

ST. JOSEPH'S COLLEGE OF ENGINEERING & TECHNOLOGY PALAI



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CSL 431 COMPILER LAB

Name :

Roll No. :

University Register No. :

Semester :

Batch :

Academic Year : **2021-2022**

ST. JOSEPH'S COLLEGE OF ENGINEERING & TECHNOLOGY PALAI

CHOONDACHERRY, BHARANANGANAM - 686579



CSL431 COMPILER LAB

RECORD BOOK

2021-2022 Academic Year

Name.....

Semester & BatchRoll No.....

Branch.....

University Examination Reg. No.....

Certificate

Certified that this is the Bonafide Record of the work done in the
..... Laboratory of St. Joseph's College of Engineering &
Technology, Palai by

Faculty in Charge

Internal Examiner

External Examiner

Palai

Date

LIST OF EXPERIMENTS

EXP. NO	DATE	NAME OF EXPERIMENT	PAGENO
1	26/09/22	Design and implement a lexical analyzer using C language to recognize all valid tokens in the input program. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments.	
2.1	10/10/22	Implement a Lexical Analyzer for a given program using Lex Tool.	
2.2	01/10/22	Write a lex program to display the number of lines, words and characters in an input text.	
2.3	17/10/22	Write a LEX Program to convert the substring abc to ABC from the given input string.	
2.4	31/10/22	Write a lex program to find out total number of vowels and consonants from the given input sting.	
3.1	07/11/22	Generate a YACC specification to recognize a valid arithmetic expression that uses operators +, −, *, / and parenthesis	
3.2	07/11/22	Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.	
3.3	07/11/22	Implementation of Calculator using LEX and YACC	
4	21/11/22	Write a program to find ϵ – closure of all states of any given NFA with ϵ transition.	
5	28/11/22	Write a program to find First and Follow of any given grammar.	
6	05/12/22	Construct a Shift Reduce Parser for a given language.	

7	12/12/22	Simulation of code optimization Techniques.	
8	12/12/22	Implement Intermediate code generation for simple expressions.	
9		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.	

DATE: 26/09/22**EXPERIMENT NO: 1****AIM:**

Design and implement a lexical analyzer using C language to recognize all valid tokens in the input program. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments.

ALGORITHM:

1. Start
2. Read the input text.
3. For each character:
 - a. if the character belongs to the operator set:
print it as an operator
 - b. if the character is an alpha-numeric character:
add it to the buffer
 - c. if the character is equal to ' ' or '\n' or '\t':
add '\0' into the buffer
 - d. if the buffer matches any of the items in keywords[] :
print the buffer as a keyword
 - e. else:
print the token as an operator
4. Stop

PROGRAM:

```
#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("inp.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 an identifier\n", buffer);  
    }  
}  
fclose(fp);  
return 0;  
}
```

OUTPUT

Enter the input file name: input.txt

if is a keyword

a is an identifier

< is an operator

b is an identifier

a is an identifier

+ is an operator

+ is an operator

RESULT

The program to design and implement a lexical analyser using C language to recognise all valid tokens in the input was successfully compiled and output was verified. Also, the course outcome CO1 was achieved.

DATE: 10/10/22**EXPERIMENT NO: 2.1****AIM:**

Write a program to implement lexical analyser for a given program using Lex Tool.

ALGORITHM:

1. Start
2. Read the input text.
3. For each yytext:
 - a. if the character is equal to 'if', 'else', 'int', 'switch' or 'char':
print the token as a keyword
 - b. if the character is in range a to z or _, followed or not followed by characters in range a to z or 0-9:
print the token as an identifier
 - c. if the character is in range 0-9 and characters which follow are also in the same range:
print the token as a number
 - d. if the character is equal to '=', '+', '*', '/', '<' or '>':
print the token as an operator
 - e. if the character is equal to '/n':
exit from the program
4. Stop

PROGRAM:

```
% {
int n=0;
int k=0, i=0, o=0, num=0;
% }

%%
if|else|while|int|switch|for|char { printf("%s is a keyword\n",yytext);k++;n++;}

_*[a-z]*[A-Z]+[0-9]*|_[a-z]+[A-Z]*[0-9]* { printf("%s is an
identifier\n",yytext);i++;n++;}

"=|'|"+|"-|"*|"/|"<|">" {printf("%s is an operator\n",yytext);o++;n++;}

[0-9]+ {printf("%s is a number\n",yytext);num++;n++;}
```



```
(.) {}
```

```
"\t"|"\\n" {exit(0); }  
%%
```

```
int yywrap()  
{  
}
```

```
int main()  
{  
printf("Enter the input text: ");  
yylex();  
return 0;  
}
```

OUTPUT

Enter the input file name: input.txt

if is a keyword

a is an identifier

< is an operator

b is an identifier

a is an identifier

+ is an operator

+ is an operator

RESULT

The program to design and implement a lexical analyser using Lex tool to recognise all valid tokens in the input was successfully compiled and output was verified. Also, the course outcome CO1 was achieved.

DATE: 17/10/22**EXPERIMENT NO: 2.2****AIM:**

Write a program in Lex to display the number of lines, words and characters in an input file.

ALGORITHM:

- a) Read the input file name.
- b) Set a pointer yyin to the input file
- c) Read the file contents.
- d) For each yytext:
 - a. if the character is in range a to z, A to Z or 0-9:
c++;
 - b. if the character is equal to ' ' or '/t':
w++;
 - c. if the character is equal to '/n':
l++
 - d. if 'End of File':
c++; w++; l++;
print c, w and l as the count of characters, words and lines respectively.
exit from the program
- e) Stop

PROGRAM:

```
% {  
int c, w, l=0;  
%}  
  
%%  
[a-z][A-Z] {c++; }  
(.) {w++;}  
"\n" {l++;}  
<<EOF>> {c++; w++; l++;  
printf("The total no. of Characters: ",c);  
printf("The total no. of Words: ",w);  
printf("The total no. of Lines: ",l);  
}  
%%  
  
int yywrap()
```

```
{  
}  
  
int main()  
{  
    char *fname[100] ;  
    printf("Enter the input file name: ");  
    scanf("%s",fname);  
    extern FILE *yyin;  
    /* yyin points to the file input.txt and opens it in read mode*/  
    yyin = fopen(fname, "r");  
    yylex();  
    return 0;  
}
```

OUTPUT:

Enter the input file name: input.txt

The total no. of Characters: 36

The total no. of Words: 7

The total no. of Lines: 2

RESULT

The program to count the number of lines, words and characters in an input file was successfully compiled and output was verified. Also, the course outcome CO1 was achieved.

DATE: 31/10/22**EXPERIMENT NO: 2.3****AIM:**

Write a program in Lex to convert abc to ABC from a string.

ALGORITHM:

1. Start
2. Read the input string.
3. Initialize s as string to store the result.
4. For each yytext:
 - i. if set of characters equals to "abc":
s= s+ "ABC";
 - ii. else:
s= s+ yytext;
5. Stop

PROGRAM:

```
% {  
char *s[100];  
% }  
  
%%  
"abc" {s=s+'ABC';}  
[A-Z][a-z][0-9] {s=s+yytext}  
(.) {}  
"\n" {  
printf("The converted text: ",s);  
}  
%%  
  
int yywrap()  
{  
}  
  
int main()  
{  
printf("Enter the String: ");
```

```
yylex();  
return 0;  
}
```

OUTPUT:

Enter the String: Pythonabc RATAabc

The converted text: PythonABC RATAABC

RESULT

The program to convert abc to ABC from a string was successfully compiled and output was verified. Also, the course outcome CO1 was achieved.

DATE: 07/11/22**EXPERIMENT NO: 2.4****AIM:**

Write a program to count the number of vowels and consonants from a string.

ALGORITHM:

1. Start
2. Read the input string.
3. For each yytext:
 - a. if the character is equal to A, E, I, O, U, a, e, i, o or u:
vow++;
 - b. else:
cons++;
4. Print vow and cons as count of the no. of vowels and consonants respectively.
5. Stop

PROGRAM:

```
% {  
int vow, cons=0;  
% }  
  
%%  
AEIOUaeiou {vow++;}  
[A-Z][a-z] {cons++;}  
(.) {}  
"\\n" {  
printf("The total no. of Vowels: ",vow);  
printf("The total no. of Consonants: ",cons)  
}  
%%  
  
int yywrap()  
{  
}  
  
int main()  
{  
printf("Enter the String: ");  
yylex();
```

```
return 0;  
}
```

OUTPUT:

Enter the String: Python Programming

The total no. of Vowels: 4

The total no. of Consonants: 13

RESULT

The program to count the number of vowels and consonants from a string was successfully compiled and output was verified. Also, the course outcome CO1 was achieved.

DATE: 07/11/22**EXPERIMENT NO: 3.1****AIM:**

Write a program using YACC to recognise a valid arithmetic expression that uses operators such as +, -, * or /.

ALGORITHM:Algorithm for tokenization (LEX)

For each yytext:

- a. if the character is in range 0-9 and characters which follow are also in the same range:
 - i. return DIGIT token
- b. if the character is in range a to z or _, followed or not followed by characters in range a to z or 0-9:
 - i. return ID token
- c. if character equals to "\n":
 - i. return NL token

Algorithm for parser (YACC)

1. Initialize DIGIT, ID and NL as tokens.
2. Set left precedence for +, - * and /.
3. Set the production rule such that the statement(stmt) must be of the form:
stmt= exp NL
4. Also, any expression(exp) in it must be of the form:
exp+exp, exp-exp, exp*exp or exp/exp, where exp can be DIGIT or ID, would be valid.
5. Accept the statement as string.
6. Parse it into yyparse() so as to compare it with the production rules.
7. If the syntax is same, print "Valid Expression".
8. Else, print "Invalid Expression".

PROGRAM:exp.y

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



```
#define YYSTYPE double
% }

%token DIGIT ID NL
%left '-' '+'
%left '*' '/'

%%
stmt : exp NL {printf("Valid expression"); exit(0);} ;
exp :exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '(' exp ')'
    | ID
    | DIGIT;
%%

int yyerror(char *msg)
{
printf("Invalid string");
}

void main ()
{
printf("Enter the exp: ");
yyparse();
}
```

exp.l

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

%%
[0-9]+ { return DIGIT;}
[a-zA-Z][a-zA-Z0-9_]* { return ID; }
\n { return NL;}
. { return yytext[0]; }
%%
```

OUTPUT:

Enter the exp: c=a+b

Valid expression

Enter the exp: a+c=a

Invalid string

RESULT

The program to recognise expression that uses operators such as +, -, * or / using YACC was successfully compiled and output was verified. Also, the course outcome CO2 was achieved.

DATE: 07/11/22	EXPERIMENT NO: 3.2
-----------------------	---------------------------

AIM:

Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.

ALGORITHM:Algorithm for tokenization (LEX)

For each yytext:

- a. if the character is in range a to z or _, followed or not followed by characters in range a to z or 0-9:
 - i. return ID token
- b. if character equals to "\n":
 - i. return NL token

Algorithm for parser (YACC)

- a. Initialize ID and NL as tokens.
- b. Set the production rule such that the statement(stmt) must be of the form:
 - a. stmt= exp NL
- c. Also, any expression(exp) in it must be of the form as mentioned in the tokenizing criteria of ID.
- d. Accept the statement as string.
- e. Parse it into yyparse() so as to compare it with the production rules.
- f. If the syntax is same, print "Valid Identifier".
- g. Else, print "Invalid Identifier".

PROGRAM:exp.y

```
% {  
#include<stdio.h>  
#define YYSTYPE double  
%}  
  
%token ID NL  
  
%%  
stmt : exp NL {printf("Valid Identifier"); exit(0);} ;
```

```
exp :ID;
%%

int yyerror(char *msg)
{
printf("Invalid Identifier ");
}

void main ()
{
printf("Enter the token: ");
yyparse();
}

exp.l
%{
#include "y.tab.h"
%}

%%
[a-zA-Z][a-zA-Z0-9_]* { return ID; }
\n { return NL;}
. { return yytext[0]; }
%%
```

OUTPUT:

Enter the token: alpha

Valid Identifier

Enter the exp: 54a

Invalid Identifier

RESULT

The program recognize a valid identifier which starts with a letter followed by any number of letters or digits using YACC was successfully compiled and output was verified. Also, the course outcome CO2 was achieved.

DATE: 07/11/22**EXPERIMENT NO: 3.3****AIM:**

Write a program to implement a calculator using LEX and YACC.

ALGORITHM:Algorithm for tokenization (LEX)

For each yytext:

- a. if the character is in range 0-9:
 - i. store the numeric value in yylval
 - ii. return DIGIT token
- b. if character equals to "\n":
 - i. return NL token

Algorithm for parser (YACC)

1. Initialize DIGIT, ID and NL as tokens.
2. Set left precedence for +, - * and /.
3. Set the production rule such that the statement(stmt) must be of the form:
stmt= exp NL
4. Also, if expression(exp) in it is of the form:
 - i. exp+'exp':
set \$\$=\$1+\$3
 - ii. exp-'exp':
\$\$=\$1-\$3
 - iii. exp'*'exp':
\$\$=\$1*\$3
 - iv. exp '/'exp':
\$\$=\$1/\$3
 - v. ID or DIGIT
\$\$=\$1
5. Accept the statement as string.
6. Parse it into yyparse() so as to analyse and calculate it with the production rules.
7. Print \$1 as the result.

PROGRAM:yac.y

% {

```
#include<stdio.h>

% }

%token NUMBER ID NL

%left '+' '-'

%left '*' '/'

%%

stmt : exp NL { printf("Value = %d\n", $1); exit(0); };
exp : exp '+' exp { $$ = $1 + $3; }
    | exp '-' exp { $$ = $1 - $3; }
    | exp '*' exp { $$ = $1 * $3; }
    | exp '/' exp { $$ = $1 / $3; }
    | '(' exp ')' { $$ = $2; }
    | ID { $$ = $1; }
    | NUMBER { $$ = $1; };
%%

int yyerror(char *msg)
{
    printf("Invalid Expression\n");
    //exit(0);
}

main ()
{
    printf("Enter the expression\n");
    yyparse();
}
```

lexc.l

```
% {

#include "y.tab.h"

extern int yylval;

% }

%%

[0-9]+ { yylval=atoi(yytext); return NUMBER; }
```

```
\n {return NL;}  
. {return yytext[0]; }  
%%
```

OUTPUT:

```
lex lexc.l  
yacc -d yac.y  
gcc lex.yy.c y.tab.c -lfl  
./a.out  
Enter the expression:  
6/3  
Value=2  
./a.out  
Enter the expression:  
5*4  
Value=20
```

RESULT

The program to recognise expression that uses operators such as +, -, * or / using LEX and YACC was successfully compiled and output was verified. Also, the course outcome CO2 was achieved.

DATE: 21/11/22**EXPERIMENT NO: 4****AIM:**

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

ALGORITHM:

1. Input the states and connections.
2. Take the starting state and set a pointer i.
3. Initialize a set called closure= { }, and add q_i to it.
4. Initialize a queue called neighbours, consisting of immediate neighbours of q_i reachable via an ϵ move.
5. Take the first element from the queue, traverse to it and set the pointer i to that state.
6. Add that state into the closure set.
7. Check for any neighbours which can be visited via an ϵ move, and if found, add them to the queue.
8. Repeat steps 5 to 7 until the queue is empty.
9. The closure set would contain the ϵ closure of the state q_i .
10. Repeat the above steps for all states.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
char result[20][20],copy[3],states[20][20];
void add_state(char a[3],int i){
    strcpy(result[i],a);
}

void display(int n){
    int k=0;
    printf("\nEpsilon closure of %s = { ",copy);
    while(k < n){
        printf(" %s",result[k]);
        k++;
    }
}
```



```
    }
    printf(" } ");
}

int main()
{
    FILE *INPUT;
    INPUT=fopen("input.dat","r");
    char state[3];
    int end,i=0,n,k=0;
    char state1[3],input[3],state2[3];
    printf("Enter the no of states: ");
    scanf("%d",&n);
    printf("Enter the states: ");
    for(k=0;k<3;k++){
        scanf("%s",states[k]);
    }

    for( k=0;k<n;k++){
        i=0;
        strcpy(state,states[k]);
        strcpy(copy,state);
        add_state(state,i++);
        while(1){
            end = fscanf(INPUT,"%s%s%s",state1,input,state2);
            if (end == EOF ){
                break;
            }

            if( strcmp(state,state1) == 0 ){
                if( strcmp(input,"e") == 0 ) {
                    add_state(state2,i++);
                    strcpy(state, state2);
                }
            }
        }
    }
}
```

```
    }  
    }  
    display(i);  
    rewind(INPUT);  
    }  
    printf("\n");  
    return 0;  
}
```

input.dat

q0 0 q0

q0 1 q1

q0 e q1

q1 e q2

OUTPUT:

Enter the no. of states: 3

Enter the states: q0 q1 q2

Epsilon closure of q0= {q0 q1 q2}

Epsilon closure of q1= {q1 q2}

Epsilon closure of q2= {q2}

RESULT

The program to implement ϵ transition of the given NFA was successfully compiled and output was verified. Also, the course outcome CO3 was achieved.

DATE: 28/11/22	EXPERIMENT NO: 5
-----------------------	-------------------------

AIM:

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

ALGORITHM:

1. If the first symbol in the R.H.S of the production is a Terminal then it can directly be included in the first set.
2. If the first symbol in the R.H.S of the production is a Non-Terminal, then call the findfirst function again on that Non-Terminal.
3. If the First of the new Non-Terminal contains an epsilon then we have to move to the next symbol of the original production which can again be a Terminal or a Non-Terminal.
4. If a Non-Terminal on the R.H.S. of any production is followed immediately by a Terminal then it can immediately be included in the Follow set of that Non-Terminal.
5. If a Non-Terminal on the R.H.S. of any production is followed immediately by a Non-Terminal, then the First Set of that new Non-Terminal gets included on the follow set of our original Non-Terminal.
6. In case encountered an epsilon i.e. " # " then, move on to the next symbol in the production.
7. If reached the end of a production while calculating follow, then the Follow set of that non-terminal will include the Follow set of the Non-Terminal on the L.H.S. of that production. This can easily be implemented by recursion.

PROGRAM:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
```

```
void followfirst(char, int, int);
```

```
void follow(char c);
```

```
void findfirst(char, int, int);
```

```
int count, n = 0;
char calc_first[10][100];
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;
    printf("Enter no. of production rules: ");
    scanf("%d",&count);
    printf("Enter the production rules\n");
    for(int k=0;k<count;k++)
    {
        scanf("%s",&production[k]);
    }
    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 already 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 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++;  
}
```

```
    }  
}
```

OUTPUT:

Enter no. of production rules: 8

Enter the production rules

E=TR

R=+TR

R=#

T=FY

Y=*FY

Y=3

F=(E)

F=i

First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { *, 3, }

First(F) = { (, i, }

Follow(E) = { \$,), }

Follow(R) = { \$,), }

Follow(T) = { +, \$,), }

Follow(Y) = { +, \$,), }

Follow(F) = { *, 3, }

RESULT

The program to find First and Follow of any given grammar was successfully compiled and output was verified. Also, the course outcome CO4 was achieved.

DATE: 28/11/22**EXPERIMENT NO: 6****AIM:**

Write a program to implement Shift-Reduce Parser for a given language.

ALGORITHM:

1. Get the input expression and store it in the input buffer.
2. Read data from the input buffer, one at a time.
3. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the stack implementation table with corresponding stack actions with input symbols.

PROGRAM:

```
#include<stdio.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()

{

    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");

    puts("enter input string ");

    scanf("%s",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%s symbols",stk,a,act);
    check();
}
}

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+2]='\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;  
        }  
}
```

OUTPUT:

GRAMMAR is $E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

enter input string

id+id

stack	input	action
\$id	+id\$	SHIFT->id
\$E	+id\$	REDUCE TO E
\$E+	id\$	SHIFT->symbols
\$E+id	\$	SHIFT->id
\$E+E	\$	REDUCE TO E
\$E	\$	ACCEPT

RESULT

The program to implement shift-reduce parser for a given language was successfully compiled and output was verified. Also, the course outcome CO4 was achieved.

DATE: 05/12/22	EXPERIMENT NO: 7
-----------------------	-------------------------

AIM:

Write a program to implement the simulation of code optimization techniques.

ALGORITHM:

1. Execute an instruction such as a print command, with and without the loop function, but to produce the same output.
2. Record the time taken on both cases.
3. Upon analysing the time_taken, it would be observed that the function block without the loop took less time for execution than the other one.
4. This is basically loop unrolling, where iterations in a source code are reduced or removed to increase the execution speed.

PROGRAM:

```
#include<stdio.h>#
#include <time.h>
void fun1()
{
for (int i=0;i<5; i++)
    printf("Hello\n");
}

void fun2()
{
printf("Hello\n");
printf("Hello\n");
printf("Hello\n");
printf("Hello\n");
printf("Hello\n");
}

void main()
{
clock_t t1, t2;

t1= clock();
fun1();
t1= clock()-t1;
double time_taken1 = ((double)t1)/CLOCKS_PER_SEC;
```

```
printf("/normal loop/\n");
printf("Time taken: %f seconds\n\n",time_taken1);

t2=clock();
fun2();
t2= clock() -t2;
double time_taken2= ((double)t2)/CLOCKS_PER_SEC;
printf("/unrolled loop/\n");
printf("Time taken: %f seconds\n",time_taken2);
}
```

OUTPUT:

Hello

Hello

Hello

Hello

Hello

/normal loop/

Time taken: 0.000818 seconds

Hello

Hello

Hello

Hello

Hello

/unrolled loop/

Time taken: 0.000036 seconds

RESULT

The program to implement code optimization using loop unrolling was successfully compiled and output was verified. Also, the course outcome CO4 was achieved.

DATE: 12/12/22	EXPERIMENT NO: 8
-----------------------	-------------------------

AIM:

Implement Intermediate code generation for simple expressions.

ALGORITHM:

1. Start
2. Get the input expression and store it in the input buffer.
3. Identify the various operators used in the expression.
4. Generate separate expressions for each operator, starting in the order of Brackets, Division, Multiplication, Addition and Subtraction.
5. Stop

PROGRAM:

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

int i=1,j=0,no=0,tmpch=90;

char str[100],left[15],right[15];

void findopr();

void explore();

void fleft(int);

void fright(int);

struct exp

{

int pos;

char op;

}k[15];

void main()

{
```

```
printf("\t\tINTERMEDIATE CODE GENERATION\n\n");

printf("Enter the Expression :");

scanf("%s",str);

printf("The intermediate code:\n");

findopr();

explore();

}

void findopr()

{

for(i=0;str[i]!='\0';i++)

if(str[i]==':')

{

k[j].pos=i;

k[j++].op=':';

}

for(i=0;str[i]!='\0';i++)

if(str[i]=='/')

{

k[j].pos=i;

k[j++].op='/';

}

for(i=0;str[i]!='\0';i++)

if(str[i]=='*')

{

k[j].pos=i;

k[j++].op='*';

}
```

```
for(i=0;str[i]!='\0';i++)
if(str[i]=='+')
{
k[j].pos=i;
k[j++].op='+';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='-')
{
k[j].pos=i;
k[j++].op='-';
}
}
void explore()
{
i=1;
while(k[i].op!='\0')
{
fleft(k[i].pos);
fright(k[i].pos);
str[k[i].pos]=tmpch--;
printf("\t%c := %s%c%s\t",str[k[i].pos],left,k[i].op,right);
printf("\n");
i++;
}
fright(-1);
if(no==0)
```

```
{
fleft(strlen(str));
printf("\t%s := %s",right,left);
exit(0);
}

printf("\t%s := %c",right,str[k[--i].pos]);
}

void fleft(int x)
{
int w=0,flag=0;

x--;

while(x!= -1 &&str[x]!='+' &&str[x]!='*' &&str[x]!='=' &&str[x]!='\0' &&str[x]!='-'
'&&str[x]!='/' &&str[x]!=':')
{
if(str[x]!='$' && flag==0)
{
left[w++]=str[x];
left[w]='\0';
str[x]='$';
flag=1;
}
x--;
}
}

void fright(int x)
{
int w=0,flag=0;
```



```
x++;

while(x!= -1 && str[x]!='+&&str[x]!='*&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[x]!='/')

{

if(str[x]!='$'&& flag==0)

{

right[w++]=str[x];

right[w]='\0';

str[x]='$';

flag=1;

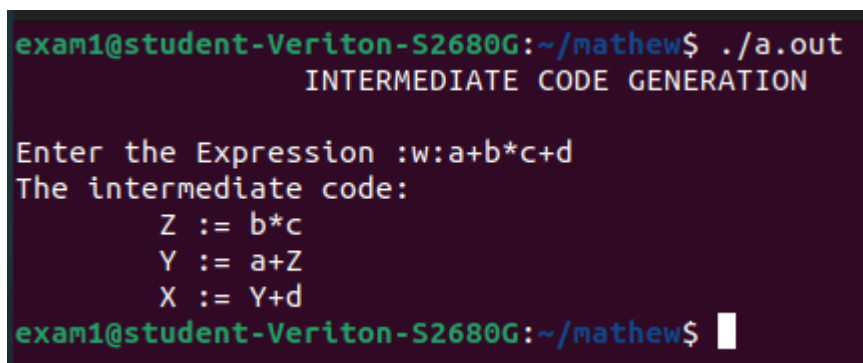
}

x++;

}

}
```

OUTPUT:



```
exam1@student-Veriton-S2680G:~/mathew$ ./a.out
INTERMEDIATE CODE GENERATION

Enter the Expression :w:a+b*c+d
The intermediate code:
    Z := b*c
    Y := a+Z
    X := Y+d
exam1@student-Veriton-S2680G:~/mathew$
```

RESULT

The program to implement intermediate code generation was successfully compiled and output was verified. Also, the course outcome CO6 was achieved.

DATE: 12/12/22**EXPERIMENT NO: 12****AIM:**

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.

ALGORITHM:

1. Start the program.
2. Get the three variables from statements and stored in the text file k.txt.
3. Compile the program and give the path of the source file.
4. Execute the program.
5. Target code for the given statement was produced.
6. Stop the program

PROGRAM:

```
#include <stdio.h>

#include <curses.h>

#include <string.h>

char op[2],arg1[5],arg2[5],result[5];

void main()

{

    FILE *fp1,*fp2;

    char f[100];

    printf("Enter input file name: ");

    scanf("%s",f);

    fp1=fopen(f,"r");

    fp2=fopen("output.txt","w");

    while(!feof(fp1))

    {

        fscanf(fp1,"%s%s%s",op,arg1,arg2,result);
```

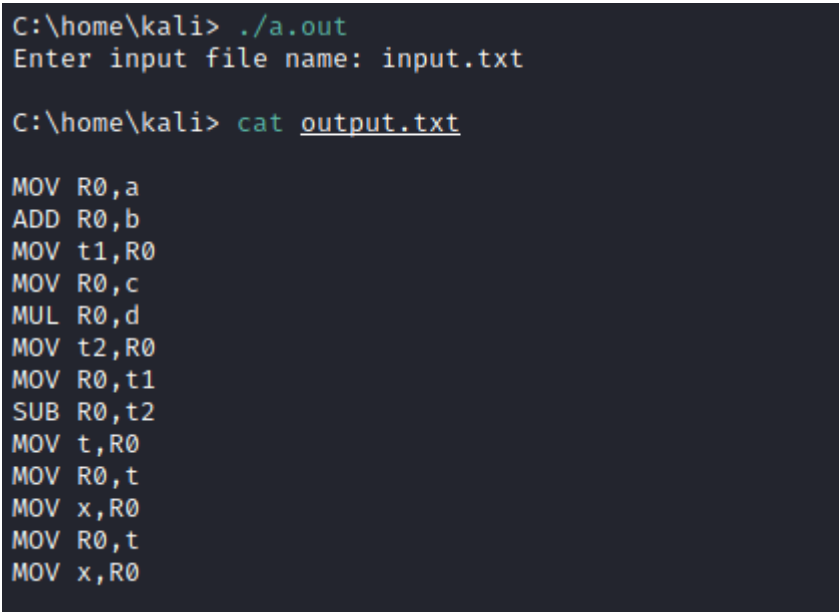
```
if(strcmp(op,"+")==0)
{
    fprintf(fp2,"\nMOV R0,%s",arg1);
    fprintf(fp2,"\nADD R0,%s",arg2);
    fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"*")==0)
{
    fprintf(fp2,"\nMOV R0,%s",arg1);
    fprintf(fp2,"\nMUL R0,%s",arg2);
    fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"-")==0)
{
    fprintf(fp2,"\nMOV R0,%s",arg1);
    fprintf(fp2,"\nSUB R0,%s",arg2);
    fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"/")==0)
{
    fprintf(fp2,"\nMOV R0,%s",arg1);
    fprintf(fp2,"\nDIV R0,%s",arg2);
    fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"")==0)
{
    fprintf(fp2,"\nMOV R0,%s",arg1);
    fprintf(fp2,"\nMOV %s,R0",result);
}
```

```
}  
    fclose(fp1);  
    fclose(fp2);  
    getchar();  
}
```

input.txt

```
+ a b t1  
* c d t2  
- t1 t2 t  
= t ? x
```

OUTPUT:



```
C:\home\kali> ./a.out  
Enter input file name: input.txt  
  
C:\home\kali> cat output.txt  
  
MOV R0,a  
ADD R0,b  
MOV t1,R0  
MOV R0,c  
MUL R0,d  
MOV t2,R0  
MOV R0,t1  
SUB R0,t2  
MOV t,R0  
MOV R0,t  
MOV x,R0  
MOV R0,t  
MOV x,R0
```

RESULT

The program to implement the backend of a compiler was successfully compiled and output was verified. Also, the course outcome CO5 was achieved.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

To evolve as a school of computing with globally reputed centres of excellence and serve the changing needs of the industry and society.

Mission

- The department is committed in bringing out career-oriented graduates who are industry ready through innovative practices of teaching and learning process
- To nurture professional approach, leadership qualities and moral values to the graduates by organizing various programs periodically
- To acquire self-sustainability and serve the society through research and consultancy

Programme Educational Objectives (PEOs)

PEO1: The graduates will have a successful career in industries associated with Computer Science and Engineering or as entrepreneurs.

PEO2: To ensure that graduates will have the ability and attitude to adapt to emerging technological changes.

PEO3: To enable graduates to pursue higher education and research.