E.G.S PILLAY ENGINEERING COLLEGE (AUTONOMOUS)

NAGAPATTINAM

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

LABORATORY MANUAL

B.Tech. Semester- VI

Subject Code : 1902CS651

Subject Name : COMPILER DESIGN LABORATORY

Class Coordinator : Dr. A. Emmanuel Peo Mariadas

ASSOCIATE PROFESSOR / CSE

Table of Contents

Vision and Mission of the Institute

Vision:

To impart Quality Education, to give an enviable growth to seekers of learning, to groom them as World Class Engineers and managers competent to match the expending expectations of the Corporate World has been ever enlarging vision extending to new horizons of Dronacharya College of Engineering

Mission:

1. To prepare students for full and ethical participation in a diverse society and encourage lifelong learning by following the principle of 'Shiksha evam Sahayata' i.e. Education & Help.
2. To impart high-quality education, knowledge and technology through rigorous academic programs, cutting-edge research, & Industry collaborations, with a focus on producing engineers& managers who are socially responsible, globally aware, & equipped to address complex challenges.
3. Educate students in the best practices of the field as well as integrate the latest research into the academics.
4. Provide quality learning experiences through effective classroom practices, innovative teaching practices and opportunities for meaningful interactions between students and faculty.
5. To devise and implement programmes of education in technology that are relevant to the changing needs of society, in terms of breadth of diversity and depth of specialization.

Vision and Mission of the Department

Vision:

"To become a Centre of Excellence in teaching and research in Information Technology for producing skilled professionals having a zeal to serve society"

Mission:

M1: To create an environment where students can be equipped with strong fundamental concepts, programming and problem-solving skills.

M2: To provide an exposure to emerging technologies by providing hands on experience for generating competent professionals.

M3: To promote Research and Development in the frontier areas of Information Technology and encourage students for pursuing higher education

M4: To inculcate in students ethics, professional values, team work and leadership skills.

Programme Educational Objectives (PEOs)

PEO1: To provide students with a sound knowledge of mathematical, scientific and engineering fundamentals required to solve real world problems.

PEO2: To develop research oriented analytical ability among students and to prepare them for making technical contribution to the society.

PEO3: To develop in students the ability to apply state-of-the–art tools and techniques for designing software products to meet the needs of Industry with due consideration for environment friendly and sustainable development.

PEO4: To prepare students with effective communication skills, professional ethics and managerial skills.

PEO5: To prepare students with the ability to upgrade their skills and knowledge for life-long learning.

Programme 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.

Program Specific Outcomes (PSOs)

PSO1: Analyze, identify and clearly define a problem for solving user needs by selecting, creating and evaluating a computer-based system through an effective project plan.

PSO2: Design, implement and evaluate processes, components and/or programs using modern techniques, skills and tools of core Information Technologies to effectively integrate secure IT-based solutions into the user environment.

PSO3: Develop impactful IT solutions by using research-based knowledge and research methods in the fields of integration, interface issues, security & assurance and implementation.

University Syllabus

1902CS651		COMPILER LABORATORY	L	T	P	C
			0	0	2	1
PREREQUISITES:		C programming language.				
COURSE OBJECTIVES:						
	1. Be exposed to compiler writing tools. Learn to implement the different Phases of compiler					
	2.Be familiar with control flow and data flow analysis					
	simple optimization techniques					

LIST OF EXPERIMENTS	
1. Implementation of Symbol Table	
2. Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)	
3. Implementation of Lexical Analyzer using Lex Tool	
4. 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. d)Implementation of Calculator using LEX and YACC	
5. Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.	
6. Implement type checking	
7. Implement control flow analysis and Data flow Analysis	
8. Implement any one storage allocation strategies(Heap, Stack, Static)	
9. Construction of DAG	
10. 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 a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.	
Total:	**45 Hours**

LIST OF EQUIPMENT FOR A BATCH OF 60 STUDENTS: Standalone desktops with C / C++ compiler and Compiler writing tools 30 Nos. (or) Server with C / C++ compiler and Compiler writing tools supporting 60 terminals or more. LEX and YACC

Additional Experiments:	
	1. Implementation of Simple Code Optimization Techniques (Constant Folding., etc.)
COURSE OUTCOMES:	
	After completion of the course, Student will be able to
	1. Implement the different Phases of compiler using tools 2. Analyze the control flow and data flow of a typical program 3. Optimize a given program 4. Generate an assembly language program equivalent to a source language program
REFERENCES:	
1. Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, “Learning from Data”, AMLBook Publishers, 2012.	
2. P. Flach, “Machine Learning: The art and science of algorithms that make sense of data”, Cambridge University Press, 2012.	
3. K. P. Murphy, “Machine Learning: A probabilistic perspective”, MIT Press, 2012.	
4. C. M. Bishop, “Pattern Recognition and Machine Learning”, Springer, 2007.	
5. D. Barber, “Bayesian Reasoning and Machine Learning”, Cambridge University Press, 2012.	
6. M. Mohri, A. Rostamizadeh, and A. Talwalkar, “Foundations of Machine Learning”, MIT Press, 2012.	
7. T. M. Mitchell, “Machine Learning”, McGraw Hill, 1997.	
8. S. Russel and P. Norvig, “Artificial Intelligence: A Modern Approach”, Third Edition, Prentice Hall, 2009.	

Course Outcomes (COs)

Upon successful completion of the course, the students will be able to:

CO1:	Implement the different Phases of compiler using tools
CO2:	Analyze the control flow and data flow of a typical program
CO3:	Optimize a given program
CO4:	Generate an assembly language program equivalent to a source Language program

CO-PO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	**2**	**3**	**1**	**1**	**3**	**1**	**3**		**2**	**2**	**1**	**3**
CO2	**3**	**2**	**1**	**3**	**3**	**1**	**1**		**3**	**3**	**2**	**3**
CO3	**2**	**2**		**3**	**3**		**2**	**1**	**3**	**2**		**2**
CO4	**2**	**2**	**3**		**3**	**1**	**1**		**2**	**1**	**1**	**2**
	2.25	**2.25**	**1.7**	**2.33**	**3**	**1**	**2.33**	**1**	**2.5**	**2**	**1.33**	**2.5**

CO-PSO Mapping

	PSO1	PSO2
CO1	**2**	**3**
CO2	**2**	**3**
CO3	**2**	**3**
CO4	**2**	**3**
CO5	**2**	**3**

Course Overview

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

This course is intended to teach his course covers all the phases of a compiler such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, target code generation, symbol table and error handler in details.

Compilers have become part and parcel of today's computer systems. They are responsible for making the user's computing requirements, specified as a piece of program, understandable to the underlying machine. There tools work as interface between the entities of two different domains – the human being and the machine. The actual process involved in this transformation is quite complex. Automata Theory provides the base of the course on which several automated tools can be designed to be used at various phases of a compiler. Advances in computer architecture, memory management and operating systems provide the compiler designer large number of options to try out for efficient code generation. This course on compiler design is to address all these issues, starting from the theoretical foundations to the architectural issues to automated tools.

List of Experiments mapped with Cos

S No.	Name of the Experiment	Course Outcome
1	Write a Program to implement the Symbol Table	CO1
2	Write a Program for developing a lexical analyzer to recognize a few patterns in C	CO1
3	Write a Program for implementation of Lexical analysis using LEX tools.	CO1
4	Write a Program using YACC 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	CO1
5	Write a Program to Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.	CO1
6	Write a Program to Implement type checking.	CO1
7	Write a Program to implement control flow analysis and Data flow Analysis.	CO2
8	Write a Program for implementation of Implement any one storage allocation strategies (Heap, Stack, Static).	CO2
9	Write the program for Construction of DAG.	CO2,CO3
10	Write a Program for implementation of 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 a 8086 assembler.	CO4
11	Innovative Experiments • Write a Program for simulating the task of a Recursive Descent Parser • Write a Program for implementation of SLR Parser.	CO2,CO3
12	Virtual Lab • Write a program to form a correct sentences using the given words • Write a program to form tokens and types	CO2,CO3

DOs and DON'Ts

DOs

1. Login-on with your username and password.
2. Log off the computer every time when you leave the Lab.
3. Arrange your chair properly when you are leaving the lab.
4. Put your bags in the designated area.
5. Ask permission to print.

DON'Ts

1. Do not share your username and password.
2. Do not remove or disconnect cables or hardware parts.
3. Do not personalize the computer setting.
4. Do not run programs that continue to execute after you log off.
5. Do not download or install any programs, games or music on computer in Lab.
6. Personal Internet use chat room for Instant Messaging (IM) and Sites is strictly prohibited.
7. No Internet gaming activities allowed.
8. Tea, Coffee, Water & Eatables are not allowed in the Computer Lab.

General Safety Precautions

Precautions (In case of Injury or Electric Shock)

1. To break the victim with live electric source, use an insulator such as fire wood or plastic to break the contact. Do not touch the victim with bare hands to avoid the risk of electrifying yourself.
2. Unplug the risk of faulty equipment. If main circuit breaker is accessible, turn the circuit off.
3. If the victim is unconscious, start resuscitation immediately, use your hands to press the chest in and out to continue breathing function. Use mouth-to-mouth resuscitation if necessary.
4. Immediately call medical emergency and security. Remember! Time is critical; be best.

Precautions (In case of Fire)

1. Turn the equipment off. If power switch is not immediately accessible, take plug off.
2. If fire continues, try to curb the fire, if possible, by using the fire extinguisher or by covering it with a heavy cloth if possible isolate the burning equipment from the other surrounding equipment.
3. Sound the fire alarm by activating the nearest alarm switch located in the hallway.
4. Call security and emergency department immediately:

Emergency	**:**	**Reception**
(Reception) Security	**:**	**Front Gate**

Guidelines to students for report preparation

All students are required to maintain a record of the experiments conducted by them. Guidelines for its preparation are as follows: -

1) All files must contain a title page followed by an index page. ***The files will not be signed by the faculty without an entry in the index page.***

2) Student’s Name, Roll number and date of conduction of experiment must be written on all pages.

3) For each experiment, the record must contain the following

(i) Aim/Objective of the experiment

(ii) Pre-experiment work (as given by the faculty)

(iii) Lab assignment questions and their solutions

(iv) Test Cases (if applicable to the course)

(v) Results/ output

Note:

1. Students must bring their lab record along with them whenever they come for the lab.

2. Students must ensure that their lab record is regularly evaluated.

Lab Assessment Criteria

An estimated 12 lab classes are conducted in a semester for each lab course. These lab classes are assessed continuously. Each lab experiment is evaluated based on 5 assessment criteria as shown in following table. Assessed performance in each experiment is used to compute CO attainment as well as internal marks in the lab course.

Grading Criteria	**Exemplary (4)**	**Competent (3)**	**Needs Improvement (2)**	**Poor (1)**
AC1: Pre-Lab written work (this may be assessed through viva)	Complete procedure with underlined concept is properly written	Underlined concept is written but procedure is incomplete	Not able to write concept and procedure	Underlined concept is not clearly Understood
AC2: Program Writing/ Modeling	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied, Program/solution written is readable	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied	Assigned problem is properly analyzed & correct solution designed	Assigned problem is properly analyzed
AC3: Identification & Removal of errors/ bugs	Able to identify errors/ bugs and remove them	Able to identify errors/ bugs and remove them with little bit of guidance	Is dependent totally on someone for identification of errors/ bugs and their removal	Unable to understand the reason for errors/ bugs even after they are explicitly pointed out
AC4: Execution & Demonstration	All variants of input /output are tested, Solution is well demonstrated and implemented concept is clearly explained	All variants of input /output are not tested, However, solution is well demonstrated and implemented concept is clearly explained	Only few variants of input /output are tested, Solution is well demonstrated but implemented concept is not clearly explained	Solution is not well demonstrated and implemented concept is not clearly explained
AC5: Lab Record Assessment	All assigned problems are well recorded with objective, design constructs and solution along with Performance analysis using all variants of input and output	More than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output	Less than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output	Less than 40 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output

LAB EXPERIMENTS

LAB EXPERIMENT 1

OBJECTIVE:

Write a Program to implement the Symbol Table

BRIEF DESCRIPTION:

Symbol table is an important data structure used in a compiler. Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

ALGORITHM:

```
Step1. Start the program for performing insert, display, delete, search and modify
        option in  symbol  table
Step 2. Define the structure of the Symbol Table
Step 3. Enter the choice for performing the operations in the symbol Table
Step 4. If the entered choice is 1, search the symbol table for the symbol to be inserted. If
         the    symbol   is already    present, it displays “Duplicate Symbol”. Else, insert
         the symbol  and the   corresponding address in the symbol table.
Step 5. If the entered choice is 2, the symbols present in the symbol table are displayed.
Step 6. If the entered choice is 3, the symbol to be deleted is searched in the symbol
         table.
Step 7. If it is not found in the symbol table it displays “Label Not found”. Else, the
         symbol is  deleted.
Step 8. If the entered choice is 5, the symbol to be modified is searched in the symbol
         table.  The label   or address or both can be modified.
Step 9.Stop the execution.
```

PRE-EXPERIMENT QUESTIONS:

1. What is pointer?
2. What is linked list?
3. What is data structure?

Program Code:

```
# include <stdio.h>
# include <conio.h>
# include <alloc.h>
# include <string.h>
# define null 0
int size=0;
void insert();
void del();
int search(char lab[]);
void modify();
void display();
struct symbtab
{
char label[10];
int addr;
struct symtab *next;
};
struct symbtab *first,*last;
void main()
{
int op;
int y;
char la[10];
clrscr();
do
{
printf("\nSYMBOL TABLE IMPLEMENTATION\n");
printf("1. INSERT\n");
printf("2. DISPLAY\n");
printf("3. DELETE\n");
printf("4. SEARCH\n");
printf("5. MODIFY\n");
printf("6. END\n");
printf("Enter your option : ");
scanf("%d",&op);
switch(op)
{
case 1:
insert();
display();
break;
case 2:
display();
break;
case 3:
del();
display();
break;
case 4:
printf("Enter the label to be searched : ");
scanf("%s",la);
y=search(la);
if(y==1)
{
```

```
printf("The label is already in the symbol Table");
}
else
{
printf("The label is not found in the symbol table");
}
break;
case 5:
modify();
display();
break;
case 6:
break;
}
}
while(op<6);
getch();
}
void insert()
{
int n;
char l[10];
printf("Enter the label : ");
scanf("%s",l);
n=search(l);
if(n==1)
{
printf("The label already exists. Duplicate cant be inserted\n");
}
else
{
struct symbtab *p;
p=malloc(sizeof(struct symbtab));
strcpy(p->label,l);
printf("Enter the address : ");
scanf("%d",&p->addr);
p->next=null;
if(size==0)
{
first=p;
last=p;
}
else
{
last->next=p;
last=p;
}
size++;
}
}
void display()
{
int i;
struct symbtab *p;
p=first;
printf("LABEL\tADDRESS\n");
for(i=0;i<size;i++)
{
```

```
printf("%s\t%d\n",p->label,p->addr);
p=p->next;
}
}
int search(char lab[])
{
int i,flag=0;
struct symbtab *p;
p=first;
for(i=0;i<size;i++)
{
if(strcmp(p->label,lab)==0)
{
flag=1;
}
p=p->next;
}
return flag;
}
void modify()
{
char l[10],nl[10];
int add, choice, i, s;
struct symbtab *p;
p=first;
printf("What do you want to modify?\n");
printf("1. Only the label\n");
printf("2. Only the address of a particular label\n");
printf("3. Both the label and address\n");
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Enter the old label\n");
scanf("%s",l);
printf("Enter the new label\n");
scanf("%s",nl);
s=search(l);
if(s==0)
{
printf("NO such label");
}
else
{
for(i=0;i<size;i++)
{
if(strcmp(p->label,l)==0)
{
strcpy(p->label,nl);
}
p=p->next;
}
}
break;
case 2:
printf("Enter the label whose address is to modified\n");
scanf("%s",l);
```

```
printf("Enter the new address\n");
scanf("%d",&add);
s=search(l);
if(s==0)
{
printf("NO such label");
}
else
{
for(i=0;i<size;i++)
{
if(strcmp(p->label,l)==0)
{
p->addr=add;
}
p=p->next;
}
}
break;
case 3:
printf("Enter the old label : ");
scanf("%s",l);
printf("Enter the new label : ");
scanf("%s",nl);
printf("Enter the new address : ");
scanf("%d",&add);
s=search(l);
if(s==0)
{
printf("NO such label");
}
else
{
for(i=0;i<size;i++)
{
if(strcmp(p->label,l)==0)
{
strcpy(p->label,nl);
p->addr=add;
}
p=p->next;
}
}
break;
}
}
void del()
{
int a;
char l[10];
struct symbtab *p,*q;
p=first;
printf("Enter the label to be deleted\n");
scanf("%s",l);
a=search(l);
if(a==0)
{
printf("Label not found\n");
```

```
}
else
{
if(strcmp(first->label,l)==0)
{
first=first->next;
}
else if(strcmp(last->label,l)==0)
{
q=p->next;
while(strcmp(q->label,l)!=0)
{
p=p->next;
q=q->next;
}
p->next=null;
last=p;
}
else
{
q=p->next;
while(strcmp(q->label,l)!=0)
{
p=p->next;
q=q->next;
}
p->next=q->next;
}
size--;
}
}
```

Output:

```
SYMBOL TABLE IMPLEMENTATION

1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 1
Enter the label : aa
Enter the address : 1000
LABEL   ADDRESS
aa      1000

SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 1
Enter the label : bb
Enter the address : 2000
```

```
LABEL   ADDRESS
aa      1000
bb      2000

SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 4
Enter the label to be searched : aa
The label is already in the symbol Table
SYMBOL TABLE IMPLEMENTATION
1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END
Enter your option : 5
What do you want to modify?
1. Only the label
2. Only the address of a particular label
3. Both the label and address
Enter your choice : 1
Enter the old label
aa
Enter the new label
aa1
LABEL   ADDRESS
aa1     1000
bb      2000
```

POST-EXPERIMENT QUESTIONS:

1. Where is symbol table stored?

2. What is symbol and its types?

3. What is local symbol table?

LAB EXPERIMENT 2

OBJECTIVE:

Write a Program to implement the Symbol Table

BRIEF DESCRIPTION:

Token is a group of characters having collective meaning: typically, a word or punctuation mark, separated by a lexical analyzer and passed to a parser. A lexeme is an actual character sequence forming a specific instance of a token, such as num.

The basic token includes:

- Terminal Symbols (TRM)- Keywords and Operators,
- Literals (LIT), and
- Identifiers (IDN).

The output of Lexical Analyzer serves as an input to Syntax Analyzer as a sequence of tokens and not the series of lexemes because during the syntax analysis phase individual unit is not vital but the category or class to which this lexeme belongs is considerable.

ALGORITHM:

```
Step 1: start the program and include the necessary header files.
Step 2: Initialize the required variables.
Step 3: Get the input file and open it in the read mode.
Step 4: In the input file if there is any identifiers then print it in the output file.
Step 5: If there is any keywords like int, char or others then print it also in the output
        file.
Step 6: If there are any constants then it is also included in the output file.
Step 7: To verify the output open the output.c file and check the idenfiers, keywords,
        constants are correctly placed which is in the input file.
Step 8: Stop the execution.
```

PRE-EXPERIMENT QUESTIONS:

1. What is tokens and explain its category.
2. What is lexical analysis in compiler design?
3. What is expression?

Program Code:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
char str[80],nkey[50][80],var[50][80],op[50][3],con[50][10],key[50][10],fn[10];
int v=0,c=0,o=0,k=0,i=0,j=0,id=0,t=0,n=0;
char opt[25]={'>','<','-',',','+','=','.','/','*','&','^','%','!','@','#','$','(',')','_'};
char
kwd[30][10]={"do","while","for","void","if","elseif","char","float","double","switch","int","long","break","case","retur
n"};
FILE *fp;
clrscr();
printf("ENTER THE INPUT FILE\n");
scanf("%s",fn);
fp=fopen(fn,"r");
while(fgets(str,80,fp))
{
i=0;
while(str[i]!='\0')
{
   if(isalpha(str[i]))
   {
               for(j=0;;j++,i++)
               {
                           if(isalnum(str[i]))
                                       var[v][j]=str[i];
                           else
                                       break;
               }
               var[v][j]='\0';
               id=0;
               for(j=0;j<32;j++)
                           if(strcmp(var[v],kwd[j])==0)
                           {
                                       strcpy(key[k],var[v]);
                                       k++;
                                       id=1;
                                       break;
                           }
                           if(str[i]=='('&&id!=1)
                           {
                                       strcpy(nkey[n],var[v]);
                                       n++;
                                       id=1;
                           }
                           if(id!=1)
                                       v++;
   }
   else
   if(isdigit(str[i]))
   {
               for(j=0;;j++,i++)
               {
                           if(isdigit(str[i]))
```

```
                                con[c][j]=str[i];
                        else
                                break;
                }
                con[c][j]='\0';
                c++;
        }
        else
                if(str[i]=='#')
                {
                        break;
                }
                else
                        if(str[i]=='"')
                        {
                                i++;
                                for(j=0;str[j]!='"';j++,i++)
                                        nkey[n][j]='\0';
                                n++;
                                i++;
                        }
                        else
                        {
                                if(str[i]==str[i+1]&&str[i]=='/')
                                        break;
                                id=0;
                                for(j=0;j<7;j++)
                                        if(str[i]==opt[j])
                                        {
                                                op[0][0]=str[i];
                                                id=1;
                                                break;
                                        }
                                        if(id==1)
                                        {
                                if(str[i]==str[i+1]&&str[i]=='+'||str[i]=='-'||str[i]=='_'||str[i]=='|')
                                {
                                        op[0][1]=str[i];
                                        op[0][2]='\0';
                                        i++;
                                }
                                else
                                        op[0][1]='\0';
                                        o++;
                }
                i++;
         }
       }
        if(str[i]==EOF)
                break;
       }
       fclose(fp);
       fp=fopen("output.c","w");
       for(i=0;i<v;i++)
        if(strcmp(var[i],var[j])==0)
                var[j][o]=' ';
                fprintf(fp,"IDENTIFIERS");
```

```
for(j=0;j<v;j++)
{
   fprintf(fp,"%s\n",var[j]);
   t++;
}
fprintf(fp,"KEYWORDS");
for(j=0;j<n;j++)
   fprintf(fp,"%s\n",nkey[j]);
fprintf(fp,"CONSTANTS");
for(j=0;j<c;j++)
   fprintf(fp,"%s\n",con[j]);
getch();
}
```

INPUT FILE

```
#include<stdio.h>
void main()
{
int a,b,c;
clrscr();
printf("enter the values");
scanf("%d%d",&a,&b);
c=a+b;
printf("%d",c);
getch();
}
```

OUTPUT:

```
IDENTIFIERS
a
b
c
e

KEYWORDS
main
clrscr
printf
scanf
printf
getch
CONSTANTS
```

POST EXPERIMENT QUESTIONS:

1. Differentiate between Tokens, Lexemes, and Pattern?
2. How many tokens are there in given expression: printf (“%d %d”, hello”); ?
3. How are regular sets different from non-regular sets?

LAB EXPERIMENT 3

OBJECTIVE:

Write a Program to implement of Lexical Analyzer using Lex Tool

BRIEF DESCRIPTION:

Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC(Yet Another Compiler Compiler).

ALGORITHM:

Step 1: start the program and declare the required variables in the declaration part.
Step 2: Define the regular definitions for identifier and number.
Step 3: In the rule part, if # is followed by any character, then print preprocessor.
Step 4: Next declare all the keyword, if any one of the above keyword is matched then return the variable is identifier.
Step 5: If the rule for identifier is matched from the above regular definition then return the variable is identifier.
Step 6: If the rule for number is matched from the above regular definition then return the variable is number.
Step 7: In the procedure part , get the name of the input file & process the rule part & print the result.
Step 8: Compile the file using the command.
Step 9: Stop the execution.

PRE-EXPERIMENT QUESTIONS:

1. What is the use of Lex tool?
2. How to compile lex?
3. What is the full form of yacc?

PROGRAM CODE:

```
%{
 int comment,store[10],no=0;
%}
identifier [a-zA-Z]*
no [0-9]*
%%
#.*    {printf("\n %s is a preprocessor directive",yytext);}
main {printf ("\n\t%s is keyword",yytext);}
int |
float |
char |
double |
do |
for |
goto {printf("\n\t%s is keywork",yytext);}
"/*"  {comment=1;}
"*/"  {comment=0;}
{identifier} {if(!comment) printf("\n%s is identifier",yytext);}
\".*\" {if(!comment) printf("\n\t%s is a string",yytext);}
{no} {if(!comment) printf("\n\t%s is anumber",yytext);}
\)(\;) {if(!comment) printf("\n\t");ECHO;printf("\n");}
\(       ECHO;
= {if(!comment) printf("\n\t%s is an assignment operator",yytext);}
\<= |
\>= |
\<  |
\>  {if(!comment) printf("\n\t%s is a relational operator",yytext);}
%%


int main(int argc,char **argv)
{
if(argc > 1)
{
FILE *file;
file = fopen(argv[1],"r");
if(!file)
{
printf("could not open %s\n",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n# of the line number is %d",no);
return 0;
}
int yywrap()
{
return 0;
}
```

OUTPUT:

```
[cseuser148@AECKUMLIN ~]$ lex lex1.l
[cseuser148@AECKUMLIN ~]$ cc lex.yy.c
[cseuser148@AECKUMLIN ~]$ ./a.out ex.c

 #include<stdio.h> is a preprocessor directive

 #include<conio.h> is a preprocessor directive

        main is keyword()
{

        int is keywork
a is identifier
        = is an assignment operator
        10 is anumber;

        float is keywork
abc is identifier;

        char is keywork
name is identifier[
        10 is anumber];
}
```

POST EXPERIMENT QUESTIONS:

1. What is the structure of Lex compiler?
2. What is lex and YACC specification?
3. What is the difference between lex compiler and C compiler?

LAB EXPERIMENT 4 a

OBJECTIVE:

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

ALGORITHM

Lex:

Step 1.Start the program and {Declaration and regular definition] Define header files to include first section

Step 2. [translation rule] Tokens generated are used in yacc files [a-z A-Z] alphabets are returned 0-9 one or more combinations of integers

Yacc:

Step 1. Accept token generated in lex part as input

Step 2. Specify the order of procedure

Step 3. Define rules with end points

Step 4. Parse input string from standard input by calling yyparse() main function.

Step 5. Print the result of any rules defined matches as arithmetic expression as valid

Step 6. If none of the rule defined matches print arithmetic expression is invalid and stop the execution.

POST EXPERIMENT QUESTIONS:

1. What is the structure of Lex compiler?
2. What is lex and YACC specification?
3. What is the difference between lex compiler and C compiler?

Program code:

LEX part

```
%{
 #include<string.h>
 int valid,i,j,k,temp,temp1,top=1,num[40];
 char arr[40][10],st[80],op[40];
%}
%%
[a-zA-Z][a-zA-Z0-9]* {
 strcpy(arr[j++],yytext);
 if(st[top-1]=='i')
 valid=1;
 else if(st[top-1]=='o')
 st[top-1]='i';
 else
 st[top++]='i';
 }
[0-9]+ {
 num[k++]=atoi(yytext);
 if(st[top-1]=='i')
 valid=1;
 else if(st[top-1]=='o')
 st[top-1]='i';
 else
 st[top++]='i';
 }
[\-+/*%^&|=] {
 op[temp++]=yytext[0];
 if(st[top-1]=='i')
 st[top-1]='o';
 else
 valid=1;
 }
"(" {
 if(st[top-1]=='('||st[top-1]=='$'||st[top-1]=='0')
 st[top++]='(';
 else
 valid=1;
 }
")" {
 if(st[top-1]=='(')
 top--;
 else if(st[top-1]=='i'&&st[top-2]=='(')
 {
 top-=2;
 if(st[top-1]=='o')
 st[top-1]='i';
 else if(st[top-1]=='i')
 valid=1;
 else
 st[top++]='i';
 }
 else
 valid=1;
 }
 valid=1;
\n return 1;
%%
```

```
int check()
{
 if(!(temp|k|j))
 return 0;
 if(valid)
 return 0;
 if(top==2&&st[top-1]=='i')
 top--;
 if(top==1&&st[top-1]=='$')
 top--;
 if(top==0)
 return 1;
 return 0;              }
int main()
{
 st[top-1]='$';
 printf("\nEnter the expression\n");
 yylex();
 if(check())
  {
 printf("\nValid expression!\n");
 if(j>0)
 printf("\nIdentifiers present are ");
 for(i=0;i<j;i++)
 printf("\n%s",arr[i]);
 if(k>0)
 printf("\nNumbers used are ");
 for(i=0;i<k;i++)
 printf("\n%d",num[i]);
 if(temp>0)
 printf("\nOperators present are ");
 for(i=0;i<temp;i++)
 printf("\n%c",op[i]);
 } else
 printf("\nInvalid expression!\n");
}
int yywrap(void)
{
return 1;
}
```

OUTPUT:

```
[egsp@localhost shaif]$ lex exno44.l
[egsp@localhost shaif]$ cc lex.yy.c
[egsp@localhost shaif]$ ./a.out
Enter the expression
A=B+C*D/E
Valid expression!
Identifiers present are
A
B
C
D
E
Operators present are
=
+
*
/
```

LAB EXPERIMENT 4b

OBJECTIVE:

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

ALGORITHM:

```
1. Include header file y.tab.h
2. Define tokens
3. Declare [a-z] as L
            [0-9] as D
4. Declare variable L D P
5. Call function yyerror()
6. If error exist then print “invalid” and exit
7. In main() call yyparse(), if yyparse() print “valid variable”
```

PROGRAM:

LEX part

```
%{

#include"y.tab.h"

%}

%%

[a-zA-Z] return letter;

[0-9] return digit;

.return yytext[0];

\n return 0;

%%

int yywrap()

{

return 1;
```

```
}
yacc part
%{
#include<stdio.h>
#include<stddef.h>
#include <string.h>
int valid = 1;
%}
%token digit letter
%%
start : letter s
s: letter s
| digit s
|
;
%%
int yyerror()
{
printf("\n Its not a identifier!\n");
valid=0;
return 0;
}
int main()
{
```

```
printf("\n Enter a name to tested for identifier");

yyparse();

if(valid)

{

printf("\n It is a identifier!\n");

}

}
```

Output :

```
 [egsp@localhost shaif]$ lex exno4b.l

[egsp@localhost shaif]$ yacc -d exno4b.y

[egsp@localhost shaif]$ cc lex.yy.c y.tab.c

[egsp@localhost shaif]$ ./a.out

Enter a name to tested for identifierguhan

It is a identifier!
```

POST EXPERIMENT QUESTIONS:

1. What are the various functions of lexical analyzer?
2. Which mathematical model is used in the lexical analyzer?
3. What elements are included in non-tokens components?

LAB EXPERIMENT 4c

OBJECTIVE:

Write a Program for implementation of calculator using YACC tool.

BRIEF DESCRIPTION:
Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). LEX is commonly used with the YACC parser generator. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language**.**

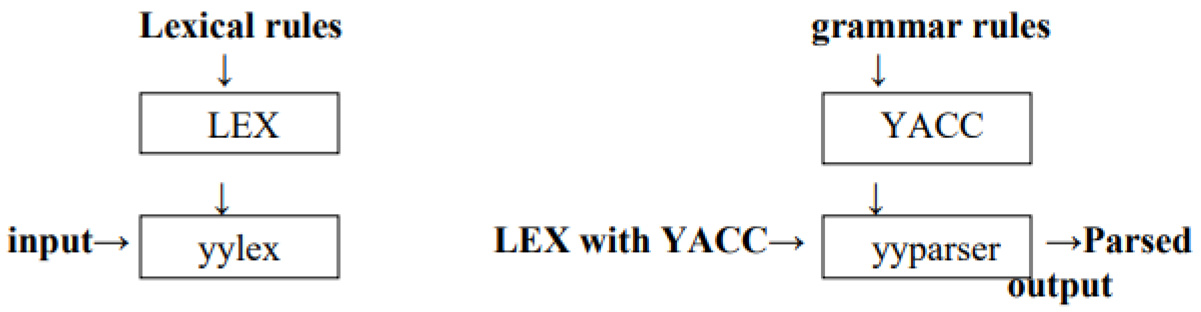
YACC is written in portable C. The class of specifications accepted is a very general one LALR (1) grammars with disambiguating rules.

PRE-EXPERIMENT QUESTIONS:

1. What is the purpose of YACC?
2. In which phase YACC is used?
3. Which table is created by YACC?

<u>Explanation:</u>

Special Functions
• yytext– where text matched most recently is stored
• yyleng– number of characters in text most recently matched
• yylval– associated value of current token
• yymore()– append next string matched to current contents of yytext
• yyless(n)– remove from yytext all but the first n characters
• unput(c) – return character c to input stream
• yywrap()– may be replaced by user and the yywrap method is called by the lexical analyzer whenever it inputs an EOF as the first character when trying to match a regular expression

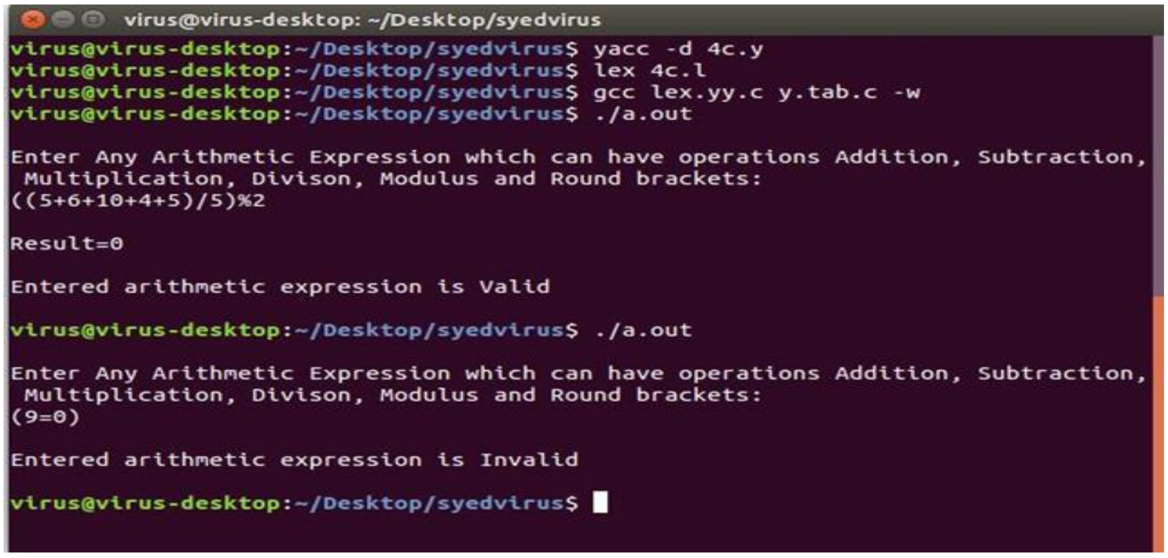


PROGRAM:
Lex Part

```
%{
#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;
}
Yacc Part
%{
#include <stdio.h>
int flag = 0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
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; }
;
%%
void main() {
printf("\nEnter any arithmetic expression (supports +, -, *, /, %, and brackets):\n");
yyparse();
if (flag == 0)
printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror(const char *s) {
printf("\nEntered arithmetic expression is Invalid\n\n");
flag = 1;
}
```

OUTPUT:



```
virus@virus-desktop: ~/Desktop/syedvirus
virus@virus-desktop:~/Desktop/syedvirus$ yacc -d 4c.y
virus@virus-desktop:~/Desktop/syedvirus$ lex 4c.l
virus@virus-desktop:~/Desktop/syedvirus$ gcc lex.yy.c y.tab.c -w
virus@virus-desktop:~/Desktop/syedvirus$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Divison, Modulus and Round brackets:
((5+6+10+4+5)/5)%2

Result=0

Entered arithmetic expression is Valid

virus@virus-desktop:~/Desktop/syedvirus$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Divison, Modulus and Round brackets:
(9=0)

Entered arithmetic expression is Invalid

virus@virus-desktop:~/Desktop/syedvirus$ 
```

POST EXPERIMENT QUESTIONS:

1. Which parsing technique is used in YACC?
2. What are the parts of a YACC file?
3. Is YACC a bottom-up parser?

LAB EXPERIMENT 5

OBJECTIVE:

Write a program to Convert The BNF rules into Yacc form and write co to generate abstract syntax tree.

ALGORITHM :

```
Step1: Start  the program.
Step2: declare the declarations as a header file
            {include<ctype.h>}
Step3: token digit
Step4: define the translations rules like line, expr, term, factor
            Line:exp „\n‟ {print("\n %d \n",$1)}
            Expr:expr‟+‟ term ($$=$1=$3}
            Term:term „+‟ factor($$ =$1*$3}
            Factor
            Factor:‟(„enter‟) „{$$ =$2)
            % %
Step5: define the supporting C routines
Step6: Stop the execution.
```

PRE EXPERIMENT QUESTIONS:

1. What is BNF in compiler construction?
2. What is the difference between BNF and CNF?
3. What is the difference between a parse tree and an abstract syntax tree?

PROGRAM CODE

<int.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];
%%
```

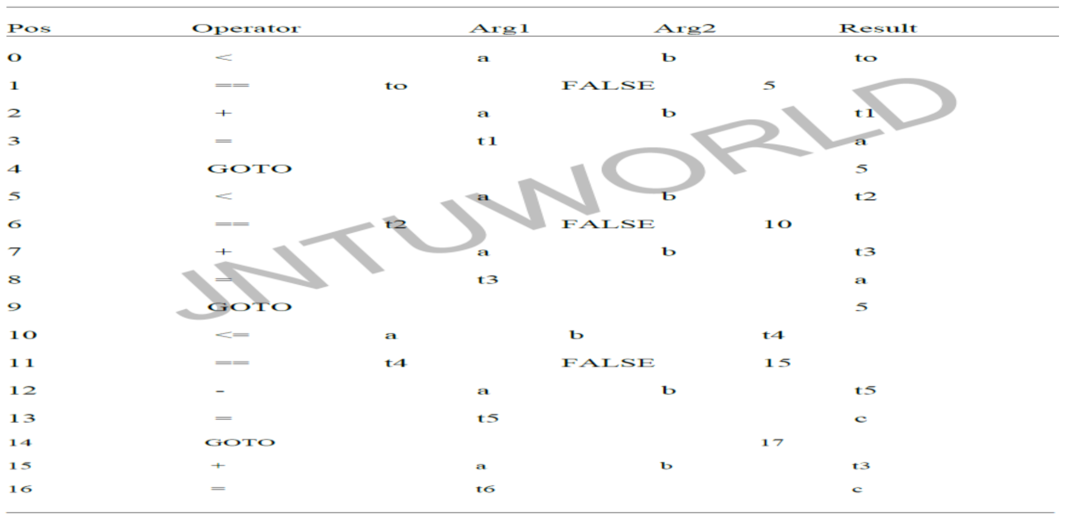
<int.y>

```
%{
#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 Arg1 Arg2 Result" "\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].result);
}
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);
}
```
Input:
```
$vi test.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;
}
}
```

Output:

```
$lex int.l
$yacc –d int.y
$gcc lex.yy.c y.tab.c –ll –lm
$./a.out test.c
```



```
Pos        Operator              Arg1       Arg2         Result
0            <                   a          b            to
1            ==         to             FALSE       5
2            +                   a          b            t1
3            =                   t1                      a
4           GOTO                                         5
5            <                   a          b            t2
6            ==         t2             FALSE       10
7            +                   a          b            t3
8            =                   t3                      a
9           GOTO                                         5
10           <=         a              b           t4
11           ==         t4             FALSE       15
12           -                   a          b            t5
13           =                   t5                      c
14          GOTO                                   17
15           +                   a          b            t3
16           =                   t6                      c
```

POST EXPERIMENT QUESTIONS

1. **How do you draw an abstract syntax tree?**
2. **What are the applications of Abstract Syntax Tree?**
3. **What is abstract syntax tree and dag?**

LAB EXPERIMENT 6

OBJECTIVE:

Write a Program to implement the concept of type checking

BRIEF DESCRIPTION:

Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language. It allows the programmer to limit what types may be used in certain circumstances and assigns types to values. The type-checker determines whether these values are used appropriately or not.

It checks the type of objects and reports a type error in the case of a violation, and incorrect types are corrected. Whatever the compiler we use, while it is compiling the program, it has to follow the type rules of the language. Every language has its own set of type rules for the language. We know that the information about data types is maintained and computed by the compiler.

The information about data types like INTEGER, FLOAT, CHARACTER, and all the other data types is maintained and computed by the compiler. The compiler contains modules, where the type checker is a module of a compiler and its task is type checking.

PRE-EXPERIMENT QUESTIONS:

1. Is type checking done before parsing?
2. What are the two rules of type checking?
3. Why type checking is performed?

ALGORITHM

Step1: Track the global scope type information (e.g. classes and their members)

Step2: Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

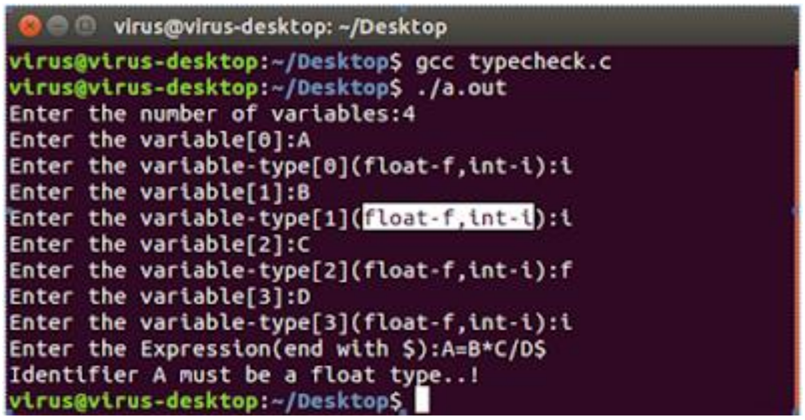
Step3: If type found correct, do the operation

Step4: Type **mismatches**, semantic error will be notified

Program Code:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int n,i,k,flag=0;
char vari[15],typ[15],b[15],c;
printf("Enter the number of variables:");
scanf(" %d",&n);
for(i=0;i<n;i++)
{
printf("Enter the variable[%d]:",i);
scanf(" %c",&vari[i]);
printf("Enter the variable-type[%d](float-f,int-i):",i);
scanf(" %c",&typ[i]);
if(typ[i]=='f')
flag=1;
}
printf("Enter the Expression(end with $):");
i=0;
getchar();
while((c=getchar())!='$')
{
b[i]=c;
i++;  }
k=i;
for(i=0;i<k;i++)
{
if(b[i]=='/')
{
flag=1;
break;  }  }
for(i=0;i<n;i++)
{
if(b[0]==vari[i])
{
if(flag==1)
{
if(typ[i]=='f')
{  printf("\nthe datatype is correctly defined..!\n");
break;  }
else
{  printf("Identifier %c must be a float type..!\n",vari[i]);
break;  }  }
else
{  printf("\nthe datatype is correctly defined..!\n");
break;  }  }
}
return 0;
}
```

OUTPUT:



```
virus@virus-desktop: ~/Desktop
virus@virus-desktop:~/Desktop$ gcc typecheck.c
virus@virus-desktop:~/Desktop$ ./a.out
Enter the number of variables:4
Enter the variable[0]:A
Enter the variable-type[0](float-f,int-i):i
Enter the variable[1]:B
Enter the variable-type[1](float-f,int-i):i
Enter the variable[2]:C
Enter the variable-type[2](float-f,int-i):f
Enter the variable[3]:D
Enter the variable-type[3](float-f,int-i):i
Enter the Expression(end with $):A=B*C/D$
Identifier A must be a float type..!
virus@virus-desktop:~/Desktop$
```

POST EXPERIMENT QUESTIONS:

1. What are the disadvantages of static type checking?
2. What is type compatibility?
3. What are the major types of type checking?

LAB EXPERIMENT 7

OBJECTIVE:

Write a program to implement the concept of Data flow Analysis

BRIEF DESCRIPTION

Data flow analysis is a global code optimization technique. The compiler performs code optimization efficiently by collecting all the information about a program and distributing it to each block of its **control flow graph** **(CFG)**. This process is known as data flow analysis.
Since optimization has to be done on the entire program, the whole program is examined, and the compiler collects information.
.

Basic Technologies

Below are some basic terminologies related to data flow analysis.

- **Definition Point-** A definition point is a point in a program that defines a data item.
- **Reference Point-** A reference point is a point in a program that contains a reference to a data item.
- **Evaluation Point-** An evaluation point is a point in a program that contains an expression to be evaluated.

ALGORITHM:

1. Start the program
2. Read the total number of expressions
3. Read left and right side of each expressions
4. Display expressions with the line number
5. Display data floe movement with particular expressions
6. Stop the execution

PRE-EXPERIMENT QUESTIONS:

1. What is data flow analysis with its properties?
2. What is meant by live variable?
3. How to do live variable analysis?

Program Code:

```
#include <stdio.h>
#include <conio.h>
#include <string.h >
struct op
{
char l[20];
char r[20];
}
op[10], pr[10];

void main()
{
int a, i, k, j, n, z = 0, m, q,lineno=1;
char * p, * l;
char temp, t;
char * tem;char *match;
clrscr();
printf("enter no of values");
scanf("%d", & n);
for (i = 0; i < n; i++)
{
printf("\tleft\t");
scanf("%s",op[i].l);
printf("\tright:\t");
scanf("%s", op[i].r);
}
printf("intermediate Code\n");
for (i = 0; i < n; i++)
{   printf("Line No=%d\n",lineno);
printf("\t\t\t%s=", op[i].l);
printf("%s\n", op[i].r);lineno++;
}
printf("***Data Flow Analysis for the Above Code ***\n");
for(i=0;i<n;i++)
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
match=strstr(op[j].r,op[i].l);
if(match)
{
printf("\n %s is live at  %s \n ", op[i].l,op[j].r);
}
}
}}
```

OUTPUT:

```
Enter no of values
4
        left    a
        right:  a+b
        left    b
        right:  a+c
        left    c
        right:  a+b
        left    d
        right:  b+c+d
 Line No=1
 a=a+b
 Line No=2
 b=a+c
Line No=3
                        c=a+b
Line No=4
                        d=b+c+d
***Data Flow Analysis for the Above Code ***

 a is live at  a+b

 a is live at  a+c

 a is live at  a+b

 b is live at  a+b

 b is live at  a+b

 b is live at  b+c+d


  c is live at  a+c

  c is live at  b+c+d

  d is live at  b+c+d
```

POST EXPERIMENT QUESTIONS:

1. What is the use of live variable?
2. What is global data flow analysis with its use in code optimization?
3. What are data flow analysis functions?

LAB EXPERIMENT 8

OBJECTIVE:

Write a C program for the implementation of Stack Allocation Strategy

BRIEF DESCRIPTION:

Compiler is a program that converts HLL(High-Level Language) to LLL(Low-Level Language) like machine language. In a compiler, there is a need for storage allocation strategies in Compiler design because it is very important to use the right strategy for storage allocation as it can directly affect the performance of the software.

Storage Allocation Strategies

There are mainly three types of Storage Allocation Strategies:

1. Static Allocation
2. Heap Allocation
3. Stack Allocation

ALGORITHM:

```
Step 1. Prompt the user  with options
            (1)PUSH
            (2)POP
            (3)DISPLAY
            (4)EXIT
Step 2 .  If the option is push then
Step2.1.        If (TOP == MAX) Then [Check for overflow]
Step 2.2.       Print: Overflow
Step 2.3. Else
Step 2.4.       Set TOP = TOP + 1 [Increment TOP by 1]
Step 2.5.       Set STACK[TOP] = ITEM [Assign ITEM to top of STACK]
Step 2.6.       Print: ITEM inserted
              [End of If]
Step 2.7. Exit
Step 3 .   If the option is pop then
Step 3 .1.      If (TOP == 0) Then [Check for underflow]
Step 3 .2.              Print: Underflow
Step 3 .3.      Else
Step 3 .4.              Set ITEM = STACK[TOP] [Assign top of STACK to ITEM]
Step 3 .5.              Set TOP = TOP - 1 [Decrement TOP by 1]
Step 3 .6.       Print: ITEM deleted
              [End of If]
Step 3 .7. Exit
Step 4.    If the option is Display  then
Step 4.1         Display the Stack elements one by one.
Step5.     If the option is Exit  then
 Step5.1        Exit from the Program.
```

PRE-EXPERIMENT QUESTIONS:

1. What is storage allocation strategies in compiler design?
2. What is static allocation in compiler design?
3. What is the difference between static and heap storage allocation?

PROGRAM:

```
#include <stdio.h>
#define MAXSIZE 5

struct stack
{
   int stk[MAXSIZE];
   int top;
};
typedef struct stack STACK;
STACK s;

void push(void);
int  pop(void);
void display(void);

void main ()
{
   int choice;
   int option = 1;
   s.top = -1;

   printf ("STACK OPERATION\n");
   while (option)
   {
      printf ("------------------------------------------\n");
      printf ("      1    -->    PUSH               \n");
      printf ("      2    -->    POP               \n");
      printf ("      3    -->    DISPLAY               \n");
      printf ("      4    -->    EXIT           \n");
      printf ("------------------------------------------\n");

      printf ("Enter your choice\n");
      scanf    ("%d", &choice);
      switch (choice)
      {
      case 1:
         push();
         break;
      case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            return;
        }
        fflush (stdin);
        printf ("Do you want to continue(Type 0 or 1)?\n");
        scanf    ("%d", &option);
    }
}
/*  Function to add an element to the stack */
void push ()
{
    int num;
    if (s.top == (MAXSIZE - 1))
    {
        printf ("Stack is Full\n");
        return;
    }
    else
    {
        printf ("Enter the element to be pushed\n");
        scanf ("%d", &num);
        s.top = s.top + 1;
        s.stk[s.top] = num;
    }
    return;
}
/*  Function to delete an element from the stack */
int pop ()
{
    int num;
    if (s.top == - 1)
    {
        printf ("Stack is Empty\n");
        return (s.top);
    }
    else
    {
        num = s.stk[s.top];
        printf ("poped element is = %dn", s.stk[s.top]);
        s.top = s.top - 1;
    }
    return(num);
}
```

```
/*  Function to display the status of the stack */
void display ()
{
   int i;
   if (s.top == -1)
   {
      printf ("Stack is empty\n");
      return;
   }
   else
   {
      printf ("\n The status of the stack is \n");
      for (i = s.top; i >= 0; i--)
      {
         printf ("%d\n", s.stk[i]);
      }
   }
   printf ("\n");
}
```

OUTPUT:

```
STACK OPERATION
------------------------------------------
     1    -->    PUSH
     2    -->    POP
     3    -->    DISPLAY
     4    -->    EXIT
------------------------------------------
Enter your choice
1
Enter the element to be pushed
34
Do you want to continue(Type 0 or 1)?
0
$ a.out
STACK OPERATION
------------------------------------------
     1    -->    PUSH
     2    -->    POP
     3    -->    DISPLAY
     4    -->    EXIT
------------------------------------------
Enter your choice
1
Enter the element to be pushed
34
Do you want to continue(Type 0 or 1)?
```

```
1
------------------------------------------
     1   -->   PUSH
     2   -->   POP
     3   -->   DISPLAY
     4   -->   EXIT
------------------------------------------
Enter your choice
2
poped element is = 34
Do you want to continue(Type 0 or 1)?
1
------------------------------------------
     1   -->   PUSH
     2   -->   POP
     3   -->   DISPLAY
     4   -->   EXIT
------------------------------------------
Enter your choice
3
Stack is empty
Do you want to continue(Type 0 or 1)?
1
------------------------------------------
     1   -->   PUSH
     2   -->   POP
     3   -->   DISPLAY
     4   -->   EXIT
------------------------------------------
Enter your choice
1
Enter the element to be pushed
50
Do you want to continue(Type 0 or 1)?
1
------------------------------------------
     1   -->   PUSH
     2   -->   POP
     3   -->   DISPLAY
     4   -->   EXIT
------------------------------------------
Enter your choice
1
Enter the element to be pushed
60
Do you want to continue(Type 0 or 1)?
1
------------------------------------------
```

```
     1   -->   PUSH
     2   -->   POP
     3   -->   DISPLAY
     4   -->   EXIT
----------------------------------------
Enter your choice
3

The status of the stack is
60
50

Do you want to continue(Type 0 or 1)?
1
----------------------------------------
     1   -->   PUSH
     2   -->   POP
     3   -->   DISPLAY
     4   -->   EXIT
----------------------------------------
Enter your choice
4
```

POST-EXPERIMENT QUESTIONS:

1. How is stack allocated?
2. Who allocates the stack?
3. What is difference between stack and heap?
4. How much can you allocate on the stack?

LAB EXPERIMENT 9

OBJECTIVE:

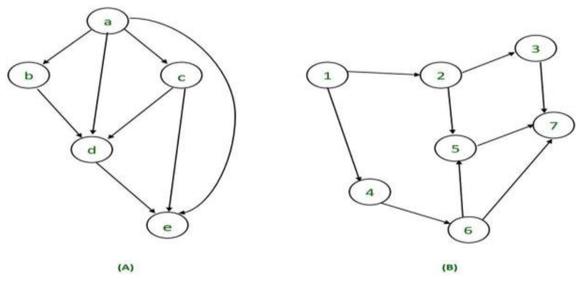
Write a C program for construction of DAG

BRIEF DESCRIPTION:

The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

- Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.

Examples of directed acyclic graph :



ALGORITHM:

There are three possible scenarios for building a DAG on three address codes:

Case 1 – x = y op z
Case 2 – x = op y
Case 3 – x = y
Directed Acyclic Graph for the above cases can be built as follows :

Step 1
- If the y operand is not defined, then create a node (y).
- If the z operand is not defined, create a node for case(1) as node(z).

Step 2
- Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
- For the case (2), see if there is a node operator (OP) with one child node (y).
- Node n will be node(y) in case (3).

Step 3
- Remove x from the list of node identifiers.
- Add x to the list of attached identifiers for node n.

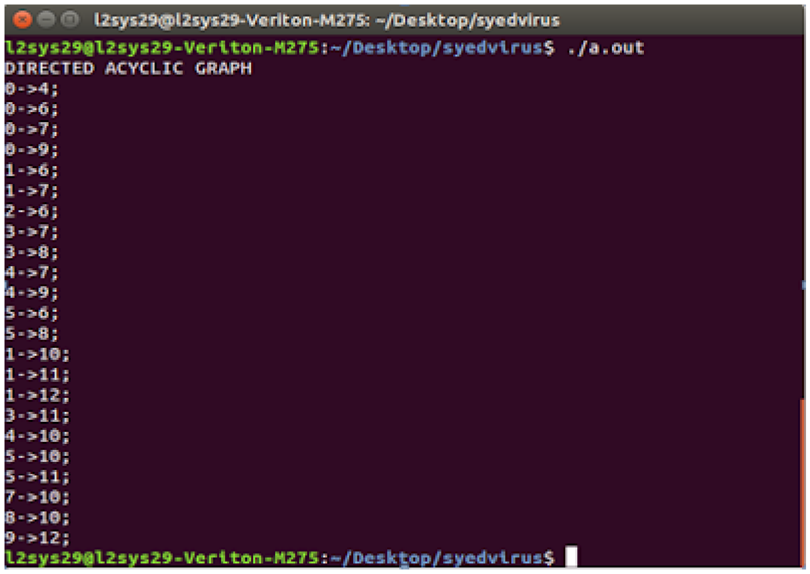
PRE-EPERIMENT QUESTIONS:

1. What is DAG in compiler design?
2. What is the purpose of DAG?
3. What is DAG and syntax tree?

PROGRAM CODE :

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define MIN_PER_RANK 1
#define MAX_PER_RANK 5
#define MIN_RANKS 3
#define MAX_RANKS 5
#define PERCENT 30
void main()
{
int i,j,k,nodes=0;
srand(time(NULL));
int ranks=MIN_RANKS+(rand()%(MAX_RANKS-MIN_RANKS+1));
printf("DIRECTED ACYCLIC GRAPH\n");
for(i=1;i<ranks;i++)
{
int new_nodes=MIN_PER_RANK+(rand()%(MAX_PER_RANK-MIN_PER_RANK+1));
for(j=0;j<nodes;j++)
for(k=0;k<new_nodes;k++)
if((rand()%100)<PERCENT)
printf("%d->%d;\n",j,k+nodes);
nodes+=new_nodes;
}
}
```

OUTPUT:



```
l2sys29@l2sys29-Veriton-M275: ~/Desktop/syedvirus
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ ./a.out
DIRECTED ACYCLIC GRAPH
0->4;
0->6;
0->7;
0->9;
1->6;
1->7;
2->6;
3->7;
3->8;
4->7;
4->9;
5->6;
5->8;
1->10;
1->11;
1->12;
3->11;
4->10;
5->10;
5->11;
7->10;
8->10;
9->12;
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ 
```

POST-EXPERIMENT QUESTIONS:

1. What is the full form of DAG?
2. Is DAG an algorithm?
3. What is DAG used for?

LAB EXPERIMENT 9b

OBJECTIVE:

Write a C program to implement Code Optimization Techniques.

ALGORITHM

Input: Set of ‘L’ values with corresponding ‘R’ values.
Output: Intermediate code & Optimized code after eliminating common expressions.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}
op[10],pr[10];
void main()
{
int a,i,k,j,n,z=0,m,q;
char *p,*l;
char temp,t;
char *tem;
clrscr();
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
op[i].l=getche();
printf("\tright: ");
scanf("%s",op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
temp=op[i].l;
for(j=0;j<n;j++)
{
```

```
p=strchr(op[j].r,temp);
if(p)
{
pr[z].l=op[i].l;
strcpy(pr[z].r,op[i].r);
z++;
}
}
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++)
{
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
p=strstr(tem,pr[j].r);
if(p)
{
t=pr[j].l;
pr[j].l=pr[m].l;
for(i=0;i<z;i++)
{
l=strchr(pr[i].r,t) ;
if(l)
{
a=l-pr[i].r;
printf("pos: %d",a);
pr[i].r[a]=pr[m].l;
}
}
}
}
}
printf("Eliminate Common Expression\n");
for(i=0;i<z;i++)
{
printf("%c\t=",pr[i].l);
printf("%s\n",pr[i].r);
}
for(i=0;i<z;i++)
```

```
{
for(j=i+1;j<z;j++)
{
q=strcmp(pr[i].r,pr[j].r);
if((pr[i].l==pr[j].l)&&!q)
{
pr[i].l='\0';
strcpy(pr[i].r,'\0');
}
}
}
printf("Optimized Code\n");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
getch();
}
```

OUTPUT:

```
Enter the Number of Values: 5
Left: a right: 9
Left: b right: c+d
Left: e right: c+d
Left: f right: b+e
Left: r right: f
Intermediate Code : a=9
b=c+d
e=c+d
f=b+e
r=:f
After Dead Code Elimination
b =c+d
e =c+d
f =b+e
r =:f
Eliminate Common Expression
b =c+d
b =c+d
f =b+b
r =:f
Optimized Code
b=c+d
f=b+b
r=:f
```

LAB EXPERIMENT 9C

OBJECTIVE:

Write a program for implementation of Code Optimization Technique in for and do-while loop using C++.

ALGORITHM:

step1. Generate the program for factorial program using for and do-while loop to specify optimization technique.
step 2. In for loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and specified increment / decrement operation is performed.
step 3. The for loop operation is activated till the condition failure.
step 4. In do-while loop the variable is initialized and the statements are executed then the condition checking and increment / decrement operation is performed.
step 5. When comparing both for and do-while loop for optimization dowhile is best because first the statement execution is done then only the condition is checked. So, during the statement execution itself we can find the inconvenience of the result and no need to wait for the specified condition result.
step 6. Finally when considering Code Optimization in loop do-while is best with respect to performance.

PROGRAM:

Before:

Using for:

```
#include<iostream.h>
#include <conio.h>
int main()
{
int i, n;
int fact=1;
cout<<"\nEnter a number: ";
cin>>n;
for(i=n ; i>=1 ; i--)
fact = fact * i;
cout<<"The factoral value is: "<<fact;
}
```

OUTPUT:

```
Enter the number: 5
The factorial value is: 120
```

After:

Using do-while:

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int n,f;
f=1;
cout<<"Enter the number:\n";
cin>>n;
do
{
f=f*n;
n--;
}while(n>0);
cout<<"The factorial value is:"<<f;
getch();
}
```

OUTPUT:

```
Enter the number: 3
The factorial value is: 6
```

LAB EXPERIMENT 10

OBJECTIVE:

Write a C program for implementation of the back end of the compiler

BRIEF DESCRIPTION:

The back end is responsible for the CPU architecture specific optimizations and for code generation. The main phases of the back end include the following: Machine dependent optimizations: optimizations that depend on the details of the CPU architecture that the compiler targets.

ALGORITHM:-

```
Step 1:  Start the program.
Step 2:  Load one 16 bit number in Ax register from memory
Step 3:  Add the content of Ax register with another 16 bit number stored in another
         memory location using ADD instruction.
Step 4:  Store the result in memory  location.
Step 5:  Load one 16 bit number in AX register from memory.
Step 6:  Subtract the content of AX register with another 16-bit number stored in another
         memory location using SUB instruction.
Step 7:  Store the result in memory location.
Step 8:  Stop the execution.
```

PRE-EPERIMENT QUESTIONS :

1. What is the output of a back end module of a compiler?
2. How does backend compiler work?
3. What is difference between front end and back end?

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char ch,s1,s2,op,t,index,str[20][20],exp[20],temp[20];
int s=0,i=0,j=0,k=0;
clrscr();
printf("\nENTER THE THREE ADDRESS CODE\n");
while((ch=getchar())!='$')
{
if(ch!='\n')
{
str[s][i]=ch;
i++;
}
else
{
str[s][i]=NULL;
printf("%s\n",str[s]);
s++;
i=0;
}
}
printf("%d %d",s,i);
putchar(ch);
printf("FORMAT:OPCODE,SRC,DEST\n");
for(j=0;j<s||j<i;j++)
{
i=0;
strcpy(exp,str[j]);
strcpy(temp,exp,2);
}
if((strcmp(temp,"if"))==0)
{
i=i+3;
s1=exp[i];
op=exp[++i];
s2=exp[++i];
i=i+2;
for(k=0;exp[i]!=' ';k++,i++)
temp[k]=exp[i];
temp[k]=NULL;
t=exp[++i];
```

```
if((strcmp,"goto")==0)
{
if(isalpha(s2))
{
printf("CMP%c%c\n",s1,s2);
printf("CJ%c%c\n\n",op,t);
}
else
printf("CS%c%c\n\n",op,t);
}
}
else
{
if(isalpha(exp[i]))
{
t=exp[i];
if(exp[++i]=='=')
{
if(isalnum(exp[++i]))
{
s1=exp[i];
op=exp[++i];
if(op=='+'||op=='-'||op=='*'||op=='/')
{
s2=exp[++i];
if(isalpha(s1))
{
printf("MOV %c,RO\n",s1);
}
else
printf("MOV #%c,RO\n",s1);
switch(op)
{
case '+':
printf("ADD #%c,RO\n",s2);
break;
case '-':
printf("SUB #%c,RO\n",s2);
break;
case '*':
printf("MUL RO,#%c\n",s2);
break;
}
case '/':
printf("DIV RO,#%c\n",s2);
break;
}
if(isalpha(s2))
printf("MOV %c,RO\n",s2);
printf("MOV RO,%c\n\n",t);
}
else
{
if(op==NULL)
{
if(isalpha(s1))
{
printf("MOV %c,RO\n",s1);
}
else
printf("MOV #%c,RO\n",s1);
printf("MOV RO,%c\n\n",t);
}
if(exp[i]=='['&&exp[i+2]==']')
{
index=exp[i+1];
printf("MOV %c(R%c)%c\n\n",s1,index,t);
}
}
}
}
else
{
if(exp[i]=='*')
{
s1=exp[++i];
printf("MOV *R%c,%c\n\n",s1,t);
}
}
}
else
{
if(exp[i]=='{'&&exp[i+2]=='}')
{
index=exp[i+1];
if(exp[i+3]=='=')
{
s1=exp[i+4];
if(isalpha(s1))
printf("MOV %c,%c(R%c)\n\n",s1,t,index);
else
printf("MOV #%c,RO\n",s1);
printf("MOV RO,%c\n\n",t);
}
}
else
```

```
{
if(exp[i]=='*')
{
t=exp[++i];
if(exp[++i]=='=')
{
s1=exp[++i];
printf("\tMOV %c,*R%c\n\n",s1,t);
}
}
}
}
}
getch();
}
```

OUTPUT:

```
Enter the three address code
A=b+4
A=b+4
$
FORMAT:
        OPCODE, SRC, DEST
        Mov      b,    r0
        Add     #4,    r0
        Mov     r0,     a
```

POST-EXPERIMENT QUESTIONS:

1. What is the basic of code generation?
2. What are code generation tools?
3. Which model is best for code generation?

LAB EXPERIMENT 11 a

OBJECTIVE:

Write a Program for implementation of Recursive Descent Parser

BRIEF DESCRIPTION:

A recursive descent parser is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right. (The actual tree is not constructed but is implicit in a sequence of function calls.) This type of parser was very popular for real compilers in the past, but is not as popular now. The parser is usually written entirely by hand and does not require any sophisticated tools.

This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol.

PRE-EXPERIMENT QUESTIONS:

1. What is the purpose of a recursive descent parser?
2. What is the another name of recursive descent parser?
3. What type of parsing is recursive descent parser?

Program Code:

```
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();
const char *pt;
char grammar[64];
int main()
 {
 sscanf("i+i*i", "%s", grammar);
 pt = grammar;
 puts("");
 puts("Input Action");
 if (E() && *pt == '\0') {
puts("String is successfully parsed");
return 0;
}
else {
puts("Error in parsing String");
return 1;
}
}
int E()
{
 printf("%-16s E -> T E'\n", pt);
 if (T())
{
if (Edash())
 return SUCCESS;
 else
 return FAILED;
 }
 else
 return FAILED;
}
int Edash() {
 if (*pt == '+')
{
 printf("%-16s E' -> + T E'\n", pt);
 pt++;
 if (T()) {
 if (Edash())
 return SUCCESS;
 else
 return FAILED;
}
else
 return FAILED;
 }
 else
 printf("%-16s E' -> $\n", pt);
 return SUCCESS;
}
```

```
int T() {
 printf("%-16s T -> F T'\n", pt);
 if (F()) {
 if (Tdash())
 return SUCCESS;
 else
 return FAILED;
 }
 else
 return FAILED;
}

int Tdash() {
 if (*pt == '*') {
 printf("%-16s T' -> *F T'\n", pt);
 pt++;
		if (F()) {
			if (Tdash())
				return SUCCESS;
			else
				return FAILED;
		}
		else
			return FAILED;
 }
 else {
		printf("%-16s T' -> $\n", pt);
		return SUCCESS;
 }
}

int F() {
 if (*pt == '(') {
		printf("%-16s F -> (E)\n", pt);
		pt++;
		if (E()) {
			if (*pt == ')') {
				pt++;
				return SUCCESS;
			}
			else
				return FAILED;
		}
		else
			return FAILED;
 }
 else if (*pt == 'i') {
		pt++;
		printf("%-16s F -> i\n", pt);
		return SUCCESS;
 }
 else
		return FAILED;
}
```

Output:

input	production
i+i*i	E->TE’
+i*i	T’->€
+i*i	E’->+TE’
i*i	T->FT’
*i	T’->*FT’
i	T’->€
	E’->€

POST EXPERIMENT QUESTIONS:

1. Why do recursive-descent parsers have to backtrack?
2. What is the complexity of recursive descent parser?
3. Discuss the pros and cons of Recursive Descent Parser.

LAB EXPERIMENT 11 b

OBJECTIVE:

Write a Program for implementation of LL (1) Parser.

BRIEF DESCRIPTION:

The first 'L' in LL(1) stands for scanning the input from left to right, the second 'L' stands for producing a leftmost derivation, and the '1' for using one input symbol of look ahead at each step to make parsing action decisions. LL(1) grammar follows Top-down parsing method. For a class of grammars called LL(1) we can construct grammars predictive parser. That works on the concept of recursive-descent parser not requiring backtracking.

PRE-EXPERIMENT QUESTIONS:

1. Is LL(1) parser a TDP?
2. What is the purpose of 1 in LL(1)?
3. What type of functions exist in LL(1) parsing table?

Explanation:

The construction of a top-down parser is aided by FIRST and FOLLOW functions, that are associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

Rules for First computation:

1) If x is terminal, then FIRST(x)={x}
2) If X→ ε is production, then add ε to FIRST(X)
3) If X is a non-terminal and X → PQR then FIRST(X)=FIRST(P)
    a) If FIRST(P) contains ε, then
    b) FIRST(X) = (FIRST(P) – {ε}) U FIRST(QR)

Rules for Follow computation:

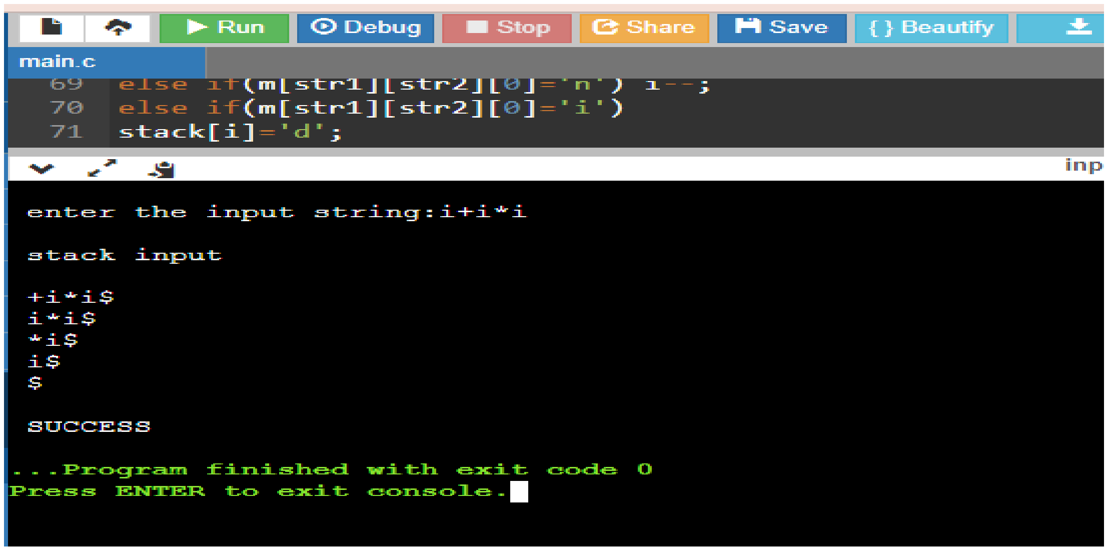
1) For Start symbol, place $ in FOLLOW(S)
2) If A→ α B, then FOLLOW(B) = FOLLOW(A)
3) If A→ α B β, then
    a) If ε not in FIRST(β),
        FOLLOW(B) = FIRST(β)
            else do,
                FOLLOW(B) = (FIRST(β)-{ε}) U FOLLOW(A)

Program Code:

```
#include<stdio.h>
#include<string.h>
char s[30],stack[20];
int main()
{
char m[5][6][3]={{"tb"," "," ","tb"," "," "},
 {" ","+tb"," "," ","n","n"},
 {"fc"," "," ","fc"," "," ",},
 {" ","n","*fc"," ","n","n"},
 {"d"," "," ","(e)"," "," "}};
int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
int i,j,k,n,str1,str2;
printf("\n enter the input string:");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
i=1;
j=0;
printf("\n stack input\n");
printf("\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
if(stack[i]=s[j])
{
i--;
j++;
}
switch(stack[i])
{
case 'e':str1=0;
 break;
case 'b':str1=1;
 break;
case 't':str1=2;
 break;
case 'c':str1=3;
break;
case 'f':str1=4;
break; }
switch(s[j]) {
case 'd':str2=0;
break;
case '+':str2=1;
 break;
case '*':str2=2;
 break;
case '(':str2=3;
 break;
case ')':str2=4;
 break;
case '$':str2=5;
 break; }
if(m[str1][str2][0]=='$') {
printf("
\n error\n");
return(0); }
else if(m[str1][str2][0]='n') i--;
else if(m[str1][str2][0]='i')
stack[i]='d';
else {
for(k=size[str1][str2]
-1;k>=0;k--
)
{
stack[i]=m[str1][str2][k];
i++; }i--; }
for(k=0;k<=i;k++)
printf("%c",stack[k]);
printf(" ");
for(k=j;k<=n;k++)
printf("%c",s[k]);
printf("\n");
}
printf("\n SUCCESS");
}
```

Output:

Students will be able to implement LL(1) parser as shown below:



```
else if(m[str1][str2][0]='n') i--;
else if(m[str1][str2][0]='i')
stack[i]='d';
```

```
enter the input string:i+i*i

stack input

+i*i$
i*i$
*i$
i$
$

SUCCESS

...Program finished with exit code 0
Press ENTER to exit console.
```

Rules for construction of parsing table:

Step 1: For each production A → α , of the given grammar perform Step 2 and Step 3.
Step 2: For each terminal symbol 'a' in FIRST(α), ADD A → α in table T[A,a], where 'A' determines row & 'a' determines column.
Step 3: If ε is present in FIRST(α) then find FOLLOW(A), ADD A → ε, at all columns 'b', where 'b' is FOLLOW(A). (T[A,b])
Step 4: If ε is in FIRST(α) and $ is the FOLLOW(A), ADD A → α to T[A,$].

POST EXPERIMENT QUESTIONS:

1. What is a lazy parser?
2. What elements makeup parsing?
3. What are the methods of parsing?

Forming Sentences – 1

A sentence is a group of words which forms a meaningful sense. A sentence is formed by following the grammatical rules of syntax. Grammatical rules vary from language to language.

Types of Sentences

Declarative Sentence, or declaration, is the most common type of sentence. It tells something.
Interrogative Sentence, or question, asks something.
Exclamatory Sentence, or exclamation, says something out of the ordinary.
Imperative Sentence, or command, tells someone to do something.

Objective

The objective of this experiment is to know how to form logically correct sentences from the given words.

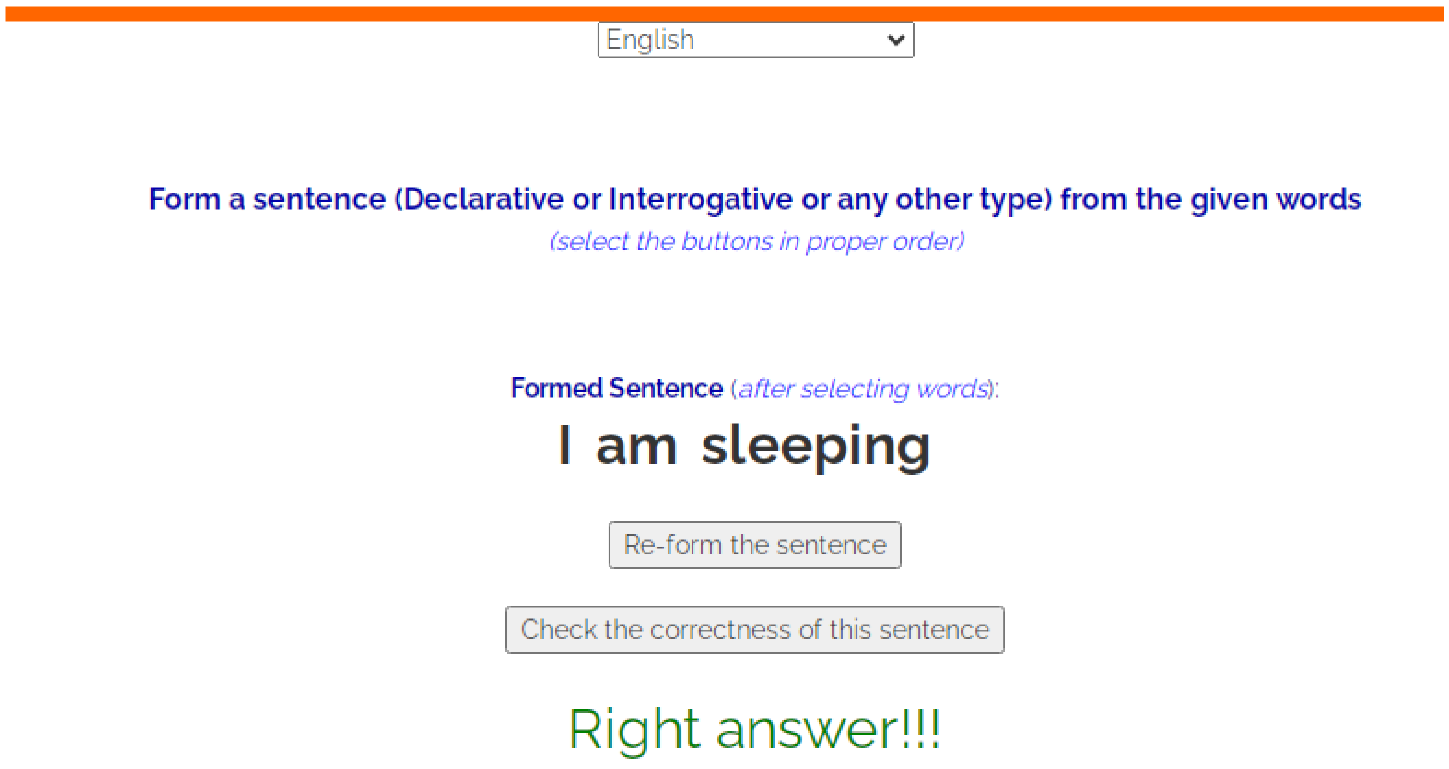
Procedure
STEP1: Select a language which you know better

STEP 2: Select the buttons which has words written on it, in a proper order

OUTPUT: Group of words in a selected order will be shown

NOTE:

If a wrong sentence is formed, Re-form the sentence is available for re-setting.
You can check whether the formed sentence is a valid or not by clicking Checking the correctness of this statement
For a wrong sentence, you can get the correct sentence by clicking Get correct sentence



English

Form a sentence (Declarative or Interrogative or any other type) from the given words

(select the buttons in proper order)

Formed Sentence *(after selecting words)*:

I am sleeping

Right answer!!!

Which of these sentences are grammatically and logically correct?

- Ram eat an apple.
- Ram eats an apple.
- Apple eats Ram.
- Eat an apple.
- Eat Ram

Tokens and Types

Definition

Tokens correspond to the total number of word counts in a text while type corresponds to the total count of unique words in a text. We can say that language consists of various types of words and all the particular instances of these words are called tokens.

For Eg: Do not waste time as wasting time does a lot of harm.

Here, #tokens=12 #types=11 (time has been repeated twice)

Type vs token distinction

The type/token distinction is related to that between universals and particulars. Tokens are concrete particular instances of a general and abstract type. There is only one word 'the' (type) but many instances of it found on this page (token).

The type/token distinction is applicable beyond language as well. For eg:

Beethovena's Fifth Symphony and performances of it

The white elephant and specimens of it

Kentucky Fried Chicken and its centres

Types - (continued)

Study this example again: Do not waste time as wasting time does a lot of harm.

Now, we notice that 'waste' and 'wasting' share a common root. So do 'do' and 'does'. Do we consider them as different types? The second approach is to consider them as a single type as inflections(different grammatical forms) of the same word (type). Therefore,

#tokens=12 #types(root)=9

Procedure

STEP 1: Select the corpus.

STEP 2: Submit the number of tokens and types in the given text box.

STEP 3: If your answer is correct, you can enter the number of types of root and check your answer.

Implementation

Give number of types and tokens for the following sentences:

1. What did you do?
2. Two and two makes four.
3. Ram eats an apple afteat eating a banana.
4. April will come here after 10th of April.
5. John drinks tea and George takes cold drink.