PROGRAM-1

INTRODUCTION TO FINITE AUTOMATA

Finite Automata:

A finite automata is a simple idealized machine used to recognize patterns within input taken from some character set C. The job of FA is to accept or reject an input depending on whether the pattern defined by the FA occurs in the input. An FA can be deterministic or non-deterministic

* Non-~~Deterministic~~ Deterministic finite Automata

Its a mathematical model that consist of a set of states a set of input symbol Σ, a transition function move that maps state symbol pairs to set of states, a state so that is distinguished as start state, a states of F(set) distinguished as accepting state.

An NFA can be represented dig by labelled directed graph called transition graph, in which the nodes are states and the labeled edges represent the transition function. Thus graph look like transition dig, but same character can be labelled two ~~or more~~ transition out of 1 state and edge can be labeled by special symbol as well as by input symbol. The easiest implementation of NFA is a transition table in which there is a row for each state & a column for each input symbol and ε, if necessary. The transition table representation has the advantage that it provides fast access to the transaction of a given state on a given character. Its disadvantage is that it can take up a lot of space when the input alphabet is large and most transaction are to empty set.

* Deterministic finite automata

A DFA is a special case of an NFA in which
→ no state has ε transition ie, a transition on input ε and for each state s and input symbol a, there is atmost one edge labeled a leaving s.

A DFA has atmost one transition from each state on any input. If we are using a transition table to represent the transition function of a DFA, then each entry in the transition table is single state.

Algorithm for simulating a DFA

Input: An input string x terminated by an end of file character eof. A DFA D with start state s_0 and set of accepting state F

Output: The answer 'yes' if D accept x, 'no' otherwise

Method: Apply the algorithm to input string x. The function move (s, c) give the state to which there is a transition from state s on input character c. The function nextchar returns the next character of the input string x.

```
s := s0
c := nextchar
while c ≠ eof do
    s := move (s,c);
    c := nextchar
end
if s is in F then
    return 'yes'
else return 'no'
```

Comparison of NFA & DFA

DFA	NFA
→ for every symbol of alphabet there is only one state transition in DFA.	→ there can be more than one transition for any symbol or no transition at all from a state
→ can't use empty string	→ can use empty string transaction
→ can be understood as one machine	→ can understood as multiple machine computing at same time.
→ DFA will reject the string if it end at other accepting state	→ if all branch of NFA lies on reject string we can say that NFA reject string

SIMULATION OF DFA

```
#include<stdio.h>
void main(){
    int i,k,j,in,fin,a,b,st,len,flag,sub;
    int ar[5][5],str[5];
    printf("Enter the number of states : ");
    scanf("%d",&a);
    printf("Enter the initial states : ");
    scanf("%d",&in);
    printf("Enter the final states : ");
    scanf("%d",&fin);
    printf("Enter the number of inputs : ");
    scanf("%d",&b);
    printf("Enter the transition states :\n");
    for(i=in;i<a;i++){
        for(j=0;j<b;j++) {
            printf("\tEnter the transition from %d with %d : ",i,j);
            scanf("%d",&ar[i][j]); }}
    printf("Enter the length of the string : ");
    scanf("%d",&len);
    printf("enter the string : ");
    for(k=0;k<len;k++){ scanf("%d",&str[k]); }
    flag=in;
    for(i=0;i<len;i++){
        sub=ar[flag][str[i]];
        flag=sub; }
    if(flag==fin) { printf("Accepted\n");}
    else{ printf("Rejected\n"); }}
```


PROGRAM-1

SIMULATION OF DFA

AIM:

Write a C program to implement DFA simulation.

Output

Enter the no of states : 4
Enter the initial state : 0
Enter the final state : 0
Enter the no of input : 2
Enter the transition from 0 with 0 : 3
Enter the transition from 0 with 1 : 1
enter the transition from 1 with 0 : 2
enter the transition from 1 with 1 : 0
enter the transition from 2 with 0 : 1
enter the transition from 2 with : 3
Enter the transition from 3 with : 0
enter the transition from 3 with : 2
enter length of string : 4
enter string : 1010

Accepted.

RESULT

The DFA was successfully implemented and verified.

NFA TO DFA

```
#include<stdio.h>
#include<string.h>
#include<math.h>
int ninputs , dfa[100][2][100]={0};
int state[10000]={0};
char ch[10],str[10000];
int go[10000][2]={0}, arr[10000]={0};
int main(){
    int st,fin,in , f[10], p,q,r,rel;
    int i,j=3,s=0,final=0,flag=0,curr1,curr2,k,l,c;
    printf("\nFollowing the one based indexing\n");
    printf("\nEnter the number of states :: ");
    scanf("%d",&st);
    printf("\nGiven state numbers from 0 to %d",st-1);
    for(i=0;i<st;i++) {
        state[(int)(pow(2,i))]=1; }
    printf("\nEnter number of final states : ");
    scanf("%d",&fin);
    printf("\nEnter final states :: ");
    for(i=0;i<fin;i++){
        scanf("%d",&f[i]); }
    printf("\nEnter the number of rules according to NFA :: ");
    scanf("%d",&rel);
    printf("\n\nDefine transition rule as \" initial state input symbol final ...\"\n\n");
    for(i=0;i<rel;i++){
        scanf("%d%d%d",&p,&q,&r);
        if(q==0)  {
            dfa[p][0][r]=1;}
    else{  dfa[p][1][r]=1;}}
```


PROGRAM-2

CONVERT NFA to DFA

AIM:

Write a C program to convert NFA to DFA

```
printf("\nEnter initial state :: ");
scanf("%d",&in);
in=pow(2,in); i=0;
printf("\nsolving according to DFA");
int x=0;
for(i=0;i<st;i++)  {
    for(j=0;j<2;j++){
        int stf=2;
        for(k=0;k<st;k++)  {
            if(dfa[i][j][k]==1)
                stf=stf+pow(2,k) }
        go[(int)(pow(2,i))][j]=stf;
        printf("%d-%d→%d\n",(int)(pow(2,i)),j,stf);
        if(state[stf]==0)
            arr[x++]=stf;
        state[stf]=1  }}
for(i=0;i<x;i++)  {
    printf("for %d ----",arr[x]);
    for(j=0;j<2;j++){  int new=0;
        for(k=0;k<st;k++){
            if(arr[i]&(1<<k) {
                int h=pow(2,k);
                if(new==0)
                    new=new | (go[h][j]); }}
            if(state[new]==0){
                arr[x++]=new;
                state[new]=1;}}}
    printf("\nThe total number of distinct states are ::\n");
    printf("STATE 0 1\n");
    for(i=0;i<10000;i++){
        if(state[i]==1)  {
```

```
        //printf("%d***",i); int y=0;
        if(i==0)
            printf("q0");
        else// dout {}
            for(j=0;j<st;j++){
                int x=1<<j;
                if(x&i){
                 printf("q%d",j);
                 y=y+pow(2,j);
                 //printf("y=%d",y);}}
            //printf("%d",y);
            printf("%d %d",go[y][0],go[y][1]);
            printf("\n");}}   j=3;
while(j--){
    printf("\nenter string : ");
    scanf("%s",str);
    l=strlen(str); curr1=in;   flag=0;
    printf("\nstring takes the following path -> \n");
    printf("%d-",curr1);
    for(i=0;i<l;i++)   {
        curr1=go[curr1][str[i]='0'];
        printf("%d-",curr1);  }
    printf("\nfinal state - %d\n",curr1);
    for(i=0;i<fin;i++){
        if(curr1&(1<<f[i])){
            flag=1; break;}
    if(flag)
        printf("\nstring accepted\n\n");
    else
        printf("\nstring rejected\n\n");
}return 0}
```

output :

Following the one based indexing
enter number of states: 3
given state number from 0 to 2
enter the no of final state : 1
enter the final state : 2
enter the number of rules according to NFA is 4
enter transition rule as 'initialstate input symbol final..'

1 0 1
1 1 1
1 0 2
2 0 4 enter initial state : 1

According to DFA

0 → 2
1-1 → 2
2-0 → 8
2-1 → 4
4-0 → 2
4-1 → 2
for 0- -- for 0 . . .

The total no of distinct state are :

State	0	1
q_0	0	0
q_1	2	2
q_2	8	4
q_3	2	2

enter string : abba
string take following path
2 -0-0-0-0-
finalstate -0 string rejected.

RESULT
The program to convert NFA to DFA is successfully implemented & verified.

NFA WITH E TO NFA WITHOUT E

```
#include<stdio.h>
#include<stdlib.h>
struct node{  int st;
      struct node *link;};
void findclosure(int,int);
void insert_trantbl(int,char,int);
int findalpha(char);
void findfinalstate();
void unionclosure(int);
void print_e_closure(int);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],r,buffer[20];
char alphabet[20],c;
static int e_closure[20][20]={0};
struct node* transition[20][20]={NULL};
void main(){
      int i,j,k,m,t,n;
      struct node* temp;
      printf("enter the number of alphabets?\n");
      scanf("%d",&noalpha);
      getchar();
      printf("note:-[use letter as epsilion]\n");
      printf("note:-[e must be last character, if it is present]\n");
      printf("\nEnter alphabets?\n");
      for(i=0;i<noalpha;i++)  {
            alphabet[i]=getchar();
            getchar();}
      printf("Enter the number of states?\n");
      scanf("%d",&nostate);
      printf("Enter the start state?\n");
```


PROGRAM - 3

CONVERT NFA with ε to NFA without ε

AIM:

To write a C program to convert NFA with ε to NFA without ε.

ALGORITHM:

1. Start
2. check if there is atleast one path from intial state to one of the many possible accepting state which can be traversed only by null transition if so add the intial state to the set of accepting state
3. Now choose two state in ε-NFA having NULL transition between them.
4. Remove the null transition
5. Any incoming transition to P is to be added as incoming transition to q
6. Repeat step 3, 4, 5 for all null transition to the ENFA
7. Stop.

```
    scanf("%d",&start);
    printf("Enter the number of final states?\n");
    scanf("%d",&nofinal);
    printf("Enter the final states?\n");
    for(i=0;i<nofinal;i++){
        scanf("%d",&finalstate[i]);  }
    printf("Enter no. Of transition?\n");
    scanf("%d",&notransition);
    printf("note:-[transition is in the form → qno alphabet ....]\n");
    printf("note:-[state number must be greater than zero]\n");
    printf("Enter transition?\n");
    for(i=0;i<notransition;i++) {
        scanf("%d%c%d",&r,&c,&s);// c initialize as int in the given pgm
        insert_trantbl(r,c,s)   }
  printf("\n");
  for(i=1;i<=nostate;i++)  {   c=0;
      for(j=0;j<20;j++){
          buffer[j]=0;
          e_closure[i][j]=0; }
      findclosure(i,i); }
  printf("equivalent nfa without epsilion\n");
  printf("start state : ");
  print_e_closure(start);
  printf("\nalphabets : ");
  for(i=0;i<noalpha;i++){
      printf("%c",alphabet[i]); }
  printf("\nstates : ");
  for(i=1;i<=nostate;i++){
      print_e_closure(i); }
  printf("\ntransitions are ..... \n");
  for(i=1;i<=nostate;i++)
```

```
{ for(j=0;j<noalpha-1;j++)  {
        for(m=1;m<=nostate;m++){
            set[m]=0;  }
        for(k=0;e_closure[i][k]!=0;k++){
            t=e_closure[i][k];
            temp=transition[t][j];
            while(temp!=NULL)  {
                unionclosure(temp->st);
                temp=temp->link;  }}
        printf("\n");
        print_e_closure(i);
        printf("%c\t",alphabet[j]);
        printf("{");
        for(n=1;n<=nostate;n++){
            if(set[n]!=0){
                printf("q%d,",n);} }
        printf("}");}  }
    printf("\nfinal states : ");
    findfinalstate();
    printf("\n"); }
void finddclosure(int x,int sta){
    struct node *temp; int i;
    if(buffer[x])
        return;
    e_closure[sta][c++]=x;
    buffer[x]=1;
    if(alphabet[noalpha-1]=='e'&&transition[x][noalpha-1]!=NULL){
        temp=transition[x][noalpha-1];
        while(temp!=NULL)  {
            finddclosure(temp->st,sta);
            temp=temp->link;}}
```

```
void insert_trantbl(int r,char c,int s){
    int j; struct node *temp;  j=findalpha(c);
    if(j==999)  {
        printf("error\n");
        exit(0); }
    temp=(struct node*)malloc(sizeof(struct node));
    temp->st=s;
    temp->link=transition[r][j];
    transition[r][j]=temp;}
int findalpha(char c){ int i;
    for(i=0;i<noalpha;i++){
        if(alphabet[i]==c)  {
            return i;}  }
    return(999);}
void unionclosure(int i){
    int j=0,k;
    while(e_closure[i][j]!=0) {
        k=e_closure[i][j];
        set[k]=1;  j++;}}
void findfinalstate(){ int i,j,k,t;
    for(i=0;i<nofinal;i++) {
        for(j=0;j<=nostate;j++){
            for(k=0;e_closure[j][k]!=0;k++)  {
                if(e_closure[j][k]==finalstate[i])  {
                    print_e_closure(j);  }}}}
void print_e_closure(int i){ int j;
    printf("{");
    for(j=0;e_closure[i][j]!=0;j++){
        printf("q%d",e_closure[i][j]); }
    printf("}\t");}
```

Output:

enter the no of alphabets 3
enter alphabet a b c
enter no of state 5
enter start state 3
enter no of final state 1
enter final state 3
enter no of transition 7
enter transition

1 a 2
1 c 3
2 a 1
3 b 4
4 b 3
3 a 5
5 b 3

Equivalent NFA without Epsilon
start state : $\{q_3\}$
alphabet : abc
States : $\{q_1, q_3\}$ $\{q_2\}$ $\{q_4\}$ $\{q_5\}$
Transition are ..

$\{q_1, q_3\}$ a $\{q_2, q_5\}$
$\{q_1, q_3\}$ b $\{q_4,\}$
$\{q_2\}$ a $\{q_1, q_3\}$
$\{q_4\}$ b $\{\ \}$
$\{q_3\}$ a $\{q_5\}$
$\{q_3\}$ b $\{q_4\}$
$\{q_1\}$ a $\{\ \}$
$\{q_4\}$ b $\{q_3,\}$
$\{q_5\}$ a $\{\ \}$
$\{q_5\}$ b $\{q_3,\}$

final state : $\{q_1, q_3\}$ $\{q_3\}$

RESULT

The program executed sucessfully and output obtained.

```
DFA MINIMIZATION
#include <stdio.h> #include <string.h>
#define STATES 6 #define SYMBOLS
void prntdfa(int tab[][SYMBOLS], int nst, int nsym, char *fnl) { int i, j;
puts("\nDFA: STATE TRANSITION TABLE");
for (i = 0; i < nsym; i++) printf(" %c ", '0' + i);
printf("\t+--");
for (i = 0; i < nsym; i++) printf("        ");
printf("\n");
for (i = 0; i < nst; i++) {
printf(" %c | ", 'A' + i); for (j = 0; j < nsym; j++)
printf(" %c ", tab[i][j]); printf("\n");}
printf("Final states = %s\n", fnl);}
void nxtst(char *nextstates, char *cur_states, int dfa[STATES][SYMBOLS], int symbol) { for (int i
= 0; i < strlen(cur_states); i++)
*nextstates++ = dfa[cur_states[i] - 'A'][symbol];
*nextstates = '\0';
}
char eqivndx(char ch, char stnt[][STATES + 1], int n) { for (int i = 0; i < n; i++)
if (strchr(stnt[i], ch))
return i + '0';return -1;} char is1nxtst(char *s) {
char equiv_class; while (*s == '@')
s++;
equiv_class = *s++; while (*s) {
if (*s != '@' && *s != equiv_class) return 0;
s++;}
return equiv_class;}
int stidx(char *state, char stnt[][STATES + 1], int n, int *pn, int cur) { int i;
char state_flags[STATES + 1]; if (!*state)
return -1;
for (i = 0; i < strlen(state); i++)
state_flags[i] = eqivndx(state[i], stnt, n); state_flags[i] = '\0';
printf(" %d:[%s]\t--> [%s] (%s)\n", cur, stnt[cur], state, state_flags);
if ((i = is1nxtst(state_flags))) return i - '0';
else {strcpy(stnt[*pn], state_flags); return (*pn)++;
}}
int init_equiv_class(char statename[][STATES + 1], int n, char *fnl) { int i, j;
if (strlen(fnl) == n) {
strcpy(statename[0], fnl); return 1;}
strcpy(statename[1], fnl); for (i = j = 0; i < n; i++) {
if (i == *fnl - 'A') { fnl++;
} else
statename[0][j++] = i + 'A';
}
statename[0][j] = '\0'; return 2;
}
int getoptdfa(char stnt[][STATES + 1], int n, int dfa[][SYMBOLS], int nsym, int newdfa[]
[SYMBOLS]){
int n2 = n;
int i, j;
char nextstate[STATES + 1]; for (i = 0; i < n; i++) {
for (j = 0; j < nsym; j++) { nxtst(nextstate, stnt[i], dfa, j); newdfa[i][j] =
stidx(nextstate, stnt, n, &n2, i) + 'A';
}}return n2;}
void chrappend(char *s, char ch) { int n = strlen(s);
*(s + n) = ch;
*(s + n + 1) = '\0';}
void sort(char stnt[][STATES + 1], int n) { int i, j;
char temp[STATES + 1]; for (i = 0; i < n - 1; i++)
for (j = i + 1; j < n; j++)
if (stnt[i][0] > stnt[j][0]) {strcpy(temp, stnt[i]); strcpy(stnt[i], stnt[j]); strcpy(stnt[j], temp);
```


PROGRAM - 4

DFA MINIMIZATION

AIM: Write a C program to minimize a DFA.

ALGORITHM:

1. Remove all the states that are unreachable from the initial state na any set of transition of DFA.
2. Draw transition table for all pair of states.
3. Split the table into 2 tables T1 & T2. T1 has all final state and T2 has non-final state
4. Find similar rows from T1 such that $\delta(q,a) = p$, $\delta(r,a) = p$
5. Repat step 3 until we find no similar row available in the transition table T1.
6. Repeat step 3 and step 4 for table T2 also.
7. Now combine the reduced T1 & T2 tables. The combined transition table is the transition table of minimized DFA.

```
int spliteqcls(char stnt[][STATES + 1], int i1, int i2, int n, int ndfa) { char *old = stnt[i1], *vec =
stnt[i2];
int i, n2, flag = 0;
char newsts[STATES][STATES + 1]; for (i = 0; i < STATES; i++)
newsts[i][0] = '\0';
for (i = 0; vec[i]; i++)
chrappend(newsts[vec[i] - '0'], old[i]); for (i = 0, n2 = n; i < ndfa; i++) {
if (newsts[i][0]) {
if (!flag) {
strcpy(stnt[i1], newsts[i]); flag = 1;
} else
strcpy(stnt[n2++], newsts[i]);}}
sort(stnt, n2); return n2;
}
int setneweqcls(char stnt[][STATES + 1], int n, int newdfa[][SYMBOLS], int nsym, int ndfa) { int i,
j, k;
for (i = 0; i < n; i++) {
for (j = 0; j < nsym; j++) {
k = newdfa[i][j] - 'A'; if (k >= n)
return spliteqcls(stnt, i, k, n, ndfa);
}}
return n;}
void prnteqcls(char stnt[][STATES + 1], int n) { printf("\nEQUIV. CLASS CANDIDATE ==>");
for (int i = 0; i < n; i++)
printf(" %d:{%s}", i, stnt[i]); printf("\n");}
int optimize_DFA(int dfa[][SYMBOLS], int ndfa, int nsym, char *fnl, char stnt[][STATES + 1], int
newdfa[][SYMBOLS]) {
int n = init_equiv_class(stnt, ndfa, fnl), n2; while (1) {
prnteqcls(stnt, n);
n2 = getoptdfa(stnt, n, dfa, nsym, newdfa); if (n != n2)
n = setneweqcls(stnt, n, newdfa, nsym, ndfa);
else
break;
}
return n;}
int issubset(char *s, char *t) { for (int i = 0; *t; i++)
if (!strchr(s, *t++))
return 0;
return 1;}
void getnewfnst(char *newfnl, char *oldfnl, char stnt[][STATES + 1], int n) { for (int i = 0; i < n;
i++)
if (issubset(oldfnl, stnt[i]))
*newfnl++ = i + 'A';
*newfnl++ = '\0';
}
int main(void) {
int DFAtab[STATES][SYMBOLS] = {
{'B','C'},
{'E','F'},
{'A','A'},
{'F','E'},
{'D','F'},
{'D','E'}
}, OptDFA[STATES][SYMBOLS];

char *DFA_fnl = "EF", StateName[STATES][STATES + 1], NEW_fnl[STATES + 1];
prntdfa(DFAtab,
STATES, SYMBOLS, DFA_fnl);int N_optDFA_states = optimize_DFA(DFAtab, STATES, SYMBOLS,
DFA_fnl,StateName, OptDFA);
getnewfnst(NEW_fnl, DFA_fnl, StateName, N_optDFA_states); prntdfa(OptDFA,N_optDFA_states,
SYMBOLS, NEW_fnl); return 0;
}
```

Output :

DFA : state transition table

	0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

Final state = EF

Equiv class candidate ⇒ 0: [ABCD] 1: [EF]

0 : [ABCD] → [BEAF] (0101)
0 : [ABCD] → [CFAE] (0101)
1 : [EF] → [DD] (00)
1 : [EF] → [FE] (11)

Equiv class candidate ⇒ 0: [AC] 1 : [BD] 2 : [EF]

0 : [AC] → [BA] (10)
0 : [AC] → [CA] (00)
1 : [BD] → [EF] (22)
1 : [BD] → [FE] (22)
2 : [EF] → [DD] (11)
2 : [EF] → [FE] (22)

Equiv class candidate ⇒ 0: [A] 1 : [BD] 2 : [C] 3 : [EF]

0 : [A] → [B] (1)
0 : [A] → [C] (2)
1 : [BD] → [EF] (33)
1 : [BD] → [FE] (33)
2 : [C] → [A] (0)
2 : [C] → [A] (0)
3 : [EF] → [DD] (11)
3 : [EF] → [FE] (3,3)

DFA state transition table

	0	1
A	B	C
B	D	D
C	A	A
D	B	D

Final state = D.

RESULT

program executed sucessfuly and expected result was obtained

LEXICAL ANALYZER

The Role of lexical Analyzer:

The lexical analyzer is the first phase of a compiler. Its main task is to read the input character and produce as output, a sequence of tokens that the parser uses for syntax analysis. This interaction is commonly implemented by making the lexical analyzer be a subroutine of the parser. upon receiving a get next token command from the parser the lexical analyzer reads the input character until it can find the next token. Sometimes lexical Analyzer are divided into a cascade of two phases, the first called Scanning and second lexical Analyzer. The scanner is responsible for doing simple task and lexical analyzer does the compiler task.

Issues in lexical Analysis:

There are several reason for separating the analysis phase of compiling into lexical analysis and parsing

1) simple Design
2) Efficiency
3) Portability

Tokens, Patterns, Lexemes

There is a set of strings for which the same token is produced as output. This set of string is described by a rule called pattern associated with the token. The pattern is said to match each string in the set. A lexeme is a sequence of character in the source program that is matched by the pattern for a token.

We treat token as terminal symbol in the grammar for the source language using bold face names to represent

tokens. The lexeme matched by the pattern for the token represents strings of characters in the source program that can be treated together as a lexical unit. A pattern is a rule describing the set of lexemes that can be used to represent a particular token in the source program. In many languages, certain strings are reserved i.e, their meaning is predefined and cannot be changed by the user. If keyword are not reserved, then the lexical analyzer must distinguish b/w a keyword and a user defined identifier.

Attribute of tokens:
When more than one pattern matches a lexeme, the lexical analyzer must provided additional information about the particular lexeme that matched to the subsequent phases of the compiler. The lexical analyzer collects information about tokens into their associated attribute. The tokens influence parsing decision, the attribute influence the translation of tokens.

Lexical Errors:
Few errors are discernible at the lexical level itself, because a lexical analyzer has a very localized view of source program. If the string fi is encountered in a C program for the first time in the context fi (a = f(x))
A lexical analyzer can't tell whether fi is misspelling of the keyword if or an undeclared function identifier. Since fi is a valid identifier, the lexical analyzer must return the token for an identifier and let some other phase of compiler handle any error. But suppose a situation does arise in which

the lexical analysis is unable to proceed because none of the patterns for tokens matches a prefix of the remaining. perhaps the simplest recovery strategy is panic mode recovery. we delete successive characters from the remaining input until a well-informed data is formed. This recovery technique may occasionally confuse the parser. other possible error-recovery actions are:

1. deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent character.

Input Buffering

There are 3 general approaches to the implementation of a lexical analyser.

1. uses a lexical analyser generator to produce the lexical analyser from a regular expression based specification.
2. write the lexical analyzer in a conventional system programming language, using the I/O facilities of the language to read the input.
3. write the lexical analyzer in assembly language and explicitly manage the reading of input.

Buffer Pairs:

we use a buffer divided into 2 N-character halves

N is the number of characters in one disk block.

[E = M * C * * 2 eof]

↑ ↑ forward

lexeme beginning

we read N input character into each half of the buffer with one system read command, rather than invoking a read command for each input character. If fewer than N character remain in the input than a special character eof is read into buffer after the input character i.e eof mark the end of the source file and is different from any input character.

Two pointers to the input buffer are maintained the string of characters between the two pointers is the current lexeme. Initially, both the pointers point to first character of next lexeme to be found. Forward pointer scans ahead until a match for a pattern is found. After that both pointers are set to the character immediately past of the lexeme. When the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of buffer.

Sentinals:

We add a sentinal character at end of each half in addition to end of input file. This reduces the no of comparison required. Whenever a sentinal character is scanned, we check to see whether it denotes the end of 1 half of end of input file.

Specification of Tokens:

Regular expression are an important notation for specifying patterns. Each pattern matches a set of string, so regular expressions will serve as names for set of string.

Strings & Languages

The term alphabet / character class denote any finite set of symbols. A string over some alphabet is a finite sequence of symbol drawn from the alphabet

Regular Expression: In pascal, identifier is a letter followed by zero or more letter or digit with regular expression we can define pascal identifier as letter (letter/digit)*. The vertical bar means 'or' the parenthesis are used to group subexpression, the * means zero or more instance of parenthesis expression. There are the rules that define the regular expression over alphabet Σ. 1) ε is a regular expression that denote {ε} ie the albapet containing empty string.

2) If a is a symbol in Σ, then a is a regular expression that denote {a} ie the set containing the string a. Although we use the same notation for all three, technically the RE is different from string a or the symbol a. It will be clear from the context when we are talking about 'a' as a regular exp string or symbol.

3) Suppose r & s are RE denoting the languages L(r) & L(s) then a (r)|(s) is a RE denoting L(r) ∪ L(s). b (r)(s) is a RE denoting L(r) L(s) (c) (r)* is a RE denoting L(r)* d. (r) is a regular expression denoting L(r).

Regular definition: For notational convenience, we may wish to give names to RE, and to define RE using these names as if they were symbol. If Σ is an alphabet of basic symbol, then Regular definition is a sequence

of definition of the form $d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
$d_n \rightarrow r_n$ where each d_i is a distinct name & each r_i is regular exp over the symbol in $\Sigma \cup \{d_1, d_2 .. d_{i-1}\}$
ie, the basic symbol & previously defined name. By restricting each r_i to symbol of Σ and previously defined names. we construct a RE names by the expression denote. If r_i is used d_j for some $j \geq i$ then r_i might be recursively defined, & substitution process terminate.

Recognition of Tokens :

Our goal is to construct a lexical analyser that will isolate the lexeme for the next token in the input buffer & produce as output a pair consisting of appropriate token and attribute value. Implementing transition diagram; a sequence of transition dig can be converted into a program to look for tokens specified by dig. we adopt a systematic approach that work for all transition dig & construct program whose size is proportional to the number of states and edges in the diagram.

Each state get a segment of code. If there are edge leaving a state, then its code reads a character and selects an edge to follow if possible. A function nextchar() is used to read the next character from input buffer. Advance the forward pointer at each call, and return the character read. If there is an edge labeled by the character read, or labeled by character class containing the character read, then control is transferred to the code for the state pointed by that edge. If there is no such edge, and curr state is not one that indicate a token has been

found, then a routine fail() is involved to retract the forward pointer to the position of the begining pointer and to initiate a search for token specified by the next transition diag to by fail() call an error recovery routine.

We can use a case statement to find the start state of next transition dig. edge in transition dig are traced by repetedly selecting the code fragment to determine the next state

RESULT

Studied about Lexical Analyser.

LEXICAL ANALYSER USING C

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void main(){
    FILE *f1;
    char c,str[10];
    int lineno=1,num=0,i=0;
    printf("\nenter the c program\n");
    f1=fopen("input.txt","r");
    while((c=getc(f1))!=EOF) {
        if(isdigit(c)){
            num=c-48;
            c=getc(f1);
            while(isdigit(c))  {
                num=num*10+(c-48);
                c=getc(f1);
            }
            printf("%d is a number\n",num);
            ungetc(c,f1);}
        else if(isalpha(c)){
            str[i++]=c;
            c=getc(f1);
            while(isdigit(c)||isalpha(c)||c=='_'||c=='$'){
                str[i++]=c;
                c=getc(f1); }
            str[i++]='\0';

if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str)==0||strcm
```


PROGRAM - 5

IMPLEMENT LEXICAL ANALYZER

AIM:

write a C program to implement Lexical analyzer.

```
p("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",str)==0||strcm
p("switch",str)==0||strcmp("case",str)==0)
            {
                printf("%s is a keyword\n",str);}
        else  {
                printf("%s is an identifier\n",str);}
            ungetc(c,f1);
            i=0;}
        else if(c==' '||c=='\t'){
            printf("\n");}
        else if(c=='\n'){
            Lineno;}
        else if(c=='+'||c=='-'||c=='*'||c=='/'||c=='=')  {
            printf("%c is an operator\n",c); }
        else
        { printf("%c is a special symbol\n",c);
        }}
    printf("total no. of lines are : %d\n",lineno);
    fclose(f1);
```

output:

Enter C program

```
int main() {
int a = 10, 20;
char ch; float f;
}
```

The number are : 10 20

keyword & identifiers are:

int keyword
main identifier
a identifier
char keyword
ch identifier
float keyword
f identifier

Special characters ( ) { = , ; ; ; }

Total no. of lines are : 5

RESULT:

The program executed successfully and expected result was obtained.

INTRODUCTION TO LEX

lex is a tool that has been widely used for specifying lexical analyzer for a variety of languages. we refer to the tool as the lex compiler and its input specification as the lex language. Specification of patterns using regular expression can be combined with actions that a lexical analyzer may be required to perform lex like specification can be used even if a lex compiler is not available. lex is generally used in manner follows first a specification of a lexical analyzer is prepared by creating a program lex.l in the lex language. Then lex.l is run through the lex compiler to produce C program lex.yy.c. The program lex.yy.c consist of tabular representation of a transition dig. constructed from regular expression of lex.l, together with standard routine that uses the table to recognize lexeme. The actions associated with regular expression in lex.l are pieces of C code & are copied over directly to lex.yy.c. finally lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transform an input stream into a sequence of tokens.

lex specification: A lex program consist of three parts
declaration
%%
transition rules
%%
Auxilary procedure

The declaration sect include declaration of variable manifest constant and regular definition. The regular definition are statement similar to those given below

letter → A|B|...|z|a|b|...|z|
digit → 0|1|2|...9
letter → letter (letter|digit)*

the translation rules of a lex program are statement of form

P_1 {action 1}

:

P_n {action n} where each P_i is a regular expression & each action is a program through fragment describing what action the lexical analyzer should taken when patter P_i matches lexem. In lex, the actions are written in C, in general. The third section hold whatever auxiliary program are needed by actions. Alternatively these procedure can be compiled separately and loaded with Lexical analyzer finds the longest prefix that matches by any P_i in the remaining input and performance corresponding action 1. Typically action 1 return control to the parser. The lexical analyzer returns token to the parser. To pass attribute value, we can set a global variable called yyval.

LEXICAL ANALYSER USING LEX TOOL

```
%{ #include<stdio.h> %}
delim [ ]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+(\.{digit}+)?(E[+|-]?{digit}+)?
%%
ws {printf("no action");}
if|else|then {printf("%s is a keyword",yytext);} // TYPE 32 KEYWORDS
{id} {printf("%s is a identifier",yytext);}
{num} {printf(" it is a number");}
"<" {printf("it is a relational operator less than");}
"<=" {printf("it is a relational operator less than or equal");}
">" {printf("it is a relational operator greater than");}
">=" {printf("it is a relational operator greater than");}
"==" {printf("it is a relational operator equal");}
"<>" {printf("it is a relational operator not equal");}
%% main() { yylex(); }
```

Output.

```
lex lexical.l
cc. lex.yy.c -ll
if is a
keyword
number
number is a
identifier 254
its a number <>
its relational operator not
equal ^%.
```


PROGRAM 6.1

LEXICAL ANALYZER USING LEX TOOLS

AIM: To implement the lexical analyzer using lex tool.

RESULT: The program was executed successfully & expected result was obtained

COUNT VOWEL AND CONSONANTS USING LEX TOOL

```
%{ #include <stdio.h>
#include<string.h>
int vow_count=0; int const_count=0; %}  %%
[aeiouAEIOU]* {vow_count++;}
[a-zA-Z] {const_count++;} "\n"
{printf("Number of vowels are: %d\n",vow_count),printf("Number of consonants are:%d\n",
const_count);}
%% int yywrap(void) {}
int main() { printf("Enter the strings of vowels and consonants:"); yylex(); return 0;
```

output.

Enter the string of vowel & constnants : hello
no of vowels : 2
no of consonant : 3

6.b. A LEX PROGRAM TO COUNT VOWELS & CONSONANT

AIM : Write a lex program to count vowels and consonant in a sentence.

RESULT :

The program was successfully implemented & output is obtained.

PROGRAM TO COUNT KEYWORDS, NUMBER AND WORDS IN A STRING

```
%{
#include<stdio.h>
%} %%
if |else |printf {printf("%s is a keyword",yytext);}
[0-9]+ {printf("%s is a number", yytext);}
[a-zA-Z]+ {printf("%s is a word", yytext);}
.\\n {ECHO;}
%%
int main(){
printf("\n enter string:");
yylex(); }
int yywrap(){
return 1;
}
```

Output.

enter string: hello 234
hello is word
234 is number

63) LEX PROGRAM TO COUNT KEYWORDS, NUMBER & WORD

AIM: write a lex program to count keywords, number, & words in a string.

RESULT:
The program was succesfully implemented & output is obtained.

PROGRAM TO RECOGNIZE KEYWORDS, IDENTIFIERS

```
%{
#include <stdio.h>
%}
delim [\t]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+(\.{digit}+)?(e[+|-]?{digit}+)?
%%
ws {printf("no action");}
if|else|then {printf(" %s is a keyword",yytext);}//TYPE 32

{id} {printf(" %s is a identifier",yytext);}
{num} {printf(" it is a number");}
"<" {printf("it is a relational operator less than");}
"<=" {printf("it is a relational operator less than or equal");}
">" {printf("it is a relational operator greater than");}
">=" {printf("it is a relational operator greater than or equal");}
"==" {printf("it is a relational operator equal");}
"<>" {printf("it is a relational operator not equal");}
%%
void main(){
yylex();}
```

Output:

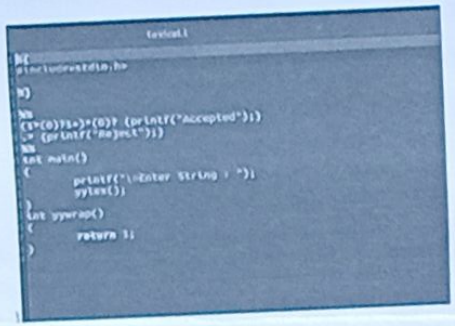
< relational operator less than
== relational operator equal

6.A) LEX PROGRAM TO RECOGNIZE KEYWORDS, NUMBERS, OPERATORS AND IDENTIFIERS

AIM: write a lex program to recognize keywords, number operators & identifiers.

RESULT

The program was successfully completed & output is obtained.



Output:

Enter string

10 string is valid.

100 string is invalid

6.5) LEX PROGRAM TO RECOGNIZE ALL STRING WITHOUT TWO CONSECUTIVE ZERO'S.

AIM: Write a lex program to recognize all string without two consecutive zeros.

RESULT

The program was successfully implemented & output is obtained.

```
lexical1                                                    vow.l
%{
#include<stdio.h>

%}

%%
a*ba*b*(b)bza*b*b* {printf("Accepted")}
.* {printf("Reject")}
%%
int main()
{
        printf("\nEnter String : ");
        yylex();
}
int yywrap()
{
        return 1;
}
```

Output:

Enter a string: c aba c

c aba c is invalid

6.6) LEX PROGRAM TO RECOGNIZE STRING WHICH DOESN'T CONTAIN SUBSTRING ABA

AIM: Write a lex program to recognize string which doesn't contain substring 'aba'.

RESULT:

The program was successfully implemented & Output is obtained.

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

%}

%%
((aa)*(bb)*|(aa)*a(bb)*b) {printf("Accepted");}
.* {printf("Reject");}
%%
int main()
{
        printf("\nEnter String : ");
        yylex();
}
int yywrap()
{
        return 1;
}
```

Output

Enter string : aaab

aaab is valid

Enter string: aabbbb

aabbbb is valid.

6.7) LEX PROGRAM TO ACCEPT ALL STRING OF FORM $a^m b^n$

AIM: Write a lex program to accept all string of form $a^m b^n$, where m+n is even.

RESULT:

The program was implemented successfully & output is obtained.

INTRODUCTION TO YACC

A parser generator can be need to facilitate the construction of the front end of a compiler. Yacc is LALR parser generator. Yacc stands for yet another compiler-compiler. The parser Generator Yacc.

A translator can be constructed using Yacc in the manner is illustrated.

First a file say translate.y, containing a yacc specification of translator is prepared. The UNIX system command, yacc translate.y transform the file translate.y into a program called y.tab.c using LALR method. The program y.tab.c is a representation of LALR parser written in C along with other C routines that the user may have prepared. By compiling y.tab.c along with library that contains the LR program using the command, cc y.tab.c -ly. The desired object program a.out is obtained that the translation specified by the original yacc program. If other procedures are needed, they can be compiled or loaded with y.tab.c.

A yacc program has 3 part

declaration

%%

translation rule

%%

supporting C routines

There are two optimal soln in the declaration part of yacc program. In the 1st section ordinary C program declaration that calls a C preprocessor to include the header file. Then its also possible for declaration of grammar tokens. token declared

in this can be used to the next two parts. The translation rule part: Each rule consist of a grammar production and associate semantic form. A set of production <leftside> → <alt-1> | alt2<> | <altn>. would be written yacc as

<leftside> : <alt1> {semantic action1}
: <altn> {semantic action n}

In yacc production, a quoted single character is taken to determined symbol and unquoted strings of letters and digit not declared to be tokens are taken to be non-terminal. Alternative right side can be separated by a vertical bar and a semicolon follows each leftside. A yacc semantic action, the symbol $$ refers to the attribute value associated with non-terminal on left while $i refers to the value associate with grammar formed whenever we produce reduce by associated production. The supporting C routine. The third part of a semantic yacc specification consist of supporting C routines. The lexical analyzer yylex() produces pair consisting of token and its associate attribute value. The attribute value is communicate to parser through a yacc defined variable yylval. Yacc resolves shift reduce conflict that occur in ambiguous grammar by attaching a precedence of associativity to each production involved in a conflict, as well as to each terminal involved in a conflict.

functions in lex & yacc

1. yylex() : Its responsible for performing lexical analysis by recognizing pattern in input and returning tokens to parser (yacc)

2. yytext : A variable that contains the matched text or lexeme recognized by current rule/pattern in the input.

3. yyparse(): This function is entry point for the parser generated by yacc. It control the parsing process according to the grammar rule defined and invokes the lexer to get tokens.

4. yyerror: used for error handling the parsing process

5. yylval: A variable used to pass semantic value from lexer to parser.

6. yywrap(): function in lex is used to enable multiple input source or signal the end of input to lexical analyzer.

7) yyin : typically points to input stream or file from which lexical analyzer read its input tokens by default it set to stdin

8) yyout points to the output stream where the parser writes its output such as generated code or parsed information. by default it set to stdout

9) yylen is a predefined variable used in parser generators. It typically employed to retrieve length of current token being process in parsing process

SIMPLE CALCULATOR USING YACC

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",$$);
```


PROGRAM-7
SYNTAX ANALYZER USING YACC TOOL

AIM:

Write a program in yacc to perform a simple calculator for ambigous grammar.

ALGORITHM:

1. A Yacc source program has three part as follow.
   Declaration %%

```
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 which can have operations
Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
yyparse();
if(flag==0)
printf("\nentered arithmetic expression is valid\n\n");}
void yyerror(){
printf("\nentered arithmetic expression is invalid\n\n");
flag=1;}
```

output:

Enter arithmetic expression : ((5+6+10+4+5)/5)%2

Result = 0

entered arithmetic expression is valid.

RESULT:

The program was implemented successfully and output obtained.

RECURSIVE DESCENT PARSER

The parser contains a string of tokens from the lexical analyzer and verify that string can be generated by the grammar for source language. Most parsing methods fall into one of two classes called top down and bottom up methods. These term refer to order in which nodes in the parse tree are constructed. In the former, the construction start at root & proceed toward leaves, while in latter, construction starts at leaves & proceed toward root. The top down construction of parse tree is done by starting with root labelled with string starting non-terminal and repeatedly perform steps.

1. At node n, labelled with non-terminal A, select one of the production for A and construct children at n for symbol on rightside of production.
2. Find next node at which a subtree is to be constructed

Recursive Descent Parser:

Its top down parsing method that may involve repeatedly scan of input ie, backtracking. However backtracking process are not seen frequently one reason is that backtracking rarely need to parse programing lang

Construct grammar $S \rightarrow cAd$

$A \rightarrow ab/a$ and input string $w = cad$. To construct a parse tree for this top down, we initially create a tree consist of a string node labelled S. An input pointer point to c first symbol of w the left most labeled c, matches the first symbol of w, we now advance the input pointer to a, the second symbol of w & consider the next leaf labelled A. We can then expand A using the first alternative for obtain tree in fig (a). We now have a match for second input symbol so we advance the input pointer to third i/p symbol so we advance the

and compare d against the next leaf b. Since d doesn't match, we report failure and go back to A to see whether there is another alternative for A that we have tried but that might produce a match. In going back to A, we must reset the input pointer to position 2 position it had when first came to A, which means that procedure for A, must store input pointer in local variable. In C, the leaf a matches the second symbol of w and leaf d match third symbol. Since we have produced a parse tree for w, we halt and announce the successful completion of parsing. A left recursive grammar can cause a recursive descent parser, even one with backtracking, to go into an infinite loop. ie if we try to expand A, we may find ourselve trying to expand A without having consumed any input.

RECURSIVE DESCENT PARSER

```
#include<stdio.h>
#include<string.h> #include<ctype.h>
char input[10];
 int i=0,error=0;
void e();void t();void eprime();
void tprime();
void f();
int main(){

    printf("enter an arithmetic expression : ");
    gets(input);
    e();
    if(strlen(input)==i&&error==0) {
        printf("\naccepted...!!!\n\n");  }
    else{
        printf("\nrejected...!!!\n\n");}
    return 0;}
void e(){
    t();   eprime();}
void eprime(){
    if(input[i]=='+'){
        i++;  t();
        eprime(); } }
void t(){
f(); tprime();}
void tprime(){
    if(input[i]=='*'){
        i++;  f();
        tprime(); } }
void f(){
    if(input[i]=='('){
        i++;e();
        if(input[i]==')'){
        else{error=1; }    else if(isalpha(input[i])){
        i++;
        while(isalnum(input[i])||input[i]=='.')  {
            i++  }} else  error=1  } }
```


PROGRAM - 8

RECURSIVE DESCENT PARSER

AIM : write a program to construct recursive descent parser for an expression.

Output :

Enter arithmetic expression :
sum + month * intrest
Accepted.

Enter arithmetic expression :
sum + avg * intrest
Rejected.

SHIFT REDUCE PARSER

Bottom up parsing also known as shift reduce parsing. It has two main action are shift & reduce. At each shift, the current symbol input string pushed to stack. At each reduction step, symbols are top of the stack will be replace by non-terminal at left side of production. There are also two more action: accept & error.

A shift reduce parser tries to reduce the given input string to start symbol a string reduce to starting symbol.

At each reduction step, a substring of input matching to right side of production rule is replaced by non-terminal at LHS rule. If the substring choosen correctly the right most derivation created in reverse order. $S \Rightarrow w$

$w \Leftarrow S$

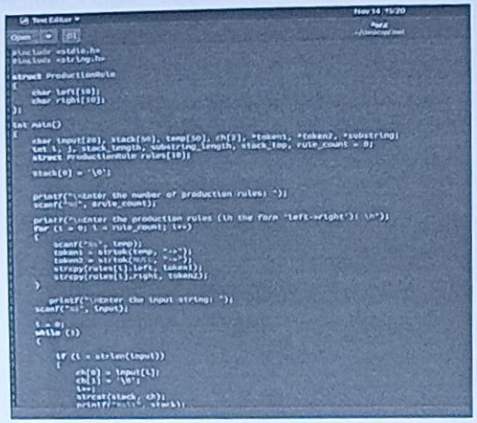
Handles: Substring that match RHS of some production and which reduction to non terminal on LHS represent one step along reverse order of right most derivation. A handle of a right sential form γ ($\alpha\beta w$) is a production rule $A \to \beta$ and position of γ where string β may be found and replaced by A to produce previous right sential form $S \Rightarrow \alpha A w \Rightarrow \alpha\beta w$ ie $A \to \beta$ is handle of $\alpha\beta\gamma$ at loc immediately after end of α. If the grammar is unambiguous then every right sential form has 1 handle.

Four possible actions of shift reduce parser

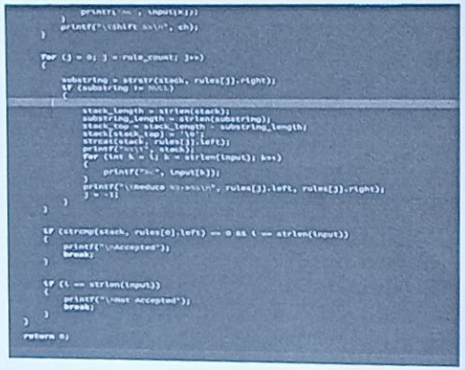
* Shift: next input symbol shift to top of stack
* Reduce: The right end of string to be reduced on top of stack locate left end of string with stack. replace handle top of stack by NT
* Accept: Sucessful completion of parsing
* Error: parser discover a syntax error, call error recovery routine. Conflict during SR parser are:

Shift reduce conflict & reduce/reduce conflict.

SHIFT REDUCE PARSER



```
Text Editor                                                    Nov 14 15:20
Open ▾   ⊡                                                     *untitled
                                                               ~/Desktop/untitled
#include <stdio.h>
#include <string.h>

struct ProductionRule
{
    char left[10];
    char right[10];
};

int main()
{
    char input[20], stack[50], temp[50], ch[2], *token1, *token2, *substring;
    int i, j, stack_length, substring_length, stack_tmp, rule_count = 0;
    struct ProductionRule rules[10];

    stack[0] = '\0';

    printf("\nEnter the number of production rules: ");
    scanf("%d", &rule_count);

    printf("\nEnter the production rules (in the form 'left->right'): \n");
    for (i = 0; i < rule_count; i++)
    {
        scanf("%s", temp);
        token1 = strtok(temp, "->");
        token2 = strtok(NULL, "->");
        strcpy(rules[i].left, token1);
        strcpy(rules[i].right, token2);
    }

    printf("\nEnter the input string: ");
    scanf("%s", input);

    i = 0;
    while (i)
    {

        if (i < strlen(input))
        {
            ch[0] = input[i];
            ch[1] = '\0';
            i++;
            strcat(stack, ch);
            printf("%s\t", stack);
```



```
            printf("%c", input[k]);
        }
        printf("\tShift %s\n", ch);
    }

    for (j = 0; j < rule_count; j++)
    {

        substring = strstr(stack, rules[j].right);
        if (substring != NULL)
        {

            stack_length = strlen(stack);
            substring_length = strlen(substring);
            stack_tmp = stack_length - substring_length;
            stack[stack_tmp] = '\0';
            strcat(stack, rules[j].left);
            printf("%s\t", stack);
            for (int k = i; k < strlen(input); k++)
            {
                printf("%c", input[k]);
            }
            printf("\tReduce %s->%s\n", rules[j].left, rules[j].right);
            j = -1;
        }
    }

    if (strcmp(stack, rules[0].left) == 0 && i == strlen(input))
    {
        printf("\nAccepted");
        break;
    }

    if (i == strlen(input))
    {
        printf("\nNot Accepted");
        break;
    }
}

return 0;
```

PROGRAM - 9

AIM SHIFT REDUCE PARSER

AIM: Write C program to implement shift-reduce parser for given grammar.

output

enter input string i+i*i$

Stack	Input string	Action
$<i	+i*i$	Reduce
$<i	+i*i$	shift
$<+	i*i$	shift
$<+<i	*i$	Reduce
$<+<i	*i$	shift
$<+<*	i$	shift
$<+<*<i	$	Reduce
$<+<*<i	$	Reduce
$<+<*<i	$	Reduce
$<+<*<i	$	string accepted

RESULT:

The program was successfully completed & output is obtained.

OPERATOR PRECEDENCE PARSING

Operator precedence parsing in this we define the disjoint precedence relation $\lessdot$, $\doteq$ and $\gtrdot$ b/w certain pairs of terminal. These precedence relation guide the selection of handle.

Relation	meaning
$a \lessdot b$	a "yield precedence to b"
$a \doteq b$	a has same precedence as b"
$a \gtrdot b$	a "takes precedence over b".

There are 2 method of determining what precedence shift relation should hold b/w a pair of terminal. The first method is based on traditional notion of associativity and precedence of operators. Eg * has higher precedence than + ie, $+ \lessdot *$. The second method of selecting operator precedence relation is that to construct an unambiguous grammar for language.

Using Operator precedence Relation

The intention of relation is to delimit the handle of right sentential form with $\lessdot$ marking the left end $\doteq$ appearing in the interior of handle and $\gtrdot$ marking the right end to be more precise, suppose if we have a right sentential form of an operator grammar. The fact that no adjacent non-terminal appear on the right side of production implies that we can rewrite the right sentential form as $\beta_0 a_1 \beta_1 \ldots a_n \beta_n$ where β_i is either ε (ie empty string) or a single nonterminal. Suppose the b/w a_i & a_{i+1} exactly one of relation $\lessdot$, $\doteq$ and $\gtrdot$ hold. $ makes end of string, $ define $ $\lessdot b$ and $b \gtrdot$ $ for all terminal b. Now we remove the nonterminal from string & place correct relation $\lessdot$, $\doteq$ or $\gtrdot$ b/w each pair of relation & between endmost terminal and $ marking the end of string.

for eg <· is inserted b/w leftmost $ and id since <· is the entry in row $ and column id. The handle can be removed found by the following process: · scan the string from left until ·> is encountered in the above, this occurs b/w first id and +. 2) Then scan backward to left over any ≐'s until a <· is encountered. Then we scan backward to $. The handle contain everything to left of first, and to right of the <· encountered in step 2, including any intervening or surrounding non-terminal. The intention of surrounding non-terminal is necessary so that 2 adjacent non terminal don't appear in right sentential form. If we delang with grammar E → E+E | E*E | (E) | id. Then we reduced id into E. then we have E+id*id. After reducing the remaining id to E by same steps, we get the right sentential form E+E*E. A single marker non-terminal can be kept on the stack of shift reduce parser to indicate place holder for attribute values. If we use stack of a precedence relation <· or ≐ hold b/w the top most terminal symbol on stack & the next i/p symbol, parser shifts, it has't yet found the right end of handle. If relation ·> hold the reduction is called for. The parser has found the right end of handle at this point & the precedence relation can be used to find the left end of handle in stack. If no precedence relation hold b/w a pair of terminal, semantic error has been detected.

OPERATOR PRECEDENCE PARSER

```
#include<stdio.h>
#include<string.h>
void main(){
    char stack[20],ip[20],opt[10][10][1],ter[10];
    int i,j,k,n,top=0,row,col;
    for(i=0;i<10;i++) {
        stack[i]='\0';
        ip[i]='\0';
        for(j=0;j<10;j++){
            opt[i][j][1]='\0';}}
    printf("enter no. of terminals: ");
    scanf("%d",&n);
    printf("enter the terminals : ");
    scanf("%s",ter);
    printf("enter the table values : \n");
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            printf("enter the value for %c %c : ",ter[i],ter[j]);
            scanf("%s",opt[i][j]);
        }}
    printf("\noperator presedence table\n");
    for(i=0;i<n;i++) {
        printf("\t%c",ter[i]);}
    printf("\n");
    for(i=0;i<50;i++){
        printf("-");}
    printf("\n");
    for(i=0;i<n;i++) {
```


PROGRAM - 10

OPERATOR PRECEDENCE PARSER

AIM: Write C program to develop an operator precedence parser for given language.

ALGORITHM:

Input: An input string w and table of precedence relation

Output: If w is well formed, a skeletal parse tree, with a place holder nonterminal E labelling all interior nodes. Otherwise an error indication.

Method: Initially, the stack contain $ and the input buffer, the string w$. To parse we execute the program

1. Setup ip to point to the first symbol of w$
2. Repeat forever
3. If $ is on top of the stack and input point to $ then return else begin
4. Let a be the topmost terminal symbol on stack & let b be the symbol pointed to by input.
6. If a <· b or a ≐ b then begin
7. push b onto the stack
8. advance ip to the next input symbol end;
9. else if a ·> b then /* reduce */ repeat
10. pop the stack
11. Until the top stack terminal is related by ·> the terminal most recently popped
12. else error() end.

```
    printf("\n%c",ter[i]);
    for(j=0;j<n;j++) {
        printf("\t%c",opt[i][j][0]);
    }}
stack[top]='$';
printf("\n\nenter the input string (append with $) : ");
scanf("%s",ip);
i=0;
printf("\nstack\t\t\tinput string\t\taction\n");
printf("\n%s\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip)){
    for(k=0;k<n;k++) {
        if(stack[top]==ter[k])    row=k;
        if(ip[i]==ter[k])
            col=k;
    }
    if((stack[top]=='$')&&(ip[i]=='$')){
        printf("string is accepted");
        break;
    }
    else if((opt[row][col][0]=='<')||(opt[row][col][0]=='='))
{   stack[++top]=opt[row][col][0];
        stack[++top]=ip[i];
        ip[i]=' ';
        printf("shift %c",ip[i]);
        i++;
    }
    else{
        if(opt[row][col][0]=='>')   {
            while(stack[top]!='<'){
                --top}
            top=top-1;
            printf("reduce");
        }
        else{  printf("\nstring is not accepted");
            }  }
    printf("\n");
    printf("%s\t\t\t%s\t\t\t",stack,ip);  }
printf("\n");,}
```

output:

enter terminal : i+*$

enter table values:

enter the value for ii –

i+ >

i* >

i$ >

+i <

++ >

+* <

+$ >

*i <

*+ >

** >

*$ >

$i <

$+ <

$* <

$$ -

Operator precedes table

	i	+	*	$
i	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

Enter input i+i*i$

Stack	I/P string	Action
Shift	+i*i$	reduce
$<i	+i*i$	Shift
$<i	i*i$	Shift
$<+	*i$	reduce
$<+<i	*i$	Shift
$<+<*	i$	Shift
$<+<*<i	$	~~Shift~~ reduce
$<+<*<i	$	~~Shift~~ reduce
$<+<*<i	$	reduce
$<+<*<i	$	String accepted

RESULT:

The program was successfully completed & output is obtained.

Intermediate code generation

Intermediate code generation can translate the source program into machine program. Intermediate code is generated by the compiler can't generate machine code directly in one pass. Therefore first it convert source program into intermediate code, which perform efficient generation of machine code further. The intermediate code can be represented in the form of postfix notation, syntax tree directed acyclic graph, three address codes, Quadruples and triples.

Need for ICG

If a compiler translate source language to its target machine language without generating IC, then for each new machine, a full native compiler is required. IC eliminate the need of a new full compiler for every machine by keeping analysis portion for all compiler synthesis part of back end depends on target machine.

Intermediate language

→ syntax tree
→ DAG
→ Postfix notation
→ 3 Address code

→ Syntax tree: It a hierarchial structure that represent syntactic structure of source code

→ DAG: It data structure represent computation with nodes & directed edge but without any cycles

→ postfix notation, mathematical expression without use of parantheses or explicit operator precedence. In postfix notation, operator placed after operant

→ 3 AC → Used in compiler to represent instruction & expression close to machine code but remain abstract enough

INTERMEDIATE CODE GENERATION

```
#include<stdio.h>
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
void main(){
    FILE *fp1,*fp2;
    fp1=fopen("input.txt","r");
    fp2=fopen("output.txt","w");
    while(!feof(fp1)){
        fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);
        if(strcmp(op,"+")==0{
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n ADD R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);}
        if(strcmp(op,"*")==0)
        { fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n MUL R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);}
        if(strcmp(op,"-")==0)
        {fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n SUB R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);}
        if(strcmp(op,"/")==0) {
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n DIV R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);
        }
        if(strcmp(op,"+")==0){
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n MOV %s,R0",result);}
    } fclose(fp1);  fclose(fp2);}
```

PROGRAM 11

INTERMEDIATE CODE GENERATION

AIM: write C program to implement intermediate code generation for single expression.

ALGORITHM

1. Start
2. open input file in read mode
3. Open the output file in write mode
4. In input file scan for operator, argument1, argument2 & result.
5. If operator is + MOV arg1 to R0
   add arg2 and R0
   MOV R0 to result
6. If the operator is - MOV arg1 to R0
   Subtract arg2 and R0
   MOV R0 to result
7. If the operator is * Mov arg1 to R0
   Multiply arg2 and R0
   MOV R0 to result
8. If the operator is '/' MOV arg1 to R0
   divide arg2 and R0
   MOV R0 to result
9. If operator is = MOV arg1 to R0
   MOV R0 to result
10. Close both file
11. stop

Output.

```
MOV  R0 a
ADD  R0 b
MOV  t1 R0
MOV  R0 c
ADD  R0 d
MOV  t2 R0
MOV  R0 t1
ADD  R0 t2
MOV  t R0
MOV  R0 t
MOV  x R0
MOV  R0 t
MOV  x R0
```

Output! for

```
+ a b t1
+ c d t2
- t1 t2 t
- t P x
```


RESULT:

The program was successfully compiled and output is obtained.

First & Follow

First and follow in compiler design is process of creating software that translate source code in high level language to machine code. These set used to generate predictive parser. specially in LL(1) grammar

First () : is a function that specifies the set of terminals that start a string derived from a production rule. It is the first terminal that appear on the RHS of production for eg:
In the input string $T \rightarrow *FT/\epsilon$, here we found that T has a production like $T \rightarrow *FT$ $T \rightarrow \epsilon$ after reading these we found the first of T in both production that which is * and ϵ. First of string $(*, \epsilon)$

Follow (): Is a set of terminal symbol that can be displayed next to the right of non-terminal symbol in any sentence format it is the first non-terminal appearing after the given terminal symbol on the RHS of production.
For eg. Input string $E \rightarrow TE'$
$F \rightarrow (E)/id$

here we found that on the RHS production statement where is the E occurs, we only found E in the production $F \rightarrow (E)/id$ through which we found follow of E. Then output Follow of $E = \{\}$ as ')' is non terminal in the input string on the RHS of production.

FIRST AND FOLLOW

```
#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; 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;
strcpy(production[0], "E=TR");
strcpy(production[1], "R=+TR");
strcpy(production[2], "R=#");
strcpy(production[3], "T=F1");
strcpy(production[4], "1=*F1");
strcpy(production[5], "1=#");
strcpy(production[6], "F=(E)");
char done[count];
pt = -1;
for(k = 0; k < count; k++) {
for(kay = 0; kay < 100; kay++) {
calc_follow[k][kay] = '!'; } }
point1 = 0;int land = 0;
for(ey = 0; e < count; e++){
ck = production[e][0];
point2 = 0;
xxx = 0;
for(kay = 0; kay <= pt; kay++)
if(ck == done[kay])
xxx = 1;
if (xxx == 1)
continue;
land += 1;
follow(ck);
pt += 1;
done[pt] = ck;
printf("\n follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;
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]; } }
```


PROGRAM 18

FIRST & FOLLOW

AIM: write a program to implement first & follow of grammar.

```
int kay; char done[count];int ptr = -1;
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;
for(kay = 0; kay <= ptr; kay++)
if(c == done[kay])
xxx = 1;
if (xxx == 1)
findfirst(c, 0, 0);
ptr += 1;
done[ptr] = c;
printf("\n first(%c) = { ", c);
calc_first[point1][point2++] = c;
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\n");
km = m;
point1 ++; } }
void follow(char c){
 int i, j;
 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'){
 followfirst(production[i][j+1], i, (j+2));}
  if(production[i][j+1]=='\0' && c!=production[i][0]){
  follow(production[i][0]);
  } } } } }
 void findfirst(char c, int q1, int q2){
  int j;
  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)) {
   findfirst(production[q1][q2], q1, (q2+1)); }
   else
   first[n++] = '#'; }
   else if(!isupper(production[j][2])) {
    first[n++] = production[j][2]; }
    else {
   findfirst(production[j][2], j, 3);
```

```
} } } }
void followfirst(char c, int c1, int c2){
int k;
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; }
while(calc_first[i][j] != '!'){
if(calc_first[i][j] != '#'){
f[m++] = calc_first[i][j]; }
else {
if(production[c1][c2] == '\0') {
follow(production[c1][0]); }
else {
followfirst(production[c1][c2], c1, c2+1); }  }
j++;} } }
```

output :

LOOP UNROLLING

```
#include <stdio.h>
void loopunrolledsum(int arr[], int n) {
int sum = 0;
 for (int i = 0; i < n-1; i += 2) {
sum += arr[i];
sum += arr[i + 1]; }
   for (int i = n – n % 2; i < n; i++) {
   sum += arr[i];
 printf("sum: %d\n", sum);}
int main() {
```

```
int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int size = sizeof(array) / sizeof(array[0]);
loopunrolledsum(array, size);
return 0;}
```

Output :

Sum : 55

PROGRAM - 13

LOOP UNROLLING

AIM: write a program to perform loop unrolling.

RESULT:

The program was succesfuly completed & output is obtained.

CONSTANT PROPOGATION

```
#include<stdio.h>#include<string.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];
```


PROGRAM 14

CONSTANT PROPOGATION

AIM: write a program to perform constant propagation

ALGORITHM:

1. Start
2. Construct a control flow graph
3. Associate transfer function with edges of CFG
4. At every node we maintain the value of program variable. at the point we initialize these to ⊥
5. Iterate until the values of variable stabilize
6. Stop.

```
    for(j=0;j<n;j++){
        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.

```
gcc prog16.c
./a.out
Enter the maximum no of expression: 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
```


RESULT:

The program was executed successfully & expected result was obtained.