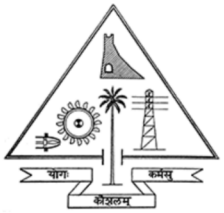
LAB MANUAL

CS431 (P) COMPILER DESIGN LAB

B.Tech VII Semester Computer Science and Engineering

(APJ Abdul Kalam Technological University)



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GOVERNMENT ENGINEERING COLLEGE

THRISSUR

Vision and Mission of the institute

Vision

To be a premier institution of excellence in engineering education and research for sustainable development

Mission

(i) Provide quality education in Engineering and Technology
(ii) Foster passion for research
(iii) Transform the students into committed technical personnel for the social and economic wellbeing of the nation

Department of Computer Science and Engineering

Vision

To become a centre of excellence and a pioneering research centre in Computer Science and Engineering.

Mission

To provide quality education and training to the students, through academic and research oriented activities in Computer Science and Engineering, for transforming them to committed technical personnel, which can contribute to the social and economic betterment of the society.

Program Educational Objectives (PEOs)

1. To empower students to identify, formulate and solve computing problems by applying their knowledge in Mathematics, Theoretical Computing and Computer Programming.

2. To develop industry focused skills and leadership qualities to become successful engineers and entrepreneurs.

3. To enable students to acquire skills to communicate effectively with the society and the constituents which enable them to collaborate as team members and team leaders.

4. To instill professional work ethics and social responsibilities so that they can contribute to the betterment of the society as committed technical personnel.

5. To inculcate a passion towards higher education, research and lifelong learning in the field of Computer Science and Engineering

Program Outcomes (POs)

PO1 : **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2 : **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4 : **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7 : **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice

PO9 : Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10 : **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change..

CONTENTS

Curriculum

Syllabus

Teaching scheme: 3 hours practical per week

List of Exercises/Experiments :

1. Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.

2. Implementation of Lexical Analyzer using Lex Tool

3. Generate YACC specification for a few syntactic categories.

a) Program to recognize a valid arithmetic expression that uses operator +, – , * and /.

b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

c) Implementation of Calculator using LEX and YACC

d) Convert the BNF rules into YACC form and write code to generate abstract syntax tree

4. Write program to find ε – closure of all states of any given NFA with ε transition.

5. Write program to convert NFA with ε transition to NFA without ε transition.

6. Write program to convert NFA to DFA

7. Write program to minimize any given DFA.

8. Develop an operator precedence parser for a given language.

9. Write program to find Simulate First and Follow of any given grammar.

10. Construct a recursive descent parser for an expression.

11. Construct a Shift Reduce Parser for a given language.

12. Write a program to perform loop unrolling.

13. Write a program to perform constant propagation.

14. Implement Intermediate code generation for simple expressions.

15. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.

References

1. Sethi R., Programming Languages: Concepts and Constructs, Addison Wesley

2. Appleby D. & Vandekopple J.J., Programming Languages: Paradigm and Practice, Tata McGraw Hill

3. Luger & Stubblefield, Artificial Intelligence, Addison Wesley

4. Samuel A. Rebelsky, Experiments in Java, Pearson Education.

5. Lex and Yacc John R Levine,Tony Mason Doug Brown.

Assessment

Internal Continuous Assessment (Maximum Marks-100)

60%-Laboratory practical and record

10%- Regularity in the class

30%- Semester End Examination (Maximum Marks-30)

Requirements

2.1 Hardware Requirements

PC with Pentium-4 processor

- Clock : 1.7 GHz
- Main Memory : 256 Mbytes
- Cache Memory (L2) : 256 Kbytes
- Hard Disk : 40 Gbytes

2.2 Software Requirements

- Operating System: Windows/Ubuntu 14.04 LTS
- Programming language: C
- Compiler : gcc
- LEX, YACC / BISON.

Course Summary

3.1 Prerequisite and Course Description

The intention of this course is to build an understanding on design and implementation of compiler. CS331 System Software Lab is the prerequisite. However introductory knowledge in programming is desirable.

3.2 Course Objectives

- To implement the different Phases of compiler.
- To implement and test simple optimization techniques.
- To give exposure to compiler writing tools.

3.3 Course Outcome (COs)

The Student will be able to :

CO1. Implement the techniques of Lexical Analysis and Syntax Analysis.

CO2. Apply the knowledge of Lex & Yacc tools to develop programs.

CO3. Generate intermediate code.

CO4. Implement Optimization techniques and generate machine level code.

3.4 Course Plan and CO Mapping

Week	Subject and Content	CO Mapping
Week 1	Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.	CO1
Week 2	Implementation of Lexical Analyzer using Lex Tool.	CO2
Week 3	Generate YACC specification for a few syntactic categories. a.Program to recognize a valid arithmetic expression that uses operator +, –, * and /. b.Program to recognize a valid variable which starts with a letter followed by any number of letters or digits. c.Implementation of Calculator using LEX and YACC. d.Convert the BNF rules into YACC form and write code to generate abstract syntax tree.	CO2

Week 4	Write program to find ε – closure of all states of any given NFA with ε transition.	CO1
Week 5	Write program to convert NFA with ε transition to NFA without ε transition..	CO1
Week 6	Write program to convert NFA to DFA.	CO1
Week 7	Write program to minimize any given DFA.	CO1
Week 8	Develop an operator precedence parser for a given language	CO2
Week 9	Write program to find Simulate First and Follow of any given grammar.	CO1
Week 10	Construct a recursive descent parser for an expression.	CO1
Week 11	Construct a Shift Reduce Parser for a given language	CO1
Week 12	Write a program to perform loop unrolling.	CO2
Week 13	Write a program to perform constant propagation.	CO2
Week 14	Implement Intermediate code generation for simple expressions.	CO3
Week 15	Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.	CO4

Cycle of Experiments

1. Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.
2. Implementation of Lexical Analyzer using Lex Tool.
3. Generate YACC specification for a few syntactic categories.
    a. Program to recognize a valid arithmetic expression that uses operator +, –, * and /.
    b. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
    c. Implementation of Calculator using LEX and YACC.
    d. Convert the BNF rules into YACC form and write code to generate abstract syntax tree.
4. Write program to find ε – closure of all states of any given NFA with ε transition.
5. Write program to convert NFA with ε transition to NFA without ε transition.
6. Write program to convert NFA to DFA.
7. Write program to minimize any given DFA.
8. Develop an operator precedence parser for a given language.
9. Write program to find Simulate First and Follow of any given grammar.
10. Construct a recursive descent parser for an expression.
11. Construct a Shift Reduce Parser for a given language.
12. Write a program to perform loop unrolling.
13. Write a program to perform constant propagation.
14. Implement Intermediate code generation for simple expressions.
15. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.

INTRODUCTION TO COMPILERS

A Compiler is a program that translates a high-level language program into a functionally equivalent low-level language program. So a compiler is a translator that translates a high level-language into a low level-language. A typical compilation broken down into phases as a shown below.

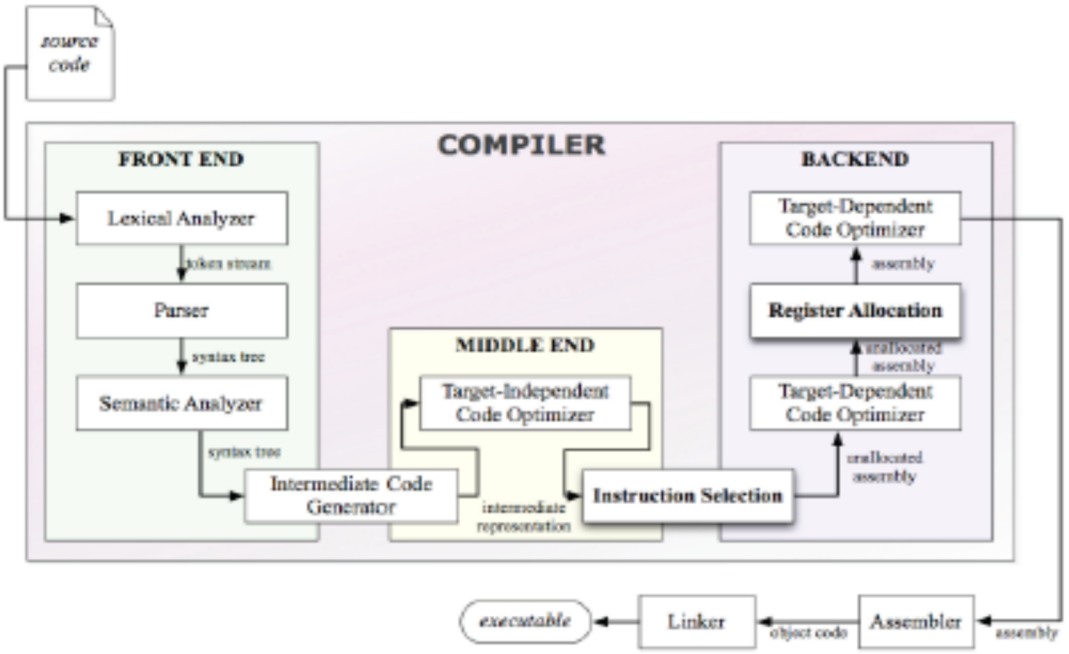


Figure 1: Structure of a typical compiler[2]

Figure 1. Phases of a compiler

Lexical analysis: The Lexical analysis phase reads the characters in the source program and groups them into streams of tokens; each token represents a logically cohesive sequence of characters, such as identifiers, operators, and keywords. The character sequence that forms a token is called a "lexeme."

Parser: Parser imposes a hierarchical structure on the token stream which is called as syntax tree. Parser check the given sentence is grammatically correct one and well formed and checking the sentence belongs to the language of the grammar.

Semantic analysis: In this phase the compiler connect variable definition to their uses, checks that each expression has a correct type and translate the abstract syntax into a simpler representation suitable for generating machine code.

Intermediate Code generation: In this phase a compiler generate an explicit intermediate representation of the source program. It should be easy to produce and easy to translate into the target program.

Code Optimization: The code optimization phase attempt to improve the intermediate code so that faster scanning machine code will result.

Code Generation: Last phase of compiler generation of target code, may be relocatable machine code or assembly code.

Symbol table: A data structure containing a record for each identifier with field for attributes of the identifier.

Error Handler: This will deals with the detected and reported error in each phase of the compiler.

The compilation process is illustrated in the figure given below.

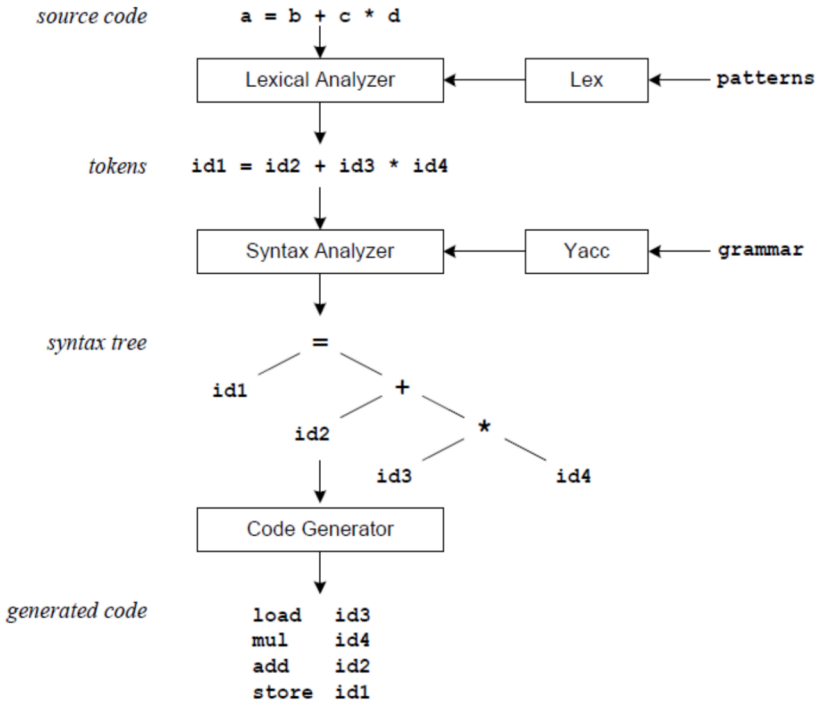


Figure 1: Compilation Sequence

Figure 2. Compiler sequence

Compiler Construction tools

Compiler Construction tools are software tools used by the compiler writers to construct a compiler. These tools are specialized languages for specifying and implementing the component. Some of the compilers construction tools are classified as

1. Scanner Generators
2. Parser generators
3. Syntax- directed translation engines
4. Automatic code generators
5. Data Flow engines

Lex and Yacc are two compiler construction tools belong to scanner generator and parser generator respectively.

INTRODUCTION TO LEX AND YACC

We code patterns and input them to lex. It will read the patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on patterns written in the input file, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. The translation using Lex is shown below.

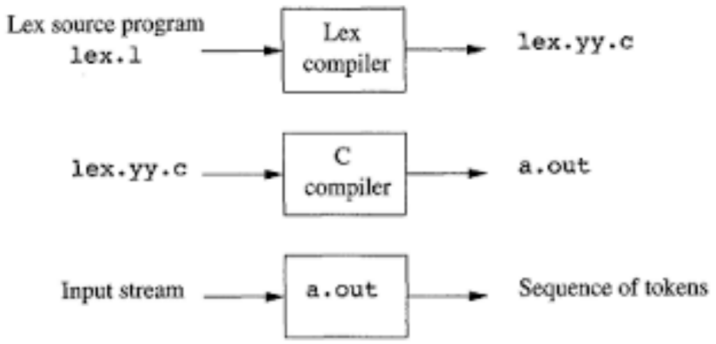


Figure 3. The translation using Lex

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

We code a grammar and input it to Yacc. Yacc will read the grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. The translation using Yacc as shown below. The below figure illustrate combined use and the file naming convention used by lex and yacc, such as "bas.y" for yacc compiler and "bas.l" for lex compiler.

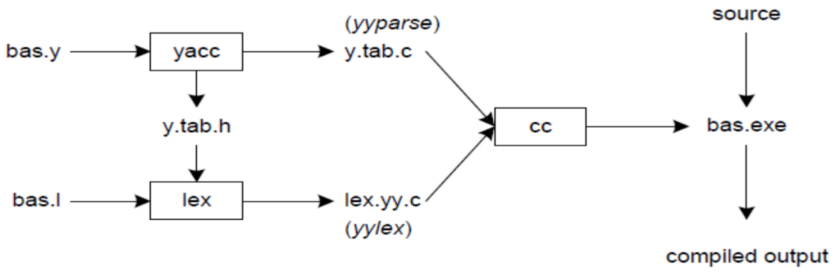


Figure 2: Building a Compiler with Lex/Yacc

Figure 4. Lex and YACC compilers

First, we need to specify all pattern matching rules for lex (**bas.l**) and grammar rules for yacc (**bas.y**). Commands to create our compiler, **bas.exe**, are listed below:

```
yacc  –d   bas.y                          # create y.tab.h, y.tab.c
lex  bas.l                                # create lex.yy.c
gcc -o bas  y.tab.c  lex.yy.c  -ll        # compile/link (output file name 'bas.exe')
```

Yacc reads the grammar descriptions in **bas.y** and generates a syntax analyzer (parser), that includes function **yyparse**, in file **y.tab.c**. Included in file **bas.y** are token declarations. The **–d** option causes yacc to generate definitions for tokens and place them in file **y.tab.h**. Lex reads the pattern descriptions in **bas.l**, includes file **y.tab.h**, and generates a lexical analyzer, that includes function **yylex**, in file **lex.yy.c**.

Finally, the lexer and parser are compiled and linked together to form the executable, **bas.exe**. From **main**, we call **yyparse** to run the compiler. Function **yyparse** automatically calls **yylex** to obtain each token.

More about Lex:

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

Letter (letter | digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.

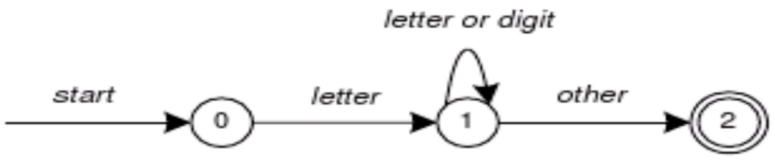


Figure 5. Finite State Automaton

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimic an FSA. Regular expressions in lex are composed of meta characters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class, normal operators lose their meaning.

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

Table 2: Pattern Matching Examples

Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match

the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

Lex Format:

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to Lex is divided into three sections, with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file.

```
%%
```

Input is copied to output, one character at a time. The first %% is always required, as there must always be a rules section. However, if we don't specify any rules, then the default action is to match everything and copy it to output. Defaults for input and output are stdin and stdout, respectively.

Here is the same example, with defaults explicitly coded.

```
%%
      /* match everything except newline */
.     ECHO;
      /* match newline */
\n    ECHO;
%%
int  yywrap(void)
    {
     return 1;
     }
int  main(void)
    {
     yylex();
     return 0;
     }
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline), and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not starting in column one is copied exactly to the generated C file (Comments can copied from lex file to C file). In this example there are two patterns, "." and "\n", with an ECHO action associated for each pattern. Several macros

and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as

#define ECHO fwrite(yytext, yyleng, 1, yyout)

Variable yytext is a pointer to the matched string (NULL-terminated), and yyleng is the length of the matched string. Variable yyout is the output file, and defaults to stdout. Function yywrap is called by lex when input is exhausted. Return 1 if you are done, or 0 if more processing is required. Every C program requires a main function. In this case, we simply call yylex, the main entry-point for lex. Some implementations of lex include copies of main and yywrap in a library, eliminating the need to code them explicitly.

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables

yywrap():

This function is called when end of file(or input) is encountered. If it return 1, the parsing stops. So this can be used to parse multiple files. Code can be written in third section, which will allow multiple file to be parsed.

yylineno: Provides current line number information.

Sample Program: Lexical Analyzer for C programming language

```
letter  [a-zA-Z]
digit  [0-9]
%%
{digit}+("E"("+"|"-")?{digit}+)?       printf("\n%s\tis real number",yytext);

{digit}+"."{digit}+("E"("+"|"-")?{digit}+)?  printf("\n%s\tis a float
number",yytext);

"if"|"else"|"int"|"float"|"switch"|"case"|"struct"|"char"|"return"|"for"|"do"|"while"|"void"|"pri
ntf"|"scanf"      printf("\n%s\tis a keyword",yytext);
```

```
"\t"|"\b"|"\\n"|"\\t"|"\\a"|"\\b"|"\a"   printf("\n%s\tis an escape sequences",yytext);

{letter}({letter}|{digit})*       printf("\n%s\tis an identifier",yytext);

"["|"]"|"{"|"}"|"("|")"|"#"|"'"|"''"|"\"|"\\"|";"|"," printf("\n%s\t is a special
character",yytext);

"&&"|"<"|">"|"<="|">="|"+"|"="|"-"|"?"|"/"|"||"|"*"|"&"|"%" printf("\n%s\t is an
operator",yytext);

"%d"|"%s"|"%c"|"%f"|"%e"   printf("\n%s\tis a format specifier",yytext);

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

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it find more than one match, it take the one matching the most text. If it find two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined which satisfying one of the regular expression or rule, the text corresponding to the match (token) is made available in the global character pointer yytext and its length in the global integer yyleng. The action corresponding to the matched pattern is then executed and then the remaining input is scanned for another match. If no match is found, then the default rule is executed.

YACC (Yet Another Compiler Compiler)

YACC is an LALR parser generator. The first version of YACC was created by S.C Johnson. Yacc translate a yacc file into a C file y.tab.c using LALR method

A Yacc source program has three parts:

```
Declarations
%%
Translation rules
%%
Supporting C- routines
```

The Declarations parts:

There are two optional sections in the declarations part of the Yacc program. In the first section put ordinary C declarations, delimited by %{ and %}. Here we place any temporaries used by the translation rules or procedures of the second and third sections. Also in the declarations part are declarations of grammar tokens.

Each grammar rule defines a symbol in terms of: Other symbols (Non terminals) and tokens (terminals) which come from the lexical analyzer.

Terminal symbols are of three types:

Named token: Defined via the %token identifier. By convention these are all uppercase

Character token: As same as character constant written in c language, Eg: + is a character

Constant: Literal string constant. Like C string constant Eg: "abc" is string constant.

The Lexer returns named tokens. Non terminals are usually represented by using lower case letters.

A few Yacc specific declarations which begins with a % sign in Yacc specification file

1) %union. It defines the stack type for the parser. It is union of various data/ structures/ objects
2) %token. These are terminals return by the yylex function to the yacc.

   %token NUM NUM is a named token
3) %type. The type of non-terminal symbol in the grammar can be specified by using this rule.
4) %nonassoc. Specifies that there is no associativity of a terminal symbol

   %nonassoc ′<′ no associativity
5) %left. Specifies that there is left associativity of a terminal symbol

   %left ′+′ ′ - ′ make + and – be the same precedence and left associative
6) %right. Specifies that there is right associativity of a terminal symbol

   %right ′+′ ′- ′ make + and – be the same precedence and right associative
7) %start. Specifies that the L.H.S non terminal symbol of production rule which should be taken as the starting point of the grammar rules.
8) %prec. Change the precedence level associated with a particular rule to that of the following token name or literal. %prec <terminal>

%left ′+′ ′ - ′

%right ′*′ ′/′

The above two statements declare the associativity and precedence exist among the tokens. The precedence of the token increases from top to bottom, so the bottom listed rule

has the highest precedence compared to the rule listed above it. Here * and / have the higher precedence than the addition and subtraction.

Translation rules part:

After the first %% put the translation rules. Each rule consists of a grammar production and the associated semantic action. Consider this production

<Left_side> → <alt 1>| <alt 2>|.....|<alt n>

would be written in yacc as

```
<left_side>     :     <alt 1> { semantic action 1}
                |     <alt 2> { semantic action 2}
….....................................
                |     <alt n> { semantic action n}
                ;
```

In Yacc production, quoted single character ′ c ′ is taken to be the terminal c and unquoted strings letter and digits not declare to be token to be taken as nonterminals.

The Yacc semantic action is a sequence of C statements, In a semantic action the symbol $$ refers to the attribute value associated with non terminals on the left, while $i refers to the value associated with the i^{th} grammar symbol(terminal or non terminals) on the right. The semantic action is performed whenever we reduce by the associated production, So normally the semantic action computes a value for $$ in terms of the $i′s.

Example: Here the non-terminal term in the first production is the third grammar symbol on the right, while ′+′ is the second terminal symbol. We have omitted the semantic action for the second production, since copying the value is the default action for productions with a single grammar symbol on the right. ($$= $1) is the default action.

Supporting C routines part:

The third part of the yacc specification consists of supporting C- routines. A lexical analyzer by the name yylex() must be provided . Other procedures such as error recover routine may be added as necessary. The lexical analyzer yylex() produce pair consists of a token and its associated attribute value. The attribute value associated with a token is communicated to the parser through a yacc defined variable yylval.

How to compile Yacc and Lex files

1. Write a parser for a given grammar in a .y file
2. Write the lexical analyzer to process input and pass tokens to the parser when it needed, save the file as .l file
3. write error handler routine(like yyerror())

4. Compile .y file using the command yacc -d <filename.y> then it generate two file y.tab.c amd y.tab.h. If you are using bison, it generate <filename>.tab.c and <filename>.tab.h respectively
5. Compile lex file using the command flex <filename.l> it generate lex.yy.c
6. Compile the c file generated by the yacc compiler using gcc y.tab.c -ll it will generate .exe file a.out
7. Run the executable file and give the necessary input

History of lex and Yacc

Bison is descended from yacc, a parser generator written between 1975 and 1978 by Stephen C. Johnson at Bell Labs. As its name, short for "yet another compiler compiler," suggests, many people were writing parser generators at the time. Johnson's tool combined a firm theoretical foundation from parsing work by D. E. Knuth, which made its parsers extremely reliable, and a convenient input syntax. These made it extremely popular among users of Unix systems, although the restrictive license under which Unix was distributed at the time limited its use outside of academia and the Bell System.

In about 1985, Bob Corbett, a graduate student at the University of California, Berkeley, re-implemented yacc using somewhat improved internal algorithms, which evolved into Berkeley yacc. Since his version was faster than Bell's yacc and was distributed under the flexible Berkeley license, it quickly became the most popular version of yacc. Richard Stallman of the Free Software Foundation (FSF) adapted Corbett's work for use in the GNU project, where it has grown to include a vast number of new features as it has evolved into the current version of bison.

Bison is now maintained as a project of the FSF and is distributed under the GNU Public License. In 1975, Mike Lesk and summer intern Eric Schmidt wrote lex, a lexical analyzer generator, with most of the programming being done by Schmidt. They saw it both as a standalone tool and as a companion to Johnson's yacc. Lex also became quite popular, despite being relatively slow and buggy. (Schmidt nonetheless went on to have a fairly successful career in the computer industry where he is now the CEO of Google.) In about 1987, Vern Paxson of the Lawrence Berkeley Lab took a version of lex written in rat for (an extended Fortran popular at the time) and translated it into C, calling it flex, for "Fast Lexical Analyzer Generator." Since it was faster and more reliable than AT&T lex and, like Berkeley yacc, available under the Berkeley license, it has completely supplanted the original lex. Flex is now a SourceForge project, still under the Berkeley license.

A flex program consists of three sections, separated by %% lines. The first section contains declarations and option settings. The second section is a list of patterns and actions, and the third section is C code that is copied to the generated scanner, usually small routines related to the code in the actions. In the declaration section, code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file. In this case it just sets up variables for lines, words, and characters.

Experiments

Experiment No:1

Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.

Lex.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
 int isKeyword(char buffer[]){
   char keywords[32][10] = {"auto","break","case","char","const","continue","default",
                            "do","double","else","enum","extern","float","for","goto",
                            "if","int","long","register","return","short","signed",
                            "sizeof","static","struct","switch","typedef","union",
                            "unsigned","void","volatile","while"};
   int i, flag = 0;
   for(i = 0; i < 32; ++i){
      if(strcmp(keywords[i], buffer) == 0){
         flag = 1;
         break;
      }
   }
   return flag;
}
int main(){
   char ch, buffer[15], operators[] = "+-*/%=";
   FILE *fp;
   int i,j=0;
   fp = fopen("prg1.txt","r");
   if(fp == NULL){
      printf("error while opening the file\n");
      exit(0);
   }
   while((ch = fgetc(fp)) != EOF){
         for(i = 0; i < 6; ++i){
            if(ch == operators[i])
```

```
            printf("%c is operator\n", ch);
        }
        if(isalnum(ch)){
            buffer[j++] = ch;
        }
        else if((ch == ' ' || ch == '\n') && (j != 0)){
            buffer[j] = '\0';
            j = 0;
            if(isKeyword(buffer) == 1)
                printf("%s is keyword\n", buffer);
            else
                printf("%s is indentifier\n", buffer);
        }
    }

    fclose(fp);
    return 0;
}
```

Input.c

```
int a,b
a+b=0;
```

Experiment No:2

Implementation of Lexical Analyzer using Lex Tool

Lex.l

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
"int" | float | char | double | while | for | struct | typedef | do | "if" | break | continue | void |
switch | return | else | goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;} {printf("\n\t %s is a COMMENT",yytext);}
\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
\}  {if(!COMMENT)printf("BLOCK ENDS ");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\( {if(!COMMENT)printf("\n\t %s is an OPENING BRACKET",yytext);}
\) {if(!COMMENT)printf("\n\t %s is an CLOSING BRACKET",yytext);}
\<= | \>= |\< | == | \> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
\+ | \- | \\ | \*          {if(!COMMENT) printf("\n\t%s is a OPERATOR",yytext);}
%%
int main()
{
FILE *file;
file=fopen("prg2.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
```

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

Lex.c

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
if(a>b)
{
    b=a+1;
else
    b=1;
}
pintf("hello");
}
```

Sample Input & Output:

```
#include<stdio.h> is a preprocessor directive
#include<conio.h> is a preprocessor directive
        void is a keyword
 main IDENTIFIER
         ( is an OPENING BRACKET
         ) is an CLOSING BRACKET
 BLOCK BEGINS
        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;
        if is a keyword
         ( is an OPENING BRACKET
 a IDENTIFIER
```

```
        > is a RELATIONAL OPERATOR
b IDENTIFIER
        ) is an CLOSING BRACKET


BLOCK BEGINS
b IDENTIFIER
        = is an ASSIGNMENT OPERATOR
a IDENTIFIER
        + is a OPERATOR
1 is a NUMBER ;
        else is a keyword
b IDENTIFIER
        = is an ASSIGNMENT OPERATOR
1 is a NUMBER ;
BLOCK ENDS
pintf IDENTIFIER
        ( is an OPENING BRACKET
        "hello" is a STRING
        ) is an CLOSING BRACKET;
BLOCK ENDS
        /* is a COMMENT    /
```

Experiment No:3

Generate YACC specification for a few syntactic categories.

a. **Program to recognize a valid arithmetic expression that uses operator +, –, * and /.**

File.c

```
%{
#include<stdio.h>
#include<ctype.h>
%}
%token LETTER DIGIT
%left '+' '-'
%left '*' '/'
%%
st:st expr '\n' {
   printf("\n valid expression \n");
}
| st '\n' | ;
expr: expr '+' expr | expr '-' expr |expr '*' expr | expr '/' expr |' (' expr ')' | NUM| LETTER;
NUM: DIGIT| NUM DIGIT;
%%
int yylex()
{
char c;
while((c=getchar())==' ');
if(isalpha(c)) return LETTER;
if(isdigit(c)) return DIGIT;
return(c);
}
int main()
{
   printf("\n enter an expression\n");
   yyparse();
}
int yyerror()
{
   printf("invalid\n");
   return 0;
```

```
}
```

File.l

```
%{ #include "y.tab.h" %}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?      return num;
[+/*]                   return op;
.                      return yytext[0];
\n                      return 0;
%%
int yywrap()
{return 1;}
```

File.y

```
%{
   #include<stdio.h>
   int valid=1;
%}
%token num id op
%%
start : id '=' s ';'
s :     id x| num x
      | '-' num x| '(' s ')' x;
x :     op s| '-' s| ;
%%
int yyerror()
{
   valid=0;
   printf("\nInvalid expression!\n");
   return 0;
}
int main()
{
   printf("\nEnter the expression:\n");
   yyparse();
   if(valid)
   {printf("\nValid expression!\n");}
```

```
}
```

b. **Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.**

File.l

```
%{#include "y.tab.h" %}
%%
[a-zA-Z_][a-zA-Z_0-9]* return letter;
[0-9]                  return digit;
.                   return yytext[0];
\n                   return 0;
%%
int yywrap()
{return 1;}
```

File.y

```
%{
   #include<stdio.h>
   int valid=1;
%}
%token digit letter
%%
start : letter s
s :     letter s| digit s| ;
%%
int yyerror(){
   printf("\nIts not a identifier!\n");
   valid=0;
   return 0;
}
int main()
{
   printf("\nEnter a name to tested for identifier ");
   yyparse();
   if(valid)
   {printf("\nIt is a identifier!\n");}
}
```

c. **Implementation of Calculator using LEX and YACC.**

File.l

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{return 1; }
```

File.y

```
%{
   #include<stdio.h>
   int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{
        printf("Res=%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;} ;
%%
```

```
void main()
{
  printf("\nEnter expression:");
  yyparse();
}
void yyerror()
{flag=1; }
```

d. **Convert the BNF rules into YACC form and write code to generate abstract syntax tree.**

File.c

```
main()
{
        int a,b,c;
        if(a<b)
                a=a+b;
        while(a<b)
                a=a+b;
        if(a<=b)
                c=a-b;
        else
                c=a+b;
}
```

File.l

```
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
```

```
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
```

File.y

```
%{
#include<string.h>
#include<stdio.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack
{
int items[100];
int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
```

```
{
char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
```

```
| EXPR '*' EXPR {AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK { strcpy(QUAD[Index].op,"GOTO"); strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
```

```
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE'('CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
```

```
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ------------------------------""\n\t\t Pos Operator \tArg1 \tArg2 \tResult" "\n\t\t----
----------------");
for(i=0;i<Index;i++)
{
printf("\n\t\t%d\t%s\t%s\t%s\t%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].re
sult);
}
printf("\n\t\t ------------------------");
printf("\n\n"); return 0; }
void push(int data)
{ stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
```

```
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

Experiment No: 4,5 and 6

(Combined Program for 4, 5 and 6)

4. **Write program to find ε – closure of all states of any given NFA with ε transition.**
5. **Write program to convert NFA with ε transition to NFA without ε transition.**
6. **Write program to convert NFA to DFA.**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100
char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;
// Structure to store DFA states and their
// status ( i.e new entry or already present)
struct DFA {
  char *states;
  int count;
} dfa;
int last_index = 0;
FILE *fp;
int symbols;
/* reset the hash map*/
void reset(int ar[], int size) {
  int i;
  // reset all the values of
  // the mapping array to zero
  for (i = 0; i < size; i++) {
    ar[i] = 0;
  }
}
// Check which States are present in the e-closure
/* map the states of NFA to a hash set*/
void check(int ar[], char S[]) {
  int i, j;
```

```
  // To parse the individual states of NFA
  int len = strlen(S);
  for (i = 0; i < len; i++) {
    // Set hash map for the position
    // of the states which is found
    j = ((int)(S[i]) - 65);
    ar[j]++;
  }
}
// To find new Closure States
void state(int ar[], int size, char S[]) {
  int j, k = 0;
  // Combine multiple states of NFA
  // to create new states of DFA
  for (j = 0; j < size; j++) {
    if (ar[j] != 0)
      S[k++] = (char)(65 + j);
  }
  // mark the end of the state
  S[k] = '\0';
}
// To pick the next closure from closure set
int closure(int ar[], int size) {
  int i;
 // check new closure is present or not
  for (i = 0; i < size; i++) {
    if (ar[i] == 1)
      return i;
  }
  return (100);
}
// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
  int i;
  for (i = 0; i < last_index; i++) {
    if (dfa[i].count == 0)
```

```
    return 1;
  }
  return -1;
}
/* To Display epsilon closure*/
void Display_closure(int states, int closure_ar[],
                     char *closure_table[],
                     char *NFA_TABLE[][symbols + 1],
                     char *DFA_TABLE[][symbols]) {
  int i;
  for (i = 0; i < states; i++) {
    reset(closure_ar, states);
    closure_ar[i] = 2;
    // to neglect blank entry
    if (strcmp(&NFA_TABLE[i][symbols], "-") != 0) {
      // copy the NFA transition state to buffer
      strcpy(buffer, &NFA_TABLE[i][symbols]);
      check(closure_ar, buffer);
      int z = closure(closure_ar, states);
      // till closure get completely saturated
      while (z != 100)
      {
        if (strcmp(&NFA_TABLE[z][symbols], "-") != 0) {
          strcpy(buffer, &NFA_TABLE[z][symbols]);
          // call the check function
          check(closure_ar, buffer);
        }
        closure_ar[z]++;
        z = closure(closure_ar, states);
      }
    }
    // print the e closure for every states of NFA
    printf("\n e-Closure (%c) :\t", (char)(65 + i));
    bzero((void *)buffer, MAX_LEN);
    state(closure_ar, states, buffer);
    strcpy(&closure_table[i], buffer);
    printf("%s\n", &closure_table[i]);
```

```
  }
}
/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {
  int i;
  // To check the current state is already
  // being used as a DFA state or not in
  // DFA transition table
  for (i = 0; i < last_index; i++) {
    if (strcmp(&dfa[i].states, S) == 0)
      return 0;
  }
 // push the new
  strcpy(&dfa[last_index++].states, S);
  // set the count for new states entered
  // to zero
  dfa[last_index - 1].count = 0;
  return 1;
}
// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int st,
  char *NFT[][symbols + 1], char TB[]) {
  int len = strlen(S);
  int i, j, k, g;
  int arr[st];
  int sz;
  reset(arr, st);
  char temp[MAX_LEN], temp2[MAX_LEN];
  char *buff;
  // Transition function from NFA to DFA
  for (i = 0; i < len; i++) {
    j = ((int)(S[i] - 65));
    strcpy(temp, &NFT[j][M]);
    if (strcmp(temp, "-") != 0) {
      sz = strlen(temp);
      g = 0;
```

```
      while (g < sz) {
        k = ((int)(temp[g] - 65));
        strcpy(temp2, &clsr_t[k]);
        check(arr, temp2);
        g++;
      }
    }
  }
  bzero((void *)temp, MAX_LEN);
  state(arr, st, temp);
  if (temp[0] != '\0') {
    strcpy(TB, temp);
  } else
    strcpy(TB, "-");
}
/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA *dfa_states,
                 char *DFA_TABLE[][symbols]) {
  int i, j;
  printf("\n\n*************************************************************\n\n");
  printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");
  printf("\n STATES OF DFA :\t\t");
  for (i = 1; i < last_index; i++)
    printf("%s, ", &dfa_states[i].states);
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR DFA: \t");
  for (i = 0; i < symbols; i++)
    printf("%d, ", i);
  printf("\n\n");
  printf("STATES\t");
  for (i = 0; i < symbols; i++)
    printf("|%d\t", i);
  printf("\n");
  // display the DFA transition state table
  printf("--------+----------------------\n");
  for (i = 0; i < zz; i++) {
    printf("%s\t", &dfa_states[i + 1].states);
```

```
    for (j = 0; j < symbols; j++) {
      printf("|%s \t", &DFA_TABLE[i][j]);
    }
    printf("\n");
  }
}
// Driver Code
int main() {
  int i, j, states;
  char T_buf[MAX_LEN];
  // creating an array dfa structures
  struct DFA *dfa_states = malloc(MAX_LEN * (sizeof(dfa)));
  states = 6, symbols = 2;
  printf("\n STATES OF NFA :\t\t");
  for (i = 0; i < states; i++)
    printf("%c, ", (char)(65 + i));
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR NFA: \t");
  for (i = 0; i < symbols; i++)
            printf("%d, ", i);
  printf("eps");
  printf("\n\n");
  char *NFA_TABLE[states][symbols + 1];
  // Hard coded input for NFA table
  char *DFA_TABLE[MAX_LEN][symbols];
  strcpy(&NFA_TABLE[0][0], "FC");
  strcpy(&NFA_TABLE[0][1], "-");
  strcpy(&NFA_TABLE[0][2], "BF");
  strcpy(&NFA_TABLE[1][0], "-");
  strcpy(&NFA_TABLE[1][1], "C");
  strcpy(&NFA_TABLE[1][2], "-");
  strcpy(&NFA_TABLE[2][0], "-");
  strcpy(&NFA_TABLE[2][1], "-");
  strcpy(&NFA_TABLE[2][2], "D");
  strcpy(&NFA_TABLE[3][0], "E");
  strcpy(&NFA_TABLE[3][1], "A");
  strcpy(&NFA_TABLE[3][2], "-");
```

```
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE \n\n\n");
printf("STATES\t");
 for (i = 0; i < symbols; i++)
  printf("|%d\t", i);
printf("eps\n");
 // Displaying the matrix of NFA transition table
printf("--------+-------------------------------------\n");
for (i = 0; i < states; i++) {
  printf("%c\t", (char)(65 + i));
   for (j = 0; j <= symbols; j++) {
    printf("|%s \t", &NFA_TABLE[i][j]);
  }
  printf("\n");
}
int closure_ar[states];
char *closure_table[states];
 Display_closure(states, closure_ar, closure_table, NFA_TABLE, DFA_TABLE);
strcpy(&dfa_states[last_index++].states, "-");
 dfa_states[last_index - 1].count = 1;
bzero((void *)buffer, MAX_LEN);
  strcpy(buffer, &closure_table[0]);
strcpy(&dfa_states[last_index++].states, buffer);
  int Sm = 1, ind = 1;
int start_index = 1;
  // Filling up the DFA table with transition values
// Till new states can be entered in DFA table
while (ind != -1) {
  dfa_states[start_index].count = 1;
  Sm = 0;
  for (i = 0; i < symbols; i++) {
      trans(buffer, i, closure_table, states, NFA_TABLE, T_buf);
```

```
      // storing the new DFA state in buffer
     strcpy(&DFA_TABLE[zz][i], T_buf);
      // parameter to control new states
     Sm = Sm + new_states(dfa_states, T_buf);
    }
    ind = indexing(dfa_states);
    if (ind != -1)
     strcpy(buffer, &dfa_states[++start_index].states);
    zz++;
   }
  // display the DFA TABLE
  Display_DFA(last_index, dfa_states, DFA_TABLE);
   return 0;
}
```

Experiment No:7

Write program to minimize any given DFA.

```
#include <stdio.h>
    7.
#include <string.h>
    8.

    9.
#define STATES  99
    10.
#define SYMBOLS 20
    11.
    12.
int N_symbols;  /* number of input symbols */
    13.
int N_DFA_states;   /* number of DFA states */
    14.
char *DFA_finals;   /* final-state string */
    15.
int DFAtab[STATES][SYMBOLS];
    16.

    17.
char StateName[STATES][STATES+1];   /* state-name table */
    18.

    19.
int N_optDFA_states;    /* number of optimized DFA states */
    20.
int OptDFA[STATES][SYMBOLS];
    21.
char NEW_finals[STATES+1];
    22.

    23.
/*
    24.
   Print state-transition table.
    25.
   State names: 'A', 'B', 'C', ...
    26.
*/
    27.
```

```
void print_dfa_table(
    28.
  int tab[][SYMBOLS], /* DFA table */
    29.
  int nstates,    /* number of states */
    30.
  int nsymbols,   /* number of input symbols */
    31.
  char *finals)
    32.
{
    33.
  int i, j;
    34.

    35.
  puts("\nDFA: STATE TRANSITION TABLE");
    36.

    37.
  /* input symbols: '0', '1', ... */
    38.
  printf("     | ");
    39.
  for (i = 0; i < nsymbols; i++) printf("  %c  ", '0'+i);
    40.

    41.
  printf("\n-----+--");
    42.
  for (i = 0; i < nsymbols; i++) printf("-----");
    43.
  printf("\n");
    44.

    45.
  for (i = 0; i < nstates; i++) {
    46.
    printf("  %c  | ", 'A'+i);  /* state */
    47.
    for (j = 0; j < nsymbols; j++)
    48.
        printf("  %c  ", tab[i][j]);    /* next state */
    49.
    printf("\n");
```

```
    }
    printf("Final states = %s\n", finals);
}

/*
   Initialize NFA table.
*/
void load_DFA_table()
{

   DFAtab[0][0] = 'B'; DFAtab[0][1] = 'C';
   DFAtab[1][0] = 'E'; DFAtab[1][1] = 'F';
   DFAtab[2][0] = 'A'; DFAtab[2][1] = 'A';
   DFAtab[3][0] = 'F'; DFAtab[3][1] = 'E';
   DFAtab[4][0] = 'D'; DFAtab[4][1] = 'F';
   DFAtab[5][0] = 'D'; DFAtab[5][1] = 'E';

   DFA_finals = "EF";
   N_DFA_states = 6;
   N_symbols = 2;
}
```

```
/*
   Get next-state string for current-state string.
*/
void get_next_state(char *nextstates, char *cur_states,
   int dfa[STATES][SYMBOLS], int symbol)
{
   int i, ch;

   for (i = 0; i < strlen(cur_states); i++)
     *nextstates++ = dfa[cur_states[i]-'A'][symbol];
   *nextstates = '\0';
}

/*
   Get index of the equivalence states for state 'ch'.
   Equiv. class id's are '0', '1', '2', ...
*/
char equiv_class_ndx(char ch, char stnt[][STATES+1], int n)
{
   int i;

   for (i = 0; i < n; i++)
     if (strchr(stnt[i], ch)) return i+'0';
```

```
    return -1;  /* next state is NOT defined */
}

/*
   Check if all the next states belongs to same equivalence class.
   Return value:
     If next state is NOT unique, return 0.
     If next state is unique, return next state --> 'A/B/C/...'
   's' is a '0/1' string: state-id's
*/
char is_one_nextstate(char *s)
{
   char equiv_class;   /* first equiv. class */

   while (*s == '@') s++;
   equiv_class = *s++; /* index of equiv. class */

   while (*s) {
     if (*s != '@' && *s != equiv_class) return 0;
     s++;
   }
```

```
    return equiv_class; /* next state: char type */
}

int state_index(char *state, char stnt[][STATES+1], int n, int *pn,
    int cur)    /* 'cur' is added only for 'printf()' */
{
    int i;
    char state_flags[STATES+1]; /* next state info. */

    if (!*state) return -1; /* no next state */

    for (i = 0; i < strlen(state); i++)
        state_flags[i] = equiv_class_ndx(state[i], stnt, n);
    state_flags[i] = '\0';

    printf("   %d:[%s]\t--> [%s] (%s)\n",
        cur, stnt[cur], state, state_flags);

    if (i=is_one_nextstate(state_flags))
        return i-'0';   /* deterministic next states */
    else {
        strcpy(stnt[*pn], state_flags); /* state-division info */
        return (*pn)++;
```

```
}
}

/*
   Divide DFA states into finals and non-finals.
*/
int init_equiv_class(char statename[][STATES+1], int n, char *finals)
{
   int i, j;

   if (strlen(finals) == n) {  /* all states are final states */
      strcpy(statename[0], finals);
      return 1;
   }

   strcpy(statename[1], finals);   /* final state group */

   for (i=j=0; i < n; i++) {
      if (i == *finals-'A') {
         finals++;
      } else statename[0][j++] = i+'A';
   }
```

```
    statename[0][j] = '\0';
    return 2;
}
/*
    Get optimized DFA 'newdfa' for equiv. class 'stnt'.
*/
int get_optimized_DFA(char stnt[][STATES+1], int n,
    int dfa[][SYMBOLS], int n_sym, int newdfa[][SYMBOLS])
{
    int n2=n;       /* 'n' + <num. of state-division info> */
    int i, j;
    char nextstate[STATES+1];

    for (i = 0; i < n; i++) {    /* for each pseudo-DFA state */
      for (j = 0; j < n_sym; j++) {    /* for each input symbol */
          get_next_state(nextstate, stnt[i], dfa, j);
          newdfa[i][j] = state_index(nextstate, stnt, n, &n2, i)+'A';
      }
    }

    return n2;
```

```
185.
}
186.

187.
/*
188.
   char 'ch' is appended at the end of 's'.
189.
*/
190.
void chr_append(char *s, char ch)
191.
{
192.
   int n=strlen(s);
193.

194.
   *(s+n) = ch;
195.
   *(s+n+1) = '\0';
196.
}
197.

198.
void sort(char stnt[][STATES+1], int n)
199.
{
200.
   int i, j;
201.
   char temp[STATES+1];
202.

203.
   for (i = 0; i < n-1; i++)
204.
     for (j = i+1; j < n; j++)
205.
         if (stnt[i][0] > stnt[j][0]) {
206.
            strcpy(temp, stnt[i]);
207.
```

```
            strcpy(stnt[i], stnt[j]);
    208.
            strcpy(stnt[j], temp);
    209.
        }
    210.
}
    211.

    212.
/*
    213.
   Divide first equivalent class into subclasses.
    214.
     stnt[i1] : equiv. class to be segmented
    215.
     stnt[i2] : equiv. vector for next state of stnt[i1]
    216.
   Algorithm:
    217.
     - stnt[i1] is splitted into 2 or more classes 's1/s2/...'
    218.
     - old equiv. classes are NOT changed, except stnt[i1]
    219.
     - stnt[i1]=s1, stnt[n]=s2, stnt[n+1]=s3, ...
    220.
   Return value: number of NEW equiv. classses in 'stnt'.
    221.
*/
    222.
int split_equiv_class(char stnt[][STATES+1],
    223.
   int i1, /* index of 'i1'-th equiv. class */
    224.
   int i2, /* index of equiv. vector for 'i1'-th class */
    225.
   int n,  /* number of entries in 'stnt' */
    226.
   int n_dfa)  /* number of source DFA entries */
    227.
{
    228.
   char *old=stnt[i1], *vec=stnt[i2];
    229.
   int i, n2, flag=0;
```

```
  230.
 char newstates[STATES][STATES+1];   /* max. 'n' subclasses */
  231.

  232.
 for (i=0; i < STATES; i++) newstates[i][0] = '\0';
  233.

  234.
 for (i=0; vec[i]; i++)
  235.
   chr_append(newstates[vec[i]-'0'], old[i]);
  236.

  237.
 for (i=0, n2=n; i < n_dfa; i++) {
  238.
   if (newstates[i][0]) {
  239.
      if (!flag) {    /* stnt[i1] = s1 */
  240.
         strcpy(stnt[i1], newstates[i]);
  241.
         flag = 1;   /* overwrite parent class */
  242.
      } else  /* newstate is appended in 'stnt' */
  243.
         strcpy(stnt[n2++], newstates[i]);
  244.
   }
  245.
 }
  246.

  247.
 sort(stnt, n2); /* sort equiv. classes */
  248.

  249.
 return n2;  /* number of NEW states(equiv. classes) */
  250.
}
  251.

  252.
```

```
/*
    253.
   Equiv. classes are segmented and get NEW equiv. classes.
    254.
*/
    255.
int set_new_equiv_class(char stnt[][STATES+1], int n,
    256.
   int newdfa[][SYMBOLS], int n_sym, int n_dfa)
    257.
{
    258.
   int i, j, k;
    259.

    260.
   for (i = 0; i < n; i++) {
    261.
     for (j = 0; j < n_sym; j++) {
    262.
        k = newdfa[i][j]-'A';   /* index of equiv. vector */
    263.
        if (k >= n)  /* equiv. class 'i' should be segmented */
    264.
            return split_equiv_class(stnt, i, k, n, n_dfa);
    265.
     }
    266.
   }
    267.

    268.
   return n;
    269.
}
    270.

    271.
void print_equiv_classes(char stnt[][STATES+1], int n)
    272.
{
    273.
   int i;
    274.
```

```
    printf("\nEQUIV. CLASS CANDIDATE ==>");
    for (i = 0; i < n; i++)
      printf(" %d:[%s]", i, stnt[i]);
    printf("\n");
}

/*
    State-minimization of DFA: 'dfa' --> 'newdfa'
    Return value: number of DFA states.
*/
int optimize_DFA(
    int dfa[][SYMBOLS], /* DFA state-transition table */
    int n_dfa,  /* number of DFA states */
    int n_sym,  /* number of input symbols */
    char *finals,   /* final states of DFA */
    char stnt[][STATES+1],  /* state name table */
    int newdfa[][SYMBOLS])  /* reduced DFA table */
{
    char nextstate[STATES+1];
    int n;  /* number of new DFA states */
    int n2; /* 'n' + <num. of state-dividing info> */

```

```
    n = init_equiv_class(stnt, n_dfa, finals);
     298.

     299.
    while (1) {
     300.
      print_equiv_classes(stnt, n);
     301.
      n2 = get_optimized_DFA(stnt, n, dfa, n_sym, newdfa);
     302.
      if (n != n2)
     303.
          n = set_new_equiv_class(stnt, n, newdfa, n_sym, n_dfa);
     304.
      else break; /* equiv. class segmentation ended!!! */
     305.
    }
     306.

     307.
    return n;   /* number of DFA states */
     308.
}
     309.

     310.
/*
     311.
    Check if 't' is a subset of 's'.
     312.
*/
     313.
int is_subset(char *s, char *t)
     314.
{
     315.
    int i;
     316.

     317.
    for (i = 0; *t; i++)
     318.
      if (!strchr(s, *t++)) return 0;
     319.
    return 1;
```

```
320.
}
321.

322.
/*
323.
   New finals states of reduced DFA.
324.
*/
325.
void get_NEW_finals(
326.
   char *newfinals,    /* new DFA finals */
327.
   char *oldfinals,    /* source DFA finals */
328.
   char stnt[][STATES+1],  /* state name table */
329.
   int n)  /* number of states in 'stnt' */
330.
{
331.
   int i;
332.

333.
   for (i = 0; i < n; i++)
334.
     if (is_subset(oldfinals, stnt[i])) *newfinals++ = i+'A';
335.
   *newfinals++ = '\0';
336.
}
337.

338.
void main()
339.
{
340.
   load_DFA_table();
341.
   print_dfa_table(DFAtab, N_DFA_states, N_symbols, DFA_finals);
342.
```

```
    343.
   N_optDFA_states = optimize_DFA(DFAtab, N_DFA_states,
    344.
         N_symbols, DFA_finals, StateName, OptDFA);
    345.
   get_NEW_finals(NEW_finals, DFA_finals, StateName, N_optDFA_states);
    346.

    347.
   print_dfa_table(OptDFA, N_optDFA_states, N_symbols, NEW_finals);
    348.
}
```

Experiment No:8

Develop an operator precedence parser for a given language.

```
#include<stdio.h>
#include<conio.h>
void main(){
char stack[20],ip[20],opt[10][10][1],ter[10];
int i,j,k,n,top=0,col,row;
clrscr();
for(i=0;i<10;i++)
{
 stack[i]=NULL;
 ip[i]=NULL;
 for(j=0;j<10;j++)
 {
 opt[i][j][1]=NULL;
 }
}
printf("Enter the no.of terminals :\n");
```

```
scanf("%d",&n);
printf("\nEnter the terminals :\n");
scanf("%s",&ter);
printf("\nEnter the table values :\n");
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 {
 printf("Enter the value for %c %c:",ter[i],ter[j]);
 scanf("%s",opt[i][j]);
 }
}
printf("\n**** OPERATOR PRECEDENCE TABLE ****\n");
for(i=0;i<n;i++)
{
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++){printf("\n%c",ter[i]);
for(j=0;j<n;j++){printf("\t%c",opt[i][j][0]);}}
stack[top]='$';
printf("\nEnter the input string:");
scanf("%s",ip);
i=0;
printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
printf("\n%s\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
col=k;
if(ip[i]==ter[k])
row=k;
}
if((stack[top]=='$')&&(ip[i]=='$')){
printf("String is accepted\n");
```

```
break;}
else if((opt[col][row][0]=='<') ||(opt[col][row][0]=='='))
{ stack[++top]=opt[col][row][0];
stack[++top]=ip[i];
 printf("Shift %c",ip[i]);
 i++;
  }
else{
 if(opt[col][row][0]=='>')
{
while(stack[top]!='<'){--top;}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
printf("\n");
for(k=0;k<=top;k++)
{
printf("%c",stack[k]);
}
printf("\t\t\t");
for(k=i;k<strlen(ip);k++){
printf("%c",ip[k]);
}
printf("\t\t\t");
}
getch();}
```

Experiment No:9

Write program to Simulate First and Follow of any given grammar.

```
// C program to calculate the First and
// Follow sets of a given grammar
#include<stdio.h>
#include<ctype.h>
#include<string.h>
// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);
// Function to calculate First
void findfirst(char, int, int);
int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];
```

```
// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    // The Input grammar
    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");

    int kay;
    char done[count];
    int ptr = -1;

    // Initializing the calc_first array
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
```

```
        calc_first[k][kay] = '!';
    }
}
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
{
    c = production[k][0];
    point2 = 0;
    xxx = 0;

    // Checking if First of c has
    // already been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(c == done[kay])
            xxx = 1;

    if (xxx == 1)
        continue;

    // Function call
    findfirst(c, 0, 0);
    ptr += 1;

    // Adding c to the calculated list
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

    // Printing the First Sets of the grammar
    for(i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;

        for(lark = 0; lark < point2; lark++) {

            if (first[i] == calc_first[point1][lark])
            {
```

```
                chk = 1;
                break;
            }
        }
        if(chk == 0)
        {
            printf("%c, ", first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
    printf("}\n");
    jm = n;
    point1++;
}
printf("\n");
printf("-----------------------------------------------\n\n");
char donee[count];
ptr = -1;

// Initializing the calc_follow array
for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
    ck = production[e][0];
    point2 = 0;
    xxx = 0;

    // Checking if Follow of ck
    // has alredy been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
```

```
            xxx = 1;

        if (xxx == 1)
            continue;
        land += 1;

        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        // Printing the Follow Sets of the grammar
        for(i = 0 + km; i < m; i++) {
            int lark = 0, chk = 0;
            for(lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}
```

```
void follow(char c)
{
   int i, j;

   // Adding "$" to the follow
   // set of the start symbol
   if(production[0][0] == c) {
      f[m++] = '$';
   }
   for(i = 0; i < 10; i++)
   {
      for(j = 2;j < 10; j++)
      {
         if(production[i][j] == c)
         {
            if(production[i][j+1] != '\0')
            {
               // Calculate the first of the next
               // Non-Terminal in the production
               followfirst(production[i][j+1], i, (j+2));
            }

            if(production[i][j+1]=='\0' && c!=production[i][0])
            {
               // Calculate the follow of the Non-Terminal
               // in the L.H.S. of the production
               follow(production[i][0]);
            }
         }
      }
   }
}

void findfirst(char c, int q1, int q2)
{
   int j;
```

```
// The case where we
// encounter a Terminal
if(!(isupper(c))) {
   first[n++] = c;
}
for(j = 0; j < count; j++)
{
   if(production[j][0] == c)
   {
      if(production[j][2] == '#')
      {
         if(production[q1][q2] == '\0')
            first[n++] = '#';
         else if(production[q1][q2] != '\0'
                  && (q1 != 0 || q2 != 0))
         {
            // Recursion to calculate First of New
            // Non-Terminal we encounter after epsilon
            findfirst(production[q1][q2], q1, (q2+1));
         }
         else
            first[n++] = '#';
      }
      else if(!isupper(production[j][2]))
      {
         first[n++] = production[j][2];
      }
      else
      {
         // Recursion to calculate First of
         // New Non-Terminal we encounter
         // at the beginning
         findfirst(production[j][2], j, 3);
      }
   }
}
```

```
}

void followfirst(char c, int c1, int c2)
{
    int k;

    // The case where we encounter
    // a Terminal
    if(!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for(i = 0; i < count; i++)
        {
            if(calc_first[i][0] == c)
                break;
        }

        //Including the First set of the
        // Non-Terminal in the Follow of
        // the original query
        while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if(production[c1][c2] == '\0')
                {
                    // Case where we reach the
                    // end of a production
                    follow(production[c1][0]);
                }
                else
```

```
            {
                // Recursion to the next symbol
                // in case we encounter a "#"
                followfirst(production[c1][c2], c1, c2+1);
            }
        }
        j++;
    }
  }
}
```

Experiment No:10

Construct a recursive descent parser for an expression.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void Tprime();
void Eprime();
void E();
void check();
void T();
char expression[10];
int count, flag;
int main()
{
    count = 0;
    flag = 0;
    printf("\nEnter an Algebraic Expression:\t");
```

```
    scanf("%s", expression);
    E();
    if((strlen(expression) == count) && (flag == 0))
    {
        printf("\nThe Expression %s is Valid\n", expression);
    }
    else
    {
        printf("\nThe Expression %s is Invalid\n", expression);
    }
}
void E()
{
    T();
    Eprime();
}
void T()
{
    check();
    Tprime();
}
void Tprime()
{
    if(expression[count] == '*')
    {
        count++;
        check();
        Tprime();
    }
}
void check()
{
    if(isalnum(expression[count]))
    {
        count++;
    }
    else if(expression[count] == '(')
```

```
    {
        count++;
        E();
        if(expression[count] == ')')
        {
            count++;
        }
        else
        {
            flag = 1;
        }
    }
    else
    {
        flag = 1;
    }
}
void Eprime()
{
    if(expression[count] == '+')
    {
        count++;
        T();
        Eprime();
    }
}
```

Experiment No:11

Construct a Shift Reduce Parser for a given language.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
void main()
  {
    clrscr();
puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
     {
```

```
        if(a[j]=='i' && a[j+1]=='d')
          {
            stk[i]=a[j];
            stk[i+1]=a[j+1];
            stk[i+2]='\0';
            a[j]=' ';
            a[j+1]=' ';
            printf("\n$%s\t%s$\t%sid",stk,a,act);
            check();
          }
        else
          {
            stk[i]=a[j];
            stk[i+1]='\0';
            a[j]=' ';
            printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
            check();
          }
      }
      getch();
  }
void check()
  {
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
      if(stk[z]=='i' && stk[z+1]=='d')
        {
          stk[z]='E';
          stk[z+1]='\0';
          printf("\n$%s\t%s$\t%s",stk,a,ac);
          j++;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
        {
          stk[z]='E';
          stk[z+1]='\0';
```

```
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        i=i-2;
       }
   for(z=0; z<c; z++)
    if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
       {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        i=i-2;
       }
   for(z=0; z<c; z++)
    if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
       {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        i=i-2;
       }
}
```

Experiment No:12

Write a program to perform loop unrolling.

```
#include<stdio.h>
#include<conio.h>
void main()
{
unsigned int n;
int x;
char ch;
clrscr();
printf("\nEnter N\n");
scanf("%u",&n);
printf("\n1. Loop Roll\n2. Loop UnRoll\n");
printf("\nEnter ur choice\n");
scanf(" %c",&ch);
switch(ch)
{
case '1':
  x=countbit1(n);
  printf("\nLoop Roll: Count of  1's    :  %d" ,x);
```

```
  break;
case '2':
  x=countbit2(n);
  printf("\nLoop UnRoll:  Count of 1's  :  %d" ,x);
  break;
default:
  printf("\n Wrong Choice\n");

}
getch();
}
int countbit1(unsigned int n)
{
    int bits = 0,i=0;
    while (n != 0)
    {
 if (n & 1) bits++;
 n >>= 1;
 i++;
    }
    printf("\n no of iterations  %d",i);
    return bits;
}
int countbit2(unsigned int n)
{
    int bits = 0,i=0;
    while (n != 0)
    {
 if (n & 1) bits++;
 if (n & 2) bits++;
 if (n & 4) bits++;
 if (n & 8) bits++;
 n >>= 4;
 i++;
    }
    printf("\n no of iterations  %d",i);
    return bits; }
```

Experiment No:13

Write a program to perform constant propagation.

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<conio.h>
void input();
void output();
void change(int p,char *res);
void constant();
struct expr
{
char op[2],op1[5],op2[5],res[5];
int flag;
}arr[10];
int n;
void main()
{
clrscr();
input();
```

```
constant();
output();
getch();
}
void input()
{
int i;
printf("\n\nEnter the maximum number of  expressions : ");
scanf("%d",&n);
printf("\nEnter the input : \n");
for(i=0;i<n;i++)
{
scanf("%s",arr[i].op);
scanf("%s",arr[i].op1);
scanf("%s",arr[i].op2);
scanf("%s",arr[i].res);
arr[i].flag=0;
}
}
void constant()
{
int i;
int op1,op2,res;
char op,res1[5];
for(i=0;i<n;i++)
{
if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=")==0) /*if both digits,
store them in variables*/
{
op1=atoi(arr[i].op1);
op2=atoi(arr[i].op2);
op=arr[i].op[0];
switch(op)
{
case '+':
res=op1+op2;
break;
```

```
case '-':
res=op1-op2;
break;
case '*':
res=op1*op2;
break;
case '/':
res=op1/op2;
break;
case '=':
res=op1;
break;
}
sprintf(res1,"%d",res);
arr[i].flag=1; /*eliminate expr and replace any operand below that uses result of this expr */
change(i,res1);
}
}
}
void output()
{
int i=0;
printf("\nOptimized code is : ");
for(i=0;i<n;i++)
{
if(!arr[i].flag)
{
printf("\n%s %s %s %s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
}
}
}
void change(int p,char *res)
{
int i;
for(i=p+1;i<n;i++)
{
if(strcmp(arr[p].res,arr[i].op1)==0)
```

```
strcpy(arr[i].op1,res);
else if(strcmp(arr[p].res,arr[i].op2)==0)
strcpy(arr[i].op2,res);
}
}
```

Sample Input and Output:

```
/*  INPUT:
        Enter the maximum number of  expressions : 4
        Enter the input :
        = 3 - a
        + a b t1
        + a c t2
        + t1 t2 t3
        OUTPUT:
        Optimized code is :
        + 3 b t1
        + 3 c t2
        + t1 t2 t3
*/
```

Experiment No:14

Implement Intermediate code generation for simple expressions.

File.y

```
%{
#include<stdio.h>
//#include<ctype.h>
#include<string.h>
%}
%token id digit  //named token
%left '+"-'
%left '*"/'
%left '^'
%right'='
%nonassoc UMINUS
%%
S:id {push();}'='E {pop();}
E:E'+' {push();}T {pop3();}
  |E'-' {push();}T {pop3();}
  |T
  ;
```

```
T:T'*' {push();}F {pop3();}
  |T'/' {push();}F {pop3();}
  |T'^'{push();}F {pop3();}
  |F
  ;
F:id {push();}
 |digit {push();}
 |'('E')'
 |'-' {push();}F {pop2();} %prec UMINUS
 ;
%%
#include"lex.yy.c"
char stack[10][10],var[10]="\0",temp[10]="\0";
int top=0;
char i='0';
int main()
{
yyparse();
yylex();
return 0;
}
yyerror(char *s)
{
 printf("%s\n",s);
}
push()
{
top++;
strcpy(stack[top],yytext);
}
pop3()
{
printf("\nt%c=%s%s%s",i,stack[top-2],stack[top-1],stack[top]);
top=top-2;
temp[0]=i;
strcpy(var,"t");
strcat(var,temp);
```

```
strcpy(stack[top],var);
i++;
}
pop()
{
printf("\n%s=%s\n",stack[top-1],stack[top]);
}
pop2()
{
printf("\nt%c=%s%s",i,stack[top-1],stack[top]);
top--;
temp[0]=i;
strcpy(var,"t");
strcat(var,temp);
strcpy(stack[top],var);
i++; }
```

File.l

```
%{
#include "y.tab.h"
%}
%%
[0-9]+  {return digit;}
[a-z]+  {return id;}
[ \t]  ;
[\n] {return 0;}
. {return yytext[0];}
%%
```

Experiment No:15

Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly-instructions can be simple move, add, sub, jump etc.

```
#include<stdio.h>
#include<string.h>
void main(){
char icode[10][30], str[20], opr[10];
int i=0;
printf("\nEnter the set of intermediate code (terminated by exit):\n");
do{
    scanf("%s", icode[i]);
}while(strcmp(icode[i++],"exit")!=0);
printf("\nTarget code generation");
printf("\n**********");
i=0;
do{
    strcpy(str,icode[i]);
    switch(str[3]){
        case '+':
```

```
                strcpy(opr,"ADD");
                break;
        case '-':
                strcpy(opr,"SUB");
                break;
        case '*':
                strcpy(opr,"MUL");
                break;
        case '/':
                strcpy(opr,"DIV");
                break;
    }
    printf("\n\tMov %c,R%d", str[2],i);
    printf("\n\%s%c,,R%d", opr,str[4],i);
    printf("\n\tMov R%d%c", i,str[0]);
}while(strcmp(icode[++i],"exit")!=0);    }
```

Model question paper

1. Program to recognize a valid arithmetic expression that uses operator +, –, * and /.
2. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
3. Implementation of Calculator using LEX and YACC.
4. Write program to convert NFA with ε transition to NFA without ε transition.
5. Write program to convert NFA to DFA.
6. Write program to minimize any given DFA.
7. Implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.
8. Write a program to perform loop unrolling.
9. Write program to find Simulate First and Follow of any given grammar.
10. Implementation of Lexical Analyzer using Lex Tool

Model Viva Questions

a. What is a Compiler?

A compiler is a program that reads a program written in one language –the source language and translates it into an equivalent program in another language-the target language. The compiler reports to its user the presence of errors in the source program.

b. What Are The Two Parts Of A Compilation? Explain Briefly.

Analysis and Synthesis are the two parts of compilation.

The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

c. Define Compiler-compiler.

Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems.

Largely they are oriented around a particular model of languages , and they are suitable for generating compilers of languages similar model.

d. List The Various Compiler Construction Tools.

The following is a list of some compiler construction tools:

Parser generators
Scanner generators
Syntax-directed translation engines
Automatic code generators
Data-flow engines

e. Differentiate Tokens, Patterns, Lexeme.

Tokens- Sequence of characters that have a collective meaning.

Patterns- There is 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 sequence of characters in the source program that is matched by the pattern for a token.

f. Write A Regular Expression For An Identifier.

An identifier is defined as a letter followed by zero or more letters or digits.
The regular expression for an identifier is given as
letter (letter | digit)*

g. Explain lex and yacc tools:-

Lex:- scanner that can identify those tokens

Yacc:- parser.yacc takes a concise description of a grammar and produces a C routine that can parse that grammar.

h. Give the structure of the lex program:-

definition section- any intitial 'c' program code
% %
Rules section- pattern and action separated by white space
%%
User subroutines section-concsit of any legal code.

9. What are the phases of a compiler?
i) Lexical analysis.
ii) Syntax analysis.
iii) Intermediate code generation.
iv) Code optimization.
v) Code generation.

10. Define Passes?
In an implementation of a compiler, portion of one or more phases are combined into a module called pass. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which is read by subsequent pass.

11. Define Lexical Analysis?
The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Identifiers, keywords, constants, operators and punctuation symbols are typical tokens.

12. Write notes on syntax analysis?
Syntax analysis is also called parsing. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.

13. What is meant by semantic analysis?
The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operand of expressions and statements.

14. Define optimization?

Certain compilers apply transformations to the output of the intermediate code generator. It is used to produce an intermediate-language from which a faster or smaller object program can be produced. This phase is called optimization phase. Types of optimization are local optimization and loop optimization.

15. What is cross compiler?

A compiler may run on one machine and produce object code for another machine is called cross compiler.

16. Define finite automata? A better way to convert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a finite automaton.

17. Define LEX?

LEX is a tool for automatically generating lexical analyzers. A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The output of LEX is a lexical analyzer program.

18. Define parse trees?

The graphical representation for derivations that filters out the choice regarding replacement order. This representation is called the parse trees. It represents the hierarchical syntactic structure of sentences that is implied by the grammar.

19. Define parser?

A parser for grammar G is a program that takes as input a string w and produces as output either a parse tree for w, if w is a sentence of G, or an error message indicating that w is not a sentence of G.

20. Define intermediate code?

In many compilers the source code is translated into a language which is intermediate in complexity between a high-level programming language and machine code. Such a language is therefore called intermediate code or intermediate text.