

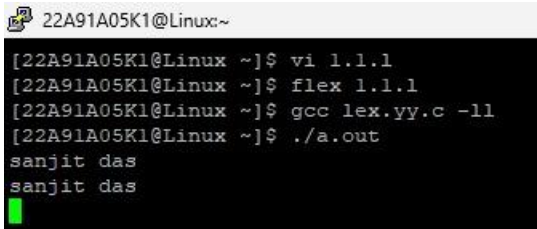
EXPERIMENT -1

1.1) Write a lex program whose output is same as input.

PROGRAM:

```
%%  
. ECHO;  
%%  
int yywrap(void)  
{ return 1;  
}  
int main(void)  
{ yylex();  
  return 0;  
}
```

OUTPUT:



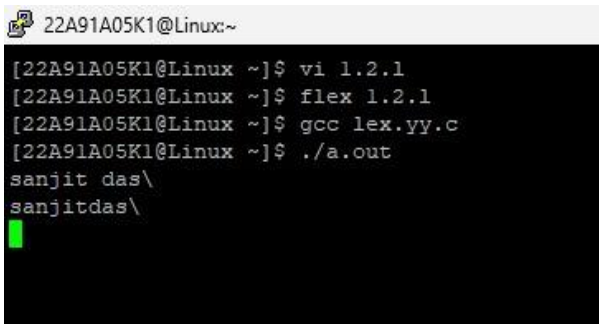
```
22A91A05K1@Linux:~  
[22A91A05K1@Linux ~]$ vi 1.1.1  
[22A91A05K1@Linux ~]$ flex 1.1.1  
[22A91A05K1@Linux ~]$ gcc lex.yy.c -ll  
[22A91A05K1@Linux ~]$ ./a.out  
sanjit das  
sanjit das
```

1.2) Write a lex program which removes white spaces from its input file.

PROGRAM:

```
%%  
[ ] {};  
. ECHO;  
%%  
int yywrap(void)  
{ return 1;  
}  
int main(void)  
{ yylex();  
  return 0;  
}
```

OUTPUT:



```
22A91A05K1@Linux:~  
[22A91A05K1@Linux ~]$ vi 1.2.1  
[22A91A05K1@Linux ~]$ flex 1.2.1  
[22A91A05K1@Linux ~]$ gcc lex.yy.c  
[22A91A05K1@Linux ~]$ ./a.out  
sanjit das\  
sanjitdas\  
█
```

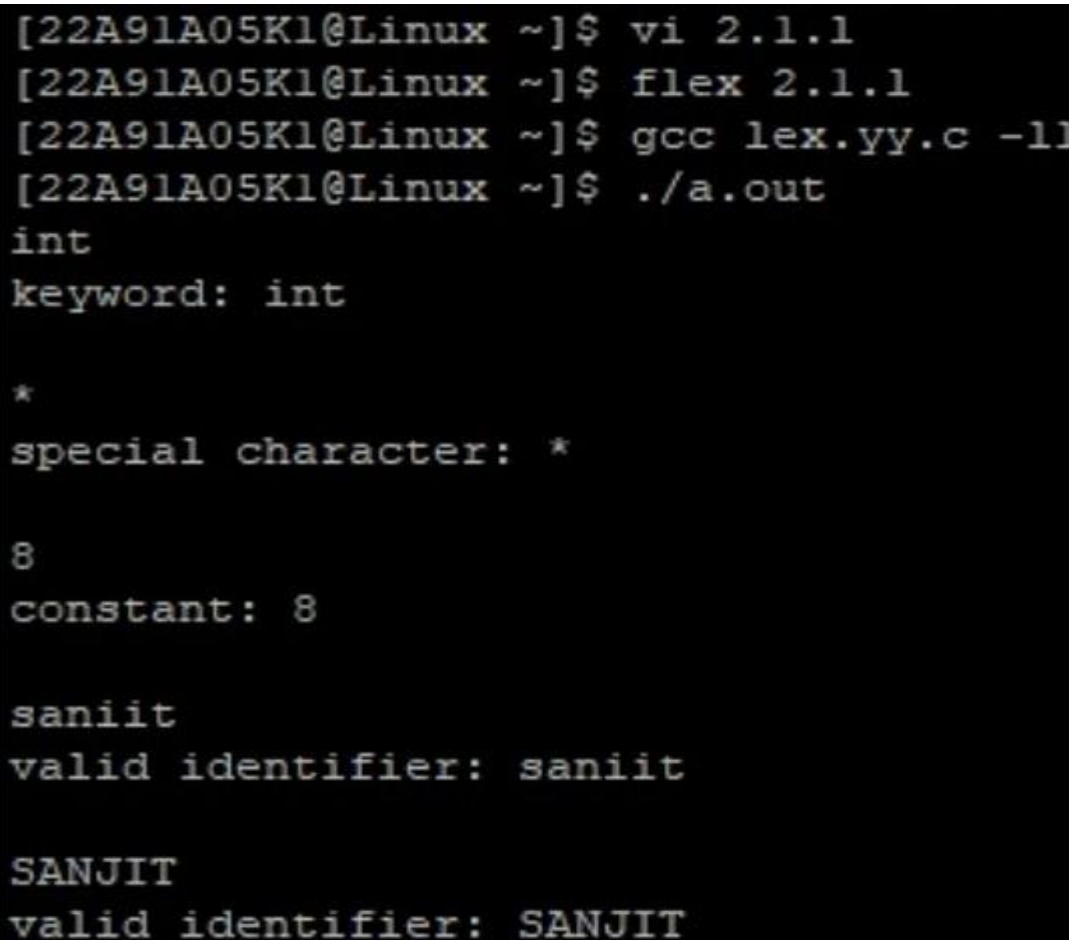
EXPERIMENT -2

2.1) Write a lex program to identify the patterns in the input file.

PROGRAM:

```
%{  
#include<stdio.h>  
%}  
%%  
["int""char""for""if""while""then""return""do"] {printf("keyword : %s\n");}  
[*%+\\-] {printf("Operator : %s ", yytext);}  
[(){};] {printf("Special Character: %s\n", yytext);}  
[0-9]+ {printf("Constant : %s\n", yytext);}  
[a-zA-Z_][a-zA-Z0-9_]* {printf("Valid Identifier is : %s\n", yytext);}  
^[^a-zA-Z_] {printf("Invalid Identifier\n");}  
%%
```

OUTPUT:



```
[22A91A05K1@Linux ~]$ vi 2.1.1  
[22A91A05K1@Linux ~]$ flex 2.1.1  
[22A91A05K1@Linux ~]$ gcc lex.yy.c -ll  
[22A91A05K1@Linux ~]$ ./a.out  
int  
keyword: int  
  
*  
special character: *  
  
8  
constant: 8  
  
saniit  
valid identifier: saniit  
  
SANJIT  
valid identifier: SANJIT
```

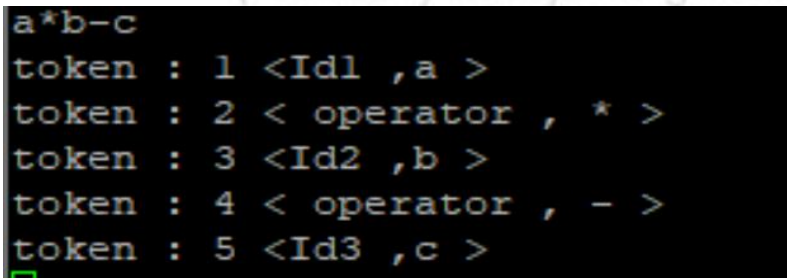
2.2) Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

PROGRAM:

```
%{
#include<stdio.h>
int i=0,id=0;
}%

%%
[#].*[<].*[>]\n {}
[ \t\n]+ {}
\\.*\n {}
\\*(.*\n)*.*\n {}
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|v
olatile|while {printf("token : %d < keyword , %s >\n",++i,yytext);}
[+|-|*|/%<>] { printf("token : %d < operator , %s >\n",++i,yytext);} [(0;{}]
{ printf("token : %d < special char , %s >\n",++i,yytext);}
[0-9]+ { printf("token : %d < constant , %s >\n",++i,yytext);}
[a-zA-Z_][a-zA-Z0-9_]* { printf("token : %d <Id%d ,%s >\n",++i,++id,yytext);}
^[^a-zA-Z_] { printf("ERROR Invaild token %s \n",yytext);}
%%
```

OUTPUT:



```
a*b-c
token : 1 <Id1 ,a >
token : 2 < operator , * >
token : 3 <Id2 ,b >
token : 4 < operator , - >
token : 5 <Id3 ,c >
```

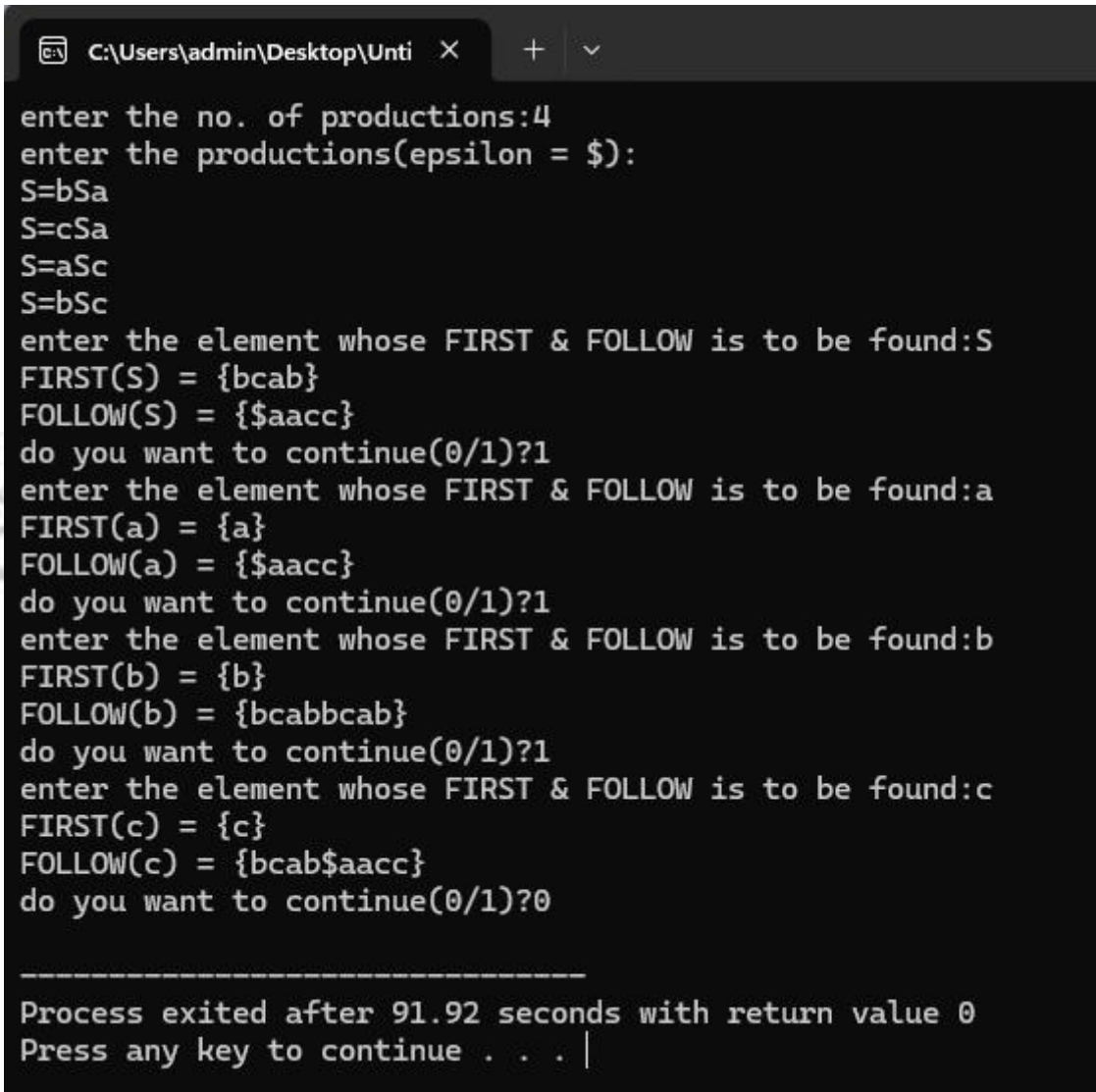
Week-3**First and Follow****3.1) AIM :** To implement First and Follow of a Grammar.**PROGRAM :**

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int n, m = 0, p, i = 0, j = 0;
char a[10][10], f[10];
void follow(char c){ if(a[0][0]
== c)
f[m++] = '$';
for(i = 0; i < n; i++){
for(j = 2; j < strlen(a[i]); j++){
if(a[i][j] == c){
if(a[i][j+1] != '\0') first(a[i][j+1]);
if(a[i][j+1] == '\0' && c != a[i][0]) follow(a[i][0]);
}
}
}
}
void first(char
c){ int k;
if(!isupper(c)) f[m++] = c;
for(k =
0; k < n; k++){ if(a[k][0] ==
c){
if(a[k][2] == '$') follow(a[i][0]);
else if(islower(a[k][2])) f[m++] = a[k][2];
else first(a[k][2]);
}
}
}
int
main(){ int
i, z; char c,
ch;
printf("enter the no. of productions:");
scanf("%d", &n);
printf("enter the productions(epsilon = $):\n");
for(i = 0; i < n; i++) scanf("%s%c", a[i], &ch);
do{
m = 0;
printf("enter the element whose FIRST & FOLLOW is to be found:");
scanf("%c", &c);
first(c);
printf("FIRST(%c) = {", c);
for(i = 0; i < m; i++) printf("%c", f[i]);
printf("}\n");
follow(c);

```

```
printf("FOLLOW(%c) = {", c);  
for(;i<m;i++) printf("%c", f[i]);  
printf("}\n");  
printf("do you want to continue(0/1)?");  
scanf("%d%c", &z, &ch);  
}  
while(z == 1);  
}
```

OUTPUT:

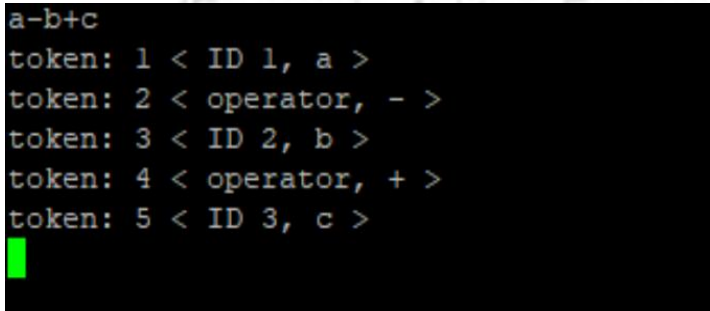
```
C:\Users\admin\Desktop\Unti X + v  
enter the no. of productions:4  
enter the productions(epsilon = $):  
S=bSa  
S=cSa  
S=aSc  
S=bSc  
enter the element whose FIRST & FOLLOW is to be found:S  
FIRST(S) = {bcab}  
FOLLOW(S) = {$aacc}  
do you want to continue(0/1)?1  
enter the element whose FIRST & FOLLOW is to be found:a  
FIRST(a) = {a}  
FOLLOW(a) = {$aacc}  
do you want to continue(0/1)?1  
enter the element whose FIRST & FOLLOW is to be found:b  
FIRST(b) = {b}  
FOLLOW(b) = {bcabbcab}  
do you want to continue(0/1)?1  
enter the element whose FIRST & FOLLOW is to be found:c  
FIRST(c) = {c}  
FOLLOW(c) = {bcab$aacc}  
do you want to continue(0/1)?0  
  
-----  
Process exited after 91.92 seconds with return value 0  
Press any key to continue . . . |
```

3.2) AIM : To implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

PROGRAM :

```
%{
#include<stdio.h>
> int i=0,id=0;
%}
%%
[#].*[*].*[>]\n
{}
[\t\n]+ {}
\\.*\n {}
\\*(.*\n)*.*\n {}
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|w
hi le {printf("token: %d < keyword, %s >\n",++i,yytext);}
[+|-|*|/|<|>] {printf("token: %d < operator, %s
>\n",++i,yytext);} [();{}] {printf("token: %d < special char,
%s >\n",++i,yytext);} [0-9]+ {printf("token: %d < constant,
%s >\n",++i,yytext);}
[a-zA-Z_][a-zA-Z0-9_]* {printf("token: %d < ID %d, %s >\n",++i,++id,yytext);}
^[^a-zA-Z_] {printf("ERROR INVALID TOKEN %s\n",yytext);}
%%
```

OUTPUT :



```
a-b+c
token: 1 < ID 1, a >
token: 2 < operator, - >
token: 3 < ID 2, b >
token: 4 < operator, + >
token: 5 < ID 3, c >
```

Experiment-4

Top-Down Parsing

4.1) Develop an operator precedence parser for a given language

PROGRAM:

```
#include<stdio.h>
#include<string.h>
char stack[20],temp;
int top=-1;
void push(char item){
    if(top>=20){
        printf("STACK OVERFLOW");
        return;
    }
    stack[++top]=item;
}
char pop(){
    if(top<=-1){
        printf("STACK UNDERFLOW");
        return;
    }
    char c;
    c=stack[top--];
    printf("Popped element:%c\n",c);
    return c;
}
char TOS(){
    return stack[top];
}
int convert(char item){
    switch(item){
        case 'i':return 0;
        case '+':return 1;
        case '*':return 2;
        case '$':return 3;
    }
}
int main(){
    char pt[4][4]={
        {'-','>','>','>'},
        {'<','>','<','>'},
        {'<','>','>','>'},
        {'<','<','<','1'}};
    char input[20];
    int lkh=0;
    printf("Enter input with $ at the end\n");
    scanf("%s",input);
    push('$');
    while(lkh<=strlen(input))
    {
```



```
if(TOS()=='$' && input[lkh]=='$'){
    printf("SUCCESS\n");
    return 1;
}
else if(pt[convert(TOS())][convert(input[lkh])]=='<'){
    push(input[lkh]);
    printf("Push---%c\n",input[lkh]);
    lkh++;
}
else    pop();
}
return 0; }
```

OUTPUT:

```
[22A91A05K1@Linux ~]$ vi 4.lcd.c
[22A91A05K1@Linux ~]$ cc -o 4.lcd 4.lcd.c
[22A91A05K1@Linux ~]$ ./4.lcd
Enter input with $ at the end
i+i+i*i$
Push---i
Popped element:i
Push---+
Push---i
Popped element:i
Popped element:+
Push---+
Push---i
Popped element:i
Push---*
Push---i
Popped element:i
Popped element:*
Popped element:+
SUCCESS
[22A91A05K1@Linux ~]$ █
```

4.2) Construct a recursive descent parser for an expression

PROGRAMS

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void Tp();
void Ep();
void E();
void T();
void check();
int count,flag;
char expr[10];
int main(){
    count=0;
    flag=0;
    printf("\nEnter an Algebraic Expression:\t");
    scanf("%s",expr);
    E();
    if((strlen(expr)==count)&&(flag==0))
        printf("\nThe expression %s is valid\n",expr);
    else
        printf("\nThe expression %s is invalid\n",expr);
    return 0;
}
void E(){
    T();
    Ep();
}
void T(){
    check();
    Tp();
}
void Tp(){
    if(expr[count]=='*'){
        count++;
        check();
        Tp();
    }
}
void check(){
    if(isalnum(expr[count]))
        count++;
    else if(expr[count]=='('){
        count++;
        E();
        if(expr[count]==')')    count++;
    }
    else
        flag=1;
}
```

```
else
flag=1;
}
void Ep(){
    if(expr[count]=='+'){
        count++;
        T();
        Ep();
    }
}
```

OUTPUT:

 22A91A05K1@Linux:~

```
[22A91A05K1@Linux ~]$ vi 4.2cd.c
[22A91A05K1@Linux ~]$ ./4.2cd
```

Enter an Algebraic Expression: ((8+6)*5)+6

The expression ((8+6)*5)+6 is valid

```
[22A91A05K1@Linux ~]$
```

 22A91A05K1@Linux:~

```
[22A91A05K1@Linux ~]$ vi 4.2cd.c
[22A91A05K1@Linux ~]$ cc -o 4.2cd 4.2cd.c
[22A91A05K1@Linux ~]$ ./4.2cd
```

Enter an Algebraic Expression: ((9*5)+6)-5

The expression ((9*5)+6)-5 is invalid

```
[22A91A05K1@Linux ~]$
```

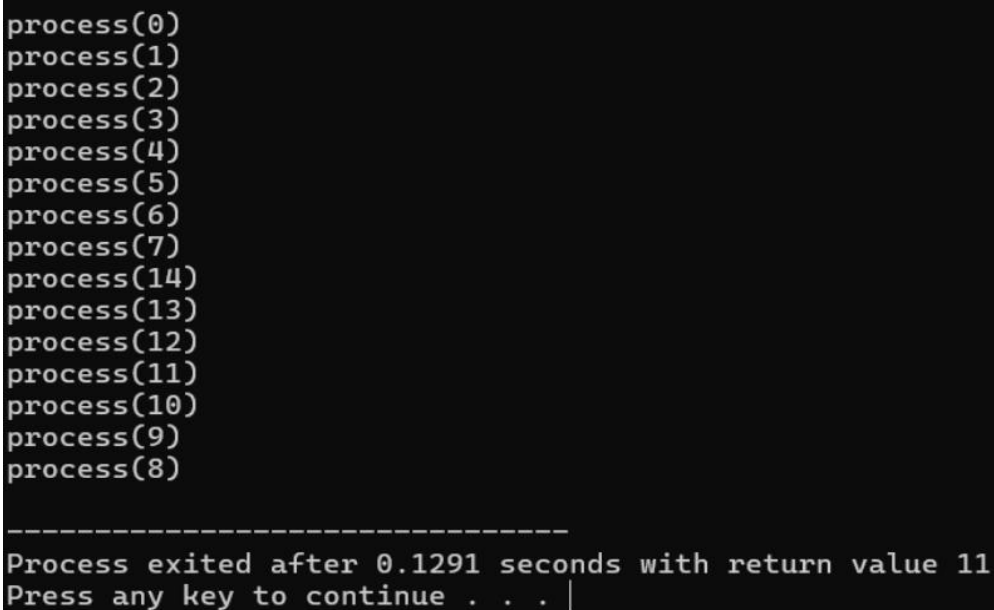
EXPERIMENT-6

6.1) Write a program to perform loop unrolling.

PROGRAM:

```
#include<stdio.h>
#define TOGETHER (8)
int main(void){
    int i = 0,entries = 15,repeat,left = 0;
    repeat = (entries / TOGETHER);
    left = (entries % TOGETHER);
    while (repeat--){
        printf("process(%d)\n", i);
        printf("process(%d)\n", i + 1);
        printf("process(%d)\n", i + 2);
        printf("process(%d)\n", i + 3);
        printf("process(%d)\n", i + 4);
        printf("process(%d)\n", i + 5);
        printf("process(%d)\n", i + 6);
        printf("process(%d)\n", i + 7);
        i += TOGETHER;
    }
    switch (left){
        case 7 : printf("process(%d)\n", i + 6);
        case 6 : printf("process(%d)\n", i + 5);
        case 5 : printf("process(%d)\n", i + 4);
        case 4 : printf("process(%d)\n", i + 3);
        case 3 : printf("process(%d)\n", i + 2);
        case 2 : printf("process(%d)\n", i + 1);
        case 1 : printf("process(%d)\n", i);
        case 0 : ;
    }
}
```

OUTPUT:



```
process(0)
process(1)
process(2)
process(3)
process(4)
process(5)
process(6)
process(7)
process(14)
process(13)
process(12)
process(11)
process(10)
process(9)
process(8)

-----
Process exited after 0.1291 seconds with return value 11
Press any key to continue . . . |
```

6.2) Write a program for constant propagation.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
void input();
void output();
void change(int p,char *res);
void constant();
struct expr{
    char op[2],op1[5],op2[5],res[5];
    int flag;
}arr[10];
int n;
void main(){
    input();
    constant();
    output();
}
void input(){
    int i;
    printf("\n\nEnter the maximum number of expressions : ");
    scanf("%d",&n);
    printf("\nEnter the input : \n");
    for(i=0;i<n;i++){
        scanf("%s",arr[i].op);
        scanf("%s",arr[i].op1);
        scanf("%s",arr[i].op2);
        scanf("%s",arr[i].res);
        arr[i].flag=0;
    }
}
void constant(){
    int i;
    int op1,op2,res;
    char op,res1[5];
    for(i=0;i<n;i++){
        if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"")==0){
            op1=atoi(arr[i].op1);
            op2=atoi(arr[i].op2);
            op=arr[i].op[0];
            switch(op){
                case '+':res=op1+op2;
                    break;
                case '-':res=op1-op2;
                    break;
                case '*':res=op1*op2;
                    break;
                case '/':res=op1/op2;
                    break;
                case '=':res=op1;
                    break;
            }
            sprintf(res1,"%d",res);
            arr[i].flag=1;
        }
    }
}
```

```
        change(i,res1);
    }
}
void output(){
    int i=0;
    printf("\nOptimized code is : ");
    for(i=0;i<n;i++){
        if(!arr[i].flag)
            printf("\n%s %s %s %s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
    }
}
void change(int p,char *res){
    int i;
    for(i=p+1;i<n;i++){
        if(strcmp(arr[p].res,arr[i].op1)==0)
            strcpy(arr[i].op1,res);
        else if(strcmp(arr[p].res,arr[i].op2)==0)
            strcpy(arr[i].op2,res);
    }
}
```

OUTPUT:

Enter the maximum number of expressions : 4

Enter the input :

= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3

Optimized code is :

+ 3 b t1
+ 3 c t2
+ t1 t2 t3