pattern is a description of the form that the lexemes of a token may take. In case of a keyword as a token the pattern is just a sequence of characters that form the keyword.
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.

3. Pre-lab Task:
- **LEX:** A tool widely used to specify lexical analyzers for a variety of languages, refer to the tool as *Lex compiler* , and to its input specification as the *Lex language.*
- The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements (*tokens*)

```
EX:     a    =    b  +    c    *    d    ;
      ID ASSIGN ID PLUS ID MULT ID  SEMI
```
- Lex is an utility to help you rapidly generate your scanners



4. In lab Task:

1. First, write down the **lexical specification** using **regular expression** to specify the lexical structure:

   identifier = letter (letter | digit |
   underscore)* letter = a | ... | z | A | ...
   | Z
   digit = 0 | 1 | ... | 9

   based on the above **lexical specification**, build the lexical analyzer (to recognize tokens) by hand,

   Regular Expression Spec ==> NFA ==> DFA ==>Transition Table ==> Lexical Analyzer

2. Sample Program Template:

```
digit

[0-9]
letter

[a-zA-
Z]
%%
{letter}({letter}|{digit})*       printf(“id: %s\n”, yytext);
\n                                printf(“new line\n”);
%%
main()
 { yylex(); }
```

5. Post-lab Task:

Outcomes:

Able to implement and simulate a lexical analyzer for a subset of C using.

```
%{
    #include<math.h>
    FILE *fp;
%}

digit            [0-9]+
header           "#include<"[a-z]+".h>"
inbuilt
      [\t]*"printf(".*")"|[\t]*"scanf(".*")"|[\t]*"clrscr()"|[\t]*"getc
h()"|[\t]*"exit(.+)"\n
comment     [\t ]*"/*".*"*/"[\t]*\n
main_function    "void main()"
function    [\t]*[a-zA-Z]+"(".*")"[\t]*\n
datatype    [\t ]*"int"|[\t ]*"char"|[\t ]*"float"
operator    "+"|"-"|"*"|"/"
terminator       ";"
bracket1         [\t]*"{"[\t]*
bracket2         [\t]*"}"[\t]*
loop             [\t ]*"if(".*")"|[\t ]*"else"|[\t ]*"for"|[\t
]*"while"|[\t ]*"do"
relational       [\t]*"<"|">"|"<="|">="|"=="|"="|"!="|"%"
logical     [\t]*"&&"|"||"
word             [a-z]+[a-z0-9]*

%%

{digit}         {printf("\n Numbers                :: %s",yytext);}
{header}        {printf("\n Header File            :: %s",yytext);}
{inbuilt}       {printf("\n Function               :: %s",yytext);}
{comment}       {printf("\n Comment                :: %s",yytext);}
{main_function} {printf("\n Main Function          :: %s",yytext);}
{function}      {printf("\n User function          :: %s",yytext);}
{datatype}      {printf("\n Datatype               :: %s",yytext);}
{operator}      {printf("\n Operator               :: %s",yytext);}
{terminator}    {printf("\n Terminator             :: %s",yytext);}
{bracket1}      {printf("\n Opening curly bracket::  %s",yytext);}
{bracket2}      {printf("\n Closing curly bracket::  %s",yytext);}
{relational}    {printf("\n Realtional operator    :: %s",yytext);}
{loop}          {printf("\n Loop Statement         :: %s",yytext);}
{logical}       {printf("\n Logical Operator       :: %s",yytext);}
{word}          {printf("\n Variables              :: %s",yytext);}

%%
int main(int argc,char *argv[])
{
fp=fopen(argv[1],"r"); if(fp!=NULL)
   {
      yyin=fp;
      yylex();
   }
   return(0);
}
```

Gangamai College of Engineering, Nagaon, Dhule
Department of Computer Engineering

Name: **Date of Performance: _ _ /_ _/20_ _**
Class: B.E. Computer **Date of Completion: _ _ /_ _/20_ _**
Div: Batch: _ _ _ _ _

Roll No:

Subject : Compiler Design lab **Sign. of Teacher with Date**

Experiment No. 2

Aim: To Implement a lexical analyzer of identification of numbers

1. Objective: Implement a lexical analyzer of identification of numbers
(Numbers can be binary, octal, decimal, hexadecimal, float or exponential)

2. Background:
A binary number is a number expressed in the binary numeral system or base-2 numeral system which represents numeric values using two different symbols: typically 0 (zero) and 1 (one). The base-2 system is a positional notation with a radix of 2. Because of its straightforward implementation in digital electronic circuitry using logic gates, the binary system is used internally by almost all modern computers and computer-based devices.

The octal numeral system, or oct for short, is the base-8 number system, and uses the digits 0 to 7. Octal numerals can be made from binary numerals by grouping consecutive binary digits into groups of three (starting from the right). For example, the binary representation for decimal 74 is 1001010, which can be grouped into (00)1 001 010 – so the octal representation is 112

The decimal numeral system (also called base 10 or occasionally denary) has ten as its base. It is the numerical base most widely used by modern civilizations.[1][2]

Decimal notation often refers to a base 10 positional notation such as the Hindu-Arabic numeral system or rod calculus;[3]however, it can also be used more generally to refer to non-positional systems such as Roman or Chinese numerals which are also based on powers of ten.

Hexadecimal (also base 16, or hex) is a positional numeral system with a radix, or base, of 16. It uses sixteen distinct symbols, most often the symbols 0–9 to represent values zero to nine, and A, B, C, D, E, F (or alternatively a,b, c, d, e, f) to represent values ten to fifteen. Hexadecimal numerals are widely used by computer system designers and programmers. Several different notations are used to represent hexadecimal constants in computing languages; the prefix "0x" is widespread due to its use in Unix and C (and related operating systems and languages).

Alternatively, some authors denote hexadecimal values using a suffix or subscript. For example, one could write 0x2AF3 or $2AF3_{16}$, depending on the choice of notation.

Floating point is the formulaic representation that approximates a real number so as to support a trade-off between range and precision. A number is, in general, represented approximately to a fixed number of significant digits (the significand) and scaled using an exponent; the base for the scaling is normally two, ten, or sixteen. A number that can be represented exactly is of the following form:

$$\text{significand} \times \text{base}^{\text{exponent}},$$

3. Pre-lab Task:

First, we've already shown regular expression for a "digit":

[O-9]

We can use this to build a regular expression for an integer:

We require at least one digit. This would have allowed no digits at all: Let's add an optional unary minus: We can then expand this to allow decimal numbers. First we will specify a decimal number (for the moment we insist that the last character always be a digit):

[0-9]*\. [0-9]+

Notice the "\" before the period to make it a literal period rather than a wild card character. This pattern matches "0.0", ***"4.5",*** or ".31415". But it won't match *"0"* or "2". We'd like to combine our definitions to match them as well. Leaving out our unary minus, we could use: We use the grouping symbols *"0*t*"*o specify what the regular expressions are for the " I " operation. Now let's add the unary minus:

-?(([0-9]+) 1 ([0-9]*\.[0-9]+))

We can expand this further by allowing a float-style exponent to be specified as well. First, let's write a regular expression for an exponent: This matches an upper- or lowercase letter E, then an optional plus or minus sign, then a string of digits. For instance, this will match "e12" or "E-3". We can then use this expression to build our final expression, one that specifies a real number:

-?((to-91+) I (10-91*\. 10-9]+) ([a - +] ? t o - 9 1 +) ?)

Our expression makes the exponent part optional. Let's write a real lexer that uses this expression. It examines the input and tells us each time it matches a number according to our regular expression.

4. In-lab Task:

```
Template :
#include<math.h>
    FILE *fp;
%}
Binary [0-1]+
…
…..
…..

%%
```

```
{Binary} {printf("this is an binary number : %s", yytext);}
…..
…..
%%

int main(int argc,char *argv[])
{
   fp=fopen(argv[1],"r");

   if(fp!=NULL)
   {
      yyin=fp;
      yylex();
   }
   return(0);
}
```

5. Post-lab Task:
Outcomes:
Able to Implement a lexical analyzer for regular expression

```
%{
#include<math.h>
    FILE *fp;
%}
Binary [0-1]+
Oct [0-7]+
Dec [0-9]+
Hex [0-9A-F]+
Float [-+]?[0-9]*\.[0-9]+
Exponent [-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?

%%
{Binary} {printf("this is an binary number : %s", yytext);}
{Oct} {printf("this is an octal number : %s", yytext);}
{Dec} {printf("this is a Decimal number : %s", yytext);}
{Hex} {printf("this is an hexadecimal number : %s",  yytext);}
{Float}  {printf("this is an Float number : %s",  yytext);}
{Exponent} {printf("this is an Exponent number : %s",  yytext);}
%%

int main(int argc,char *argv[])
{
   fp=fopen(argv[1],"r");

   if(fp!=NULL)
   {
      yyin=fp;
      yylex();
   }
   return(0);
}
```

Gangamai College of Engineering, Nagaon, Dhule
Department of Computer Engineering

Name: **Date of Performance: _ _ /_ _/20_ _**
Class: B.E. Computer **Date of Completion: _ _ /_ _/20_ _**
Div: **Batch: _ _ _ _ _**

Roll No:

Subject: Compiler Design lab **Sign. of Teacher with Date**

Experiment No. 3

Aim: To Implement a Calculator using LEX and YACC.

1. Objective: Student able to understand implementation of grammar rule by using a Calculator implementation through LEX and YACC.

2. Background:

Yacc takes a grammar that you specify and writes a parser that recognizes valid "sentences" in that grammar. We use the term "sentence" here in a fairly general way-for a C language grammar the sentences are syntactically valid C programs.
a grammar is a series of rules that the parser uses to recognize syntactically valid input. For example, here is a version of the grammar we'll use later in this chapter to build a calculator.

Statement □*NAME* = expression

Expression □*NUMBER* + *NUMBER* | *NUMBER* -*NUMBER*

The vertical bar, "|", means there are two possibilities for the same symbol,
i.e., an expression can be either an addition or a subtraction. The symbol to the left of the □is known as the left-hand side of the rule, often abbreviated LHS, and the symbols to the right are the right-hand side, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar is just a short hand for this. Symbols that actually appear in the input and are returned by the lexer are terminal symbols or tokens, while those that appear on the left-hand side of some rule are non-terminal symbols
or non-terminals. Terminal and non-terminal symbols must be different; it is an error to write a rule with a token on the left side.

3.Pre-lab Task:

A yacc grammar has the same **three-part** structure as a lex specification. (Lex copied its structure from yacc.) The first section, the **definition section**, handles control information for the yacc-generated parser (from here on we will call it the parser), and generally sets up the execution environment in which the parser will operate. The second section contains the **rules for the parser**, and the third section is **C code** copied verbatim into the generated C program.

Definition Section

The definition section includes declarations of the tokens used in the grammar,the types of values used on the parser stack, and other odds and ends.It can also include a literal block, C code enclosed in %{ %} lines. We start our first parser by declaring two symbolic tokens.

```
 %{
        #include<stdio.h>

%}

%union {
        double dval;
        int vblno;
        }

%token <vblno> NAME
%token <dval> NUMBER
%left '+''-'
%left '*''/'
%nonassoc uniminus
%type <dval> expression
```

You can use single quoted characters as tokens without declaring them, sowe don't need to declare "=", "+", or "-".

Rules Section

The rules section simply consists of a list of grammar rules in much thesame format as we used above. Since **ASCII** keyboards don't have a ☐ key, we use a colon between the left- and right-hand sides of a rule, and we put a semicolon at the end of each rule:

```
%%
statement:NAME '=' expression {$1=$3;}
         | expression {printf("%f",$1);}
         ;
expression: expression '+' expression {$$=$1+$3;}
          | expression '-' expression {$$=$1-$3;}
          | expression '*' expression {$$=$1*$3;}
```

```
           | expression '/' expression {
                                 if($3==0.0)
                                 {
                                         yyerror("divide by zero error");
                                 }
                                 else
                                         {$$=$1/$3;}
                                        }
           |'-'expression %prec uniminus {$$=-$2;}
           | '('expression')'{$$=$2;}
           | NUMBER{$$=$1;}


%%
```

C code

The parser reduces a rule, it executes user C code associated withthe rule, known as the rule's *action.* The action appears in braces after theend of the rule, before the semicolon or vertical bar. The action code can refer to the values of the right-hand side symbols as $1, **$2,** . . . , and can set the value of the left-hand side by setting $$. In our parser, the value of an *expression* symbol is the value of the expression it represents. We add some code to evaluate and print expressions, bringing our grammar up to that used:

```
int main()
{
        yyparse();
        return(0);
}
void yyerror(char *s)
{
        printf("%s",s);
}
```

The rules that build an expression compute the appropriate values, and the rule that recognizes an expression as a statement prints out the result. Inthe expression building rules, the first and second

numbers' values are **$1**and **$3,** respectively. The operator's value would be ***$2,*** although in this grammar the operators do not have interesting values. The action on the last rule is not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value $1 to $$.

The Lexer

To try out our parser, we need a lexer to feed it tokens. The parser is the higher level routine, and calls the lexer yylex() whenever it needs a token from the input. As soon as the lexer finds a token of interest to the parser, it returns to the parser, returning the token code as the value. Yacc defines the token names in the parser as C preprocessor names in *y.tab.h* (or some similar name on MS-DOS systems) so the lexer can use them.

```
%{
#include<math.h>
#include"y.tab.h"

%}
%%
[0-9]+ {yylval.dval=atoi(yytext); return NUMBER;}
[\t] ;
\n  return(0);
. return yytext[0];
```

```
                    %%
```

3. In-lab Task:

1. **Definition section**
2. **Rules for the parser**
3. **C Code**
4. **lexer**

5. Post-lab Task:

Outcomes:

Able to implement grammar for calculater using lex and Yacc

```
%{
/* Definition section */
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
              yylval=atoi(yytext);
              return NUMBER;

       }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()
{
return 1;
}
```

```
%{
/* Definition section */
#include<stdio.h>
int flag=0;
%}
%{
int yylex();
void yyerror();
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

/* Rule Section */
%%

ArithmeticExpression: E   {
                  printf("\nResult=%d\n", $$);
                  return 0;
                   };
```

```
            E:E'+'E {$$=$1+$3;}

             |E'-'E {$$=$1-$3;}

             |E'*'E {$$=$1*$3;}

             |E'/'E {$$=$1/$3;}

             |E'%'E {$$=$1%$3;}

             |'('E')' {$$=$2;}

             | NUMBER {$$=$1;}

            ;

%%

//driver code
void main()
{
printf("\nEnter Any Arithmetic Expression which can have operations
Addition,Subtraction, Multiplication,Divison,     Modulus and Round
brackets:\n");

yyparse();
if(flag==0)
printf("\nEntered arithmetic expression is Valid\n\n");
}

void yyerror()
{
printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
}
```

Gangamai College of Engineering, Nagaon, Dhule
Department of Computer Engineering

Name: **Date of Performance: _ _ /_ _/20_ _**
Class: B.E. Computer **Date of Completion: _ _ /_ _/20_ _**
Div: **Batch: _ _ _ _ _**

Roll No:

Subject : Compiler Design lab **Sign. of Teacher with Date**

Experiment No. 4

Aim: Implementation of Context Free Grammar

1. Objective: Students will be able to implement any context free grammar.

2. Background:

A context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to generate patterns of strings.

A CFG consists of the following components:

- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production). A☐ α where α is any combination of terminals or non-terminals.
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.

3.Pre-lab Task:

For any language given generate its equivalent context free grammer.

CFG for L= { 0^n1^n | n > 1} can be generated by using following production rules.

S ☐ 0S1 | 01

For implementing this CFG, lexical analyzer should pass the token i.e. terminal symbols of a language to parser.

Example : Lex file for above language can be given as :

```
%{
#include "y.tab.h"
%}

%%
"0" {return ZERO;}
"1" {return ONE;}
[\n] {return NL;}
. ;
%%
```

Parser code can be written as:

```
%token ONE ZERO NL

%%

str1: str2 n1 { }
    ;

str2:  ZERO str2 ONE { }
     | ZERO ONE { }
    ;

n1  : NL  { printf("\n The string is  accepted");return;}
    ;
%%
main()
{
        yyparse();
}
```

3. In-lab Task:

1. Create lex file (_____.l) with definitions of required language. Tokens must be of the terminal symbols of CFG.
2. Create parser file (______.y) with the actual production rules of CFG.

4. Post-lab Task:

Outcomes:
Able to implement of Context Free Grammar for CFL.

```
%{
#include "y.tab.h"
%}

%%
"0" {return ZERO;}
"1" {return ONE;}
[\n] {return NL;}
. ;

%%
```

```
/* Definition section */
#include<stdio.h>
int flag=0;
%}
%{
```

```
int yylex();
void yyerror();
%}
%token ONE ZERO NL



/* Rule Section */
%%
str1: str2 n1 { }
;
str2:
ZERO str2 ONE { }
| ZERO ONE { } ;
n1 : NL   {return(0);} ;

%%

//driver code
void main()
{
printf("\nEnter  string (any combination of 0 and 1)\n");yyparse();
if(flag==0)
printf("\nEntered string  is Valid for L=[0^n1^n]\n\n");
}

void yyerror()
{
printf("\nEntered arithmetic is Invalid for L=[0^n1^n]\n\n");
flag=1;
}
```

Gangamai College of Engineering, Nagaon, Dhule
Department of Computer Engineering

Name: **Date of Performance: _ _ /_ _/20_ _**
Class: B.E. Computer **Date of Completion: _ _ /_ _/20_ _**
Div: **Batch: _ _ _ _ _**

Roll No:

Subject: Compiler Design lab **Sign. of Teacher with Date**

Experiment No. 5

Aim: Implementation of Code Generator.

1. Objective: To understand and simulate code generation phase of compiler.

2. Background:

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

It should carry the exact meaning of the source code.
It should be efficient in terms of CPU usage and memory management

Code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language:** The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type:** Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction:** The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation:** A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- **Ordering of instructions:** At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

3. Pre-lab Task:

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way.

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

- Instruction selection: which instructions to use.
- Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.
- Register allocation: the allocation of variables to processor registers
- Debug data generation if required so the code can be debugged.

Instruction selection is typically carried out by doing a recursive postorder traversal on the abstract syntax tree, matching particular tree configurations against templates; for example, the tree W := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.

In a compiler that uses an intermediate language, there may be two instruction selection stages — one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the instruction set of the target machine. This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding opcodes. However, if the compiler is actually a language translator (for example, one that converts Eiffel to C), then the second code-generation phase may involve *building* a tree from the linear intermediate code.

4. In-lab Task:

Lex program Template :

```
%{

    - - - - - - - - - - - -
     - - - - - - - - - - - -

%}


%%
 - - - - - - - - - - - - |
  - - - - - - - - - - - {
 strcpy(yylval.vname,yytext);

                              return NAME;

                           }

 - - - - - - - - - - - -       ;


\n                    { return yytext[0]; }


%%
```

Yacc Program Template:

```
%{
- - - - - - - - - - - -
- - - - - - - - - - - -
- - - - - - - - - - - -
%}

%union
{
       - - - - - - - - - - - -
}


%left '+' '-'
%left '*' '/'
%token <vname> NAME
%type <vname> expression
%%

 input : line '\n' input
         | '\n' input
         | ;
 line    : NAME '=' expression {
                                fprintf(fpOut,"MOV %s, AX\n",$1);
                                      }
       ;

expression-------------------------{

                               fprintf(.................................);

                               fprintf(................................);

                                      }


                |- - - - - - - - - - - -


                |- - - - - - - - - - - -        {

                 | NAME '/' NAME        {

                               fprintf(fpOut,"MOV AX, %s\n",$1);

                               fprintf( ---------------------------------);

                                      }
                | NAME                {

                               fprintf(fpOut,"MOV AX, %s\n",$1);
                               strcp$$, $1y();
                                      }
                ;
%%

FILE *yyin;
main()
{
      FILE *fpInput;
       fpInput = fopen(" ---------,"r");
       fpOut = fopen(" ------------ ","w");
```

```
        yyin = fpInput;
        yyparse();
        fcloseall();

}
```

5. Post-lab Task:
Outcomes:
Able to implement of Code Generator phase of complier using LEX and YACC.

```
%{
#include"stdio.h"
#include"y.tab.h"
%}
%%
[a-z][a-zA-Z0-9]* |
[0-9]+                  {
                                strcpy(yylval.vname,yytext);
                                return NAME;
                        }

[ |\t]                  ;

.|\n                    { return yytext[0]; }

%%
```

```
#include"stdio.h"
#include"string.h"
#include"stdlib.h"
int yylex();
void yyerror();
void fcloseall();

FILE *fpOut;

%}

%union
{
        char vname[10];
        int val;
}

%left '+' '-'
%left '*' '/'

%token <vname> NAME
%type <vname> expression

%%

input   : line '\n' input
          | '\n' input
          | ;
```

```
line    : NAME '=' expression  {
                                   fprintf(fpOut,"MOV  %s, AX\n",$1);
                                          }
        ;
expression: NAME '+' NAME      {
                                   fprintf(fpOut,"MOV AX, %s\n",$1);
                                   fprintf(fpOut,"ADD  AX, %s\n",$3);
                                          }

                  | NAME '-' NAME         {
                                   fprintf(fpOut,"MOV AX, %s\n",$1);
                                   fprintf(fpOut,"SUB AX,  %s\n",$3);
                                          }

                  | NAME '*' NAME         {
                                   fprintf(fpOut,"MOV AX, %s\n",$1);
                                   fprintf(fpOut,"MUL AX,  %s\n",$3);
                                          }

                  | NAME '/' NAME         {
                                   fprintf(fpOut,"MOV AX, %s\n",$1);
                                   fprintf(fpOut,"DIV AX,  %s\n",$3);
                                          }
                  | NAME                  {
                                   fprintf(fpOut,"MOV AX, %s\n",$1);
                                   strcpy($$, $1);
                                          }
                 ;

%%

FILE *yyin;
int main()
{
        FILE *fpInput;

        fpInput = fopen("input.txt","r");
        if(fpInput=='\0')
        {
                printf("file read error");
                exit(0);
        }
        fpOut = fopen("output.txt","w");
            //printf("%s",msg);
        if(fpOut=='\0')
        {
                printf("file creation error");
                exit(0);
        }
        yyin = fpInput;
        yyparse();
        void fcloseall();
        return 0;
}


void yyerror(){}
```

Gangamai College of Engineering, Nagaon, Dhule
Department of Computer Engineering

Name: **Date of Performance: _ _ /_ _/20_ _**
Class: B.E. Computer **Date of Completion: _ _ /_ _/20_ _**
Div: **Batch: _ _ _ _ _**

Roll No:

Subject: Compiler Design lab **Sign. of Teacher with Date**

Experiment No. 6

Aim: Implementation of Deterministic Finite Automaton.

1. Objective: To study and demonstrate the concept of deterministic finite automata.

2. Background:
A finite automaton is a state machine that takes a string of symbols as input and changes its state accordingly. A finite automaton is a recognized for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reach its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:
•Finite set of states (Q)
•Finite set of input symbols (Σ)
•One Start state (q0)
•Set of final states (qf)
•Transition function (δ)

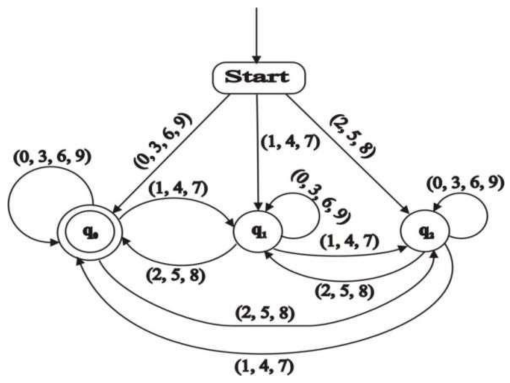
The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).
•States: States of FA are represented by circles. State names are written inside circles.
•Start state: The state from where the automata starts, is known as the start state.Start state has an arrow pointed towards it.
•Intermediate states: All intermediate states have at least two arrows; one pointing to and another pointing out from them.

•Final state: If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. odd = even+1.

•Transition: The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next. state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state.
If automata stay on the same state, an arrow pointing from a state to itself is drawn.
Example : We assume FA accepts decimal number divisible by three . FA = {Q(Start, q_0,q_1,q_2), Σ(0 to 9), q0, qf, δ}



3. Pre-lab Task:
Step1. State identification using lex
Step2. Prepare production rule for problem e.g. any number divisible by 3
Step3. Implement parser program (YACC) for step 2

4. In-lab Task:

Template for LEX program:

```
%{
-----------------------

%}

%%
[ ------------------- ] {return ZERO;}
[ ------------------- ] {return ONE;}
[ ------------------- ] {return TWO;}
[\n] {return NL;}
. ;
%%
```

Template for YACC Program :

```
%{
/* Definition section */
#include<stdio.h>
int flag=0;
%}
%{
int yylex();
void yyerror();
%}
%token ZERO ONE TWO NL
%start q0

%%


q0 : ZERO q0 {$$ = $2;}
        |       ONE q1 {$$ = $2;}
        |       TWO q2 {$$ = $2;}
        | NL {printf("Number is divisible by 3\n"); return(0);}
        ;
q1 : ZERO q1 {$$ = $2;}
        | ONE q2 {$$ = $2;}
        | TWO q0 {$$ = $2;}
      | NL {printf("Number is not divisible by 3 reminder is 1 \n"); return(0);}
        ;

q2: ZERO q2 {$$ = $2;}
        | ONE q0 {$$ = $2;}
        | TWO q1 {$$ = $2;}
        |NL {printf("Number is not divisible by 3 reminder is 2 \n"); return(0);}
        ;

%%
void main()
{
printf("\n Enter Decimal number to check divisibility by 3:\n");
yyparse();
}
```

Command for Execution:

```
$lex file_name.l
$yacc -d file_name.y
$cc lex.yy.c y.tab.c -ll –ly
$/a.out
```

5. Post-lab Task:

Outcomes:
Able to implementation of Deterministic Finite Automaton using LEX and YACC.

```
%{
#include "y.tab.h"
%}

%%
[0369] {return ZERO;}
[147] {return ONE;}
[258] {return TWO;}
[\n] {return NL;}
. ;
%%
```

```
/* Definition section */
#include<stdio.h>
int flag=0;
%}
%{
int yylex();
void yyerror();
%}
%token ZERO ONE TWO NL
%start q0

%%

q0 : ZERO q0 {$$ = $2;}
      |     ONE q1 {$$ =  $2;}
      |     TWO q2 {$$ =  $2;}
      | NL {printf("Number is divisible by 3\n"); return(0);}
      ;
q1 : ZERO q1 {$$ = $2;}
      | ONE q2 {$$ =  $2;}
      | TWO q0 {$$ =  $2;}
         | NL {printf("Number is  not divisible by 3 reminder is 1
\n"); return(0);}
      ;

q2: ZERO q2 {$$ = $2;}
      | ONE q0 {$$ =  $2;}
      | TWO q1 {$$ =  $2;}
      |NL {printf("Number is  not divisible by 3 reminder is 2 \n");
return(0);}
      ;

%%

void main()
{
printf("\n Enter Decimal number to check divisibility by 3:\n");
yyparse();
```