26/10/18

Program : 3

GENERATING YACC SPECIFICATIONS

AIM:

To generate the following YACC specifications for the syntactic categories.

PROGRAMS

(a) Valid Expression

(i) expr.l

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

(ii) expr.y

```
%{ #include <stdio.h>
   #include <stdlib.h>
%}
% token number NL
% left '+' '-'
% left '*' '%'
%%
```

```
stmt : exp NL { printf ("valid expression \n");
                exit(0); }
;
exp: exp '+' exp { $$ = $1 + $3;} |
     exp '-' exp { $$ = $1 - $3; } |
     exp '*' exp {$$ = $1 * $3; } |
     exp '/' exp { if ($3 ==0) { printf ("cannot divide
                   by zero \n");
                 else { $$ = $1/$3 ; }} |

'(' expr ')' { $$ = $2 ; } |
  NUMBER { $$ = $1 ; }
 %%
int yyerror ( char *msg)
{ printf ("Invalid Expression \n");
      exit(0); }

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

b) Valid Variable

(1) code.l

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

```
[0-9]+ {return DIGIT;}
[a-z A-Z]+ {return LETTER;}
[\t] {;}
\n {return 0;}
{return yytext[0];}
%%
```

(ii) code.y

```
%{ #include <stdio.h>
%}
%token DIGIT LETTER
%%
stmt: A
    ;
A : LETTER B
    ;
B : LETTER B
  | DIGIT B
  | LETTER
  | DIGIT;
%%
void main()
{printf("Enter the string: \n");
  yyparse();
printf("valid");
  exit(0); }
void yyerror()
{printf "invalid"); exit(0); }
```

(c) Calculator

(i) code.l

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

(ii) code.y

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

```
Arithmetic Expression : E }
        printf("\n Result = %d \n", $$);
        return 0; };
E : 'E' + E  {$$ = $1 + $3 ;}
  | E '-' E  {$$ = $1 - $3;}
  | E '*' E  { $$ = $1 * $3;}
  | E '/' E  { $$ = $1 * $3;}
  | '('E')'  {$$ = $2 ;}
  | NUMBER   ;
%%
void main()
{ printf("\n Enter any arithmetic expression which can have the
  operations addition, subtraction, multiplication, division, modulus
  and round brackets : \n");
  yyparse();
  if (flag == 0)
  printf("\n entered arithmetic expression is valid \n\n");
    flag=1;
  }
```

RESULT:

The program is successfully implemented & results are obtained.

Output

Enter expression

a+b

Entered arithmetic expression is valid

Enter the expression

hello

Invalid expression

Output

Enter the string

9-12

valid.

Enter string

22

Invalid

Output

Enter the aethmetic expression that can have operations addition, subtraction, multiplication, division, modulus & round brackets

(2+3)*5

Result : 25

8/11/18

Program: 4.

RECURSIVE DESCENT PARSER

Aim:

To construct a recursive descent parser for given expression

PROGRAM:

```
#include <stdio.h>
# include <conio.h>
char input [100];
char prod [100][100];
int pos = -1, l, st = -1;
char id, num;
 void E();
 void T();
 void F();
 void advance();
 void Td();
 void Ed();
 void advance();
 {  pos++;
 if (pos < l)
{ if (input [pos] >= '0' && input [pos] <= '9')
    { num = input [pos];
                id = '\0'; }
```

```
if ((input[pos] >= 'a' || input[pos] >= 'A' && (input
[pos] <= 'z' || input[pos] <= 'Z' ))
    { id = input[pos];
        num = '\0'; } } }

void E()
{ strcpy (prod[++st], "E→TE'");
  T();
  Ed(); }
void Ed()
{ int p=1;
if (input[pos]=='+')
  { p=0;
  strcpy (prod[++st], "E'→+TE'");
      advance();
      T();
      Ed(); }
if (input[pos == '-')
    { p=0;
  strcpy (prod[++st], "E'→ -TE'");
      advance();
       T();
        Ed(); }
```

// Recursive descent parser

```
if (p==1)
{ strcpy (prod [++st], "E'→ null"); }}
void T().
{ strcpy (prod [++st], "T→ FT' ");
        F();
        Td(); }
void Td();
{ int p=1;
  if (input [pos]=='*')
  { p=0;
   Strcpy (prod [++st], "T'→ *FT' ");
       advance();
        F();
        Td(); }
  if (input [pos]=='/')
    { p=0;
   strcpy (prod [++st], "T'→ /FT' ");
        advance();
          F();
          Td(); }
 if (p==1)
   strcpy (prod [++st], "T'→ null"); }
```

```
void F()
{ if (input[pos]==id)
 { strcpy(prod[++st], "F→id");
   advance();
      E();
   if (input[pos]==')')
     { // strcpy(prod[++st], "F→(E)");
              advance(); }
     }
int main()
 { int i;
   printf("Enter input string");
      scanf("%s", input);
          l = strlen(input);
           input[l] = "$"
           advance();
             E();
          if (pos==l)
            { printf("string accepted \n");
              for(i=0; i<=st; i++)
               { printf("%s\n" prod[i]); }
            }
          else
            printf("string rejected \n");
          }
```

Output

Enter Input String (a+b)

String accepted

E → TE'

T → FT'

F → (E)

E → TE'

T → FT'

F → id

T' → null

E' → +TE'

T → FT'

F → id

T' → null

E' → null

T' → null

E' → null

```
getch();
  return 0;
}
```

RESULT:

Implemented recursive descent parser and output obtained successfully

8/11/18

Program: 5

SHIFT REDUCE PARSER

AIM:
To implement shift reduce parser for a given language

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
#include< string.h>
int k=0, z=0, i=0, j=0, c=0;
char a[16], ac[20], stk[15], act[10];
void check();
void main()
{ clrscr();
puts("GRAMMAR is E→ E+E \n E→E*E
\n E→ (E) \n E→ id ");
gets(a);
c= strlen(a);
strcpy(act, "SHIFT→");
puts("stack \t input \t action");
for(k=0; i=0; j<c; k++, i++, j++)
{ if(a[j] == 'i' && a[j+1] == 'd')
{ stk[i]= a[j];
stk[i+1]= a[j+1];
stk[i+2]= '\0';
```

```
        a[j] = '';
        a[j+1] = ' ';
printf("\n $ %s \t %s $ \t %s rd", 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(); } }
        getch(); }

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

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

RESULT:

The program is implemented & result obtained successfully

Output

GRAMMAR is E = E+E

E = E*E

E = (E)

E = id

input string is (id*id)+id

stack	input	action
$ C	id*id)+id$	SHIFT → symbols
$(id	*id)+id$	SHIFT → id
$(E	*id)+id$	REDUCE TO E
$(E*	id)+id$	SHIFT → symbols
$(E*id	)+id$	SHIFT → id
$(E*E	)+id$	REDUCE TO E
$(E	)+id$	REDUCE TO E
$EE)	+id$	SHIFT → symbols
$E	+id$	REDUCE TO E
$E+	id$	SHIFT → symbols
$E+id	$	SHIFT → id
$E+E	$	REDUCE TO E
$E	$	REDUCE TO E

Program: 6

OPERATOR PRECEDENCE PARSER

AIM:

To develop an operator precedence parser for a given language

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
void main()
{ char stack[20], ip[20], opt[10][10][1], ter[10];
int i, j, k, n, top=0, col, row;
clrscr();
for(i=0; i<10; i++)
{ stack[i]=NULL;
  ip[i]= NULL
for(j=0; j<10; j++)
 { opt[i][j][1]= NULL; }}
printf("Enter the number of terminals: \n");
scanf("%d", &n);
Printf("\n Enter the terminals: ");
 scanf("%d", &ter);
Printf("\n enter the table values");
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 (" Operator Precedence Table : \n");
  for (i=0; i<n; i++)
  printf("\t %c", ter[i]);
   Printf ("\n");
 for (i=0; i<n; i++)
  } printf ("\n %c", ter[i]);
   for (j=0; j<n; j++)
 } peintf ("\n %c", opt[i][j][0]); } }
 stack[top]="$";
 printf (" Enter input string ");
 scanf ("%s", ip);
  i=0;
printf("\n Stack \t\t\t Input string \t\t\t Action");
printf("\n %s\t\t\t\t %s\t\t\t\t", stack, ip);
 while (i < strlen(ip))
   } for (k=0; k<n; k++)
     } if (stack[top] == ter[k])
          col=k;
       if(ip[i] == ter[k])
          row=k; }
```

```
if (stack[top] == '$') && (ip[i] == '$'))
  { printf ("string is accepted");
        break; }
else if ((opt[col][row][0] == '<') || (opt[col][row]
[0] == '='))
   { stack[++top] = opt[col][row][0];
     Stack[++top] = ip[i];
     printf ("shift %c", ip[i]);
           i++ }
else.
  { if (opt[col][row][0] == '>')
    { while (stack[top] != '<')
        --top;
      top = top-1;
        printf ("Reduce"); }

else { printf ("\n string is not accepted");
        break; } }
printf ("\n");
 for (k=0; k<=top; k++)
   { printf ("%c", stack[k]); }
printf ("\t\t\t");
```

```
for ( k=1 ; k< strlen (ip) ; k++)
        printf ( "%c", ip[k] );
            printf("\t\t\t"); }

            getch();
}
```

RESULT

Program is implemented and successfully obtained the result.

Output

```
Enter the number of terminals.
Enter the terminals + * i $
Enter the table values:
Enter the value for + + : =
Enter the value for + * : <
Enter the value for + i : <
Enter the value for + $ : >
Enter the value for * + : >
Enter the value for * * : =
Enter the value for * i : <
Enter the value for * $ : >
Enter the value for i + : >
Enter the value for i * : >
Enter the value for i i : =
Enter the value for i $ : >
Enter the value for $ + : <
Enter the value for $ * : <
Enter the value for $ i : <
Enter the value for $ $ : =
```

Operator Precedence Table:

	+	*	i	$
+	=	<	<	>
*	>	=	<	>
i	>	>	=	>
$	<	<	<	=

Output (contd...)

Enter the input string i+i*i$

Stack	Input String	Action
$	i+i*i$	Shift i
$<i	+i*i$	Reduce
$	+i*i$	shift +
$<+	i*i$	Reduce
$	i*i$	Shift i
$<i	*i$	Reduce
$	*i$	shift *
$<*	i$	Reduce
$	i$	Shift i
$<i	$	Reduce
$	$	

String is accepted.

Program: 7

FIRST AND FOLLOW

AIM:

To implement a program to find the first and follow of any given Grammar.

PROGRAM:

```
#include <stdio.h>
# include <math.h>
# include <string.h>
# include <ctype.h>
# include <stdlib.h>
int n, m=0, p, i=0, j=0;
char a[10][10], f[10];
  void follow (char c);
  void first (char c);
  int main()
{ int i, z;
   char c, ch;
     clrscr();
printf ("Enter the number of productions: \n");
 scanf ("%d", &n);
printf ("Enter the productions: \n");
 for (i=0; i<n; i++)
```

```
scanf ("%s %c", &a[i], &ch);
do
{m=0;
   printf ("Enter the elements whose first & follow is
    to be found : \n");
scanf ("%c", &c);
first (c);
printf ("First of %c = { ", c);
  for (i=0; i<m; i++)
     printf ("%c", f[i]);
    printf ("} \n");
     strcpy (f, " ");
        m=0;
    follow(c);
    printf ("Follow of %c = { ", c);
       for (i=0; i<m; i++)
          printf ("%c", f[i]);
           printf ("} \n");
           printf ("continue (0/1) ?");
            scanf ("%d %c", &z, &ch); }

while (z==1);
   return(0);
    getch(); }
```

```
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]=='$')
        follow(a[k][0]==c)
       elseif(islower(a[k][2]))
          f[m++]=a[k][2];
       else
          first(a[k][2]); } }
}
void follow(char c)
{ if(a[0][0]==c)
      f[m++]='$';
  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];
             elseif(a[i][j+1]=='\0' &&
                    c!=a[i][0])
            follow(a[i][0]) }}}}
```

RESULT.

Programs is implemented & result is obtained successfully.

Output

Enter the number of productions. : 5

Enter the productions

S = AbCd

A = Cf

A = a

C = gE

E = h

Enter the element whose first & follow is to be found : : S

First of s : { g a }

Follow of s : { $ }

Continue (0/1)? 0

10/11/18

Program: 8

CONSTANT PROPOGATION

AIM:

Write a program to implement constant propogation.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<conio.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; }
  array[10];
  int n;
  void main()
  { clrscr();
   input();
  constant();
  output()
```

```
getch();}
void input()
{ int i;
printf("\n\n Enter the max number of expressions :");
scanf("%d",&n);
printf("\n enter 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; }
 printf (res1, "%d", res);
 arr[i].flag = 1;
 change (i, res1); } } }
void output ()
{ int i=0;
  printf ("\n Optimized 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);
        }
  }
```

RESULT:

The program is implemented and result obtained successfully

Output

Enter the num 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

Program: 9

INTERMEDIATE CODE GENERATOR

Aim:

To implement an intermediate code generator for simple expressions.

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int i=1, j=0, no=0, tempch=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]
int main()
{printf("\t\t Intermediate code generator \n\n");
 printf("\n Enter the expression \n");
 scanf("%s", str);
printf("the intermediate code: \t\t Expression \n");
```

```
find opr();
explore();
getch(); }

void find opr()
{ 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='-'; } }
```

```
vord explore()
{i=1;
while(k[i].op!='\0')
 { fleft(k[i].pos);
   fright(k[i].pos);
str[k[i].pos]=tempch--;
printf("\t %c := %s %c %s \t\t", str[k[i].pos],
        left, k[i].op, right);
for(j=0; j< strlen(str); j++)
   if(str[j]!='$')
      printf("%c", str[j]);
         printf("\n");
            i++;}
fright(-1);
if(no==0)
 fleft(strlen(str));
 printf("\t %s := %s", right, left);
   getch();}
printf("\t %s := %c", right, str[k[--i].pos]);
 getch();}
vord fleft(int x)
{ int w=0, flag=0;
     x--;
while(x!=-1 && 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 fight (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++; } }
```

RESULT:

Successfully implemented intermediate code generator for simple expressions.

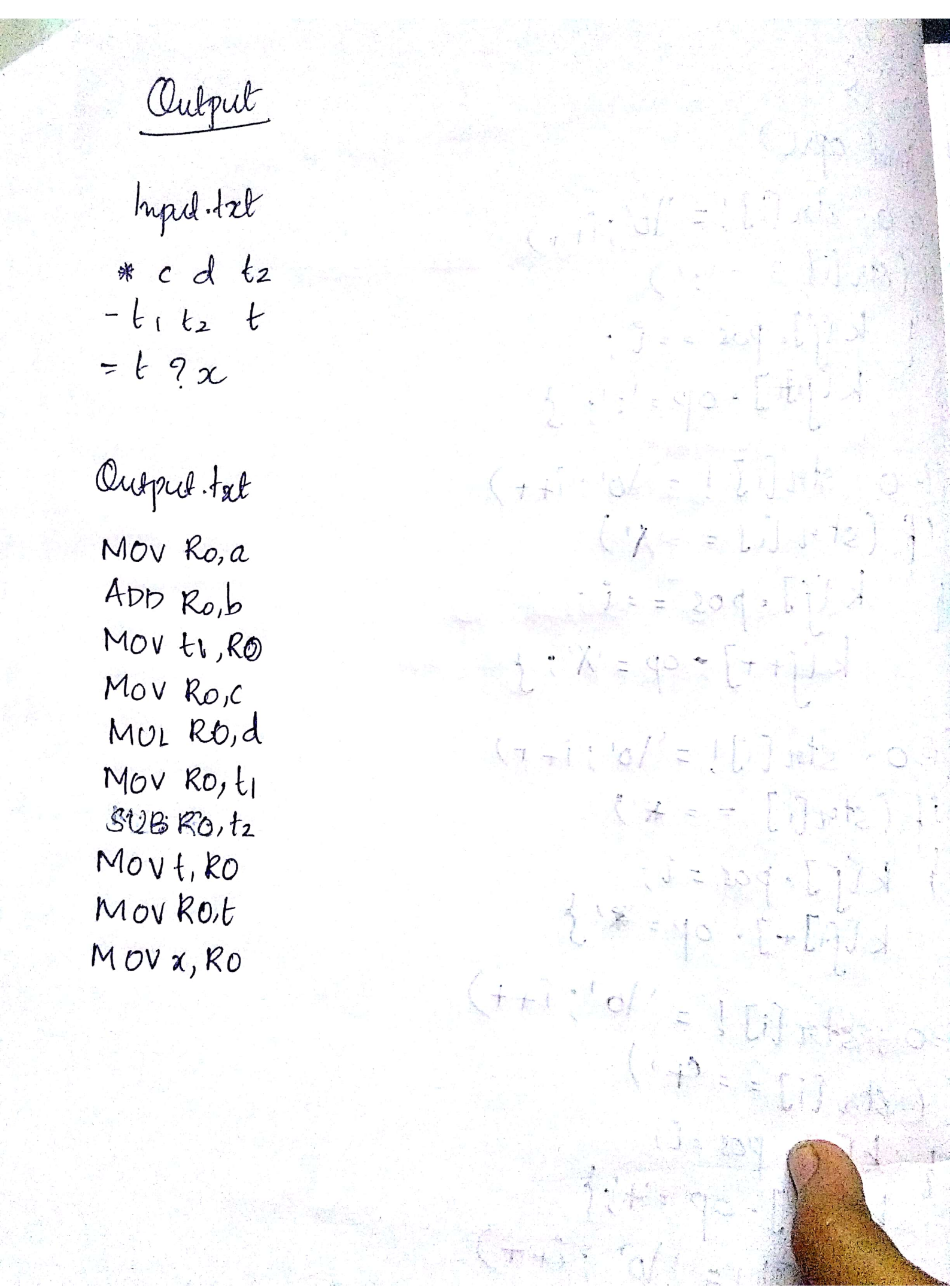
Output

Input.txt

```
* c d t2
- t1 t2 t
= t ? x
```

Output.txt

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



10/11/16

Program:10

NFA to DFA CONVERSION

AIM:

To implement a program that would convert any given NFA to DFA after finding ε-closure if any.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX_LEN 100

char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int ff=0;

struct DFA
{ char *states;
  int count; } dfa;

int last_index=0;
FILE *fp;
int symbols;
void reset(int a[], int size)
{ int i;
    for(i=0; i<size; i++)
        { a[i]=0; } }
```

```
void check (int ar[], char s[])
  { int i, j;
int len = strlen(s);
for (i=0; i< len ; i++)
  {  j = ((int) (s[i]) - 65);
        ar[j]++; } }
void state (int ar[], int size, char s[])
 { int j, k=0 ;
  for (j=0; j<size ; j++)
        { if (ar[j] !=0)
          s[k++] = (char)(65 + j); }
     s[k] = '\0'; }
int closure ( int ar[], int size)
   { int i;
     for (i=0; i<size ; i++)
         { if ( ar[i] == 1)
             return i ; }
    return (100); }
int indexing (struct DFA * dfa)
    { int i;
     for (i=0 ; i< last_index ; i++)
        { if (dfa[i].count == 0)
              return 1; }
```

```
return -1; }
void Display_closure(int states, int closure_ar[],
        char* closure_table[], char* NFA_TABLE[
symbols+1], char* DFA_TABLE[][symbols])
  { int i;
 for(i=0; i<states; i++)
   { reset( closure_ar, states);
     closure_ar[i]=2;
 if (strcmp (&NFA_TABLE[i][symbols], "  ")!=0)
{ strcpy (buffer, &NFA_TABLE[i][symbols]);
  check (closure_ar, buffer);
  int j = closure (closure_ar, states);
while (j!=100)
  { if ( strcmp( &NFA_TABLE[j][symbols], " ")!=0)
      { strcpy (buffer; &NFA_TABLE[j] symbols]);
    check (closure_ar, buffer); }
    closure_ar[j]++;
      j = closure (closure_ar, states); } }
printf ("\n e-closure (%c): \t", (char)(65+i));
    bzero(void *) buffer, MAX_LEN);
      state (closure_ar, states, buffer);
      strcpy (&closure_table[i], buffer);
      Printf ("%s\n", &closure_table[i]); } }
```

```
int new_states(struct DFA *dfa, char s[])
{ int i;
   for (i=0; i<last_index ; i++)
   { if (strcmp (&dfa[i].states, s)==0)
        return 0; }
strcpy (&dfa [last_index++].states, s);
dfa [last_index-1].count =0;
     return 1; }
void trans(char s[], int nn, char *c [sn_t[], int st,
char* NFA [] [symbols +1], char tB[]))
{ int len = strlen (s);
  int i, j, k, g;
  int arr [st];
  int sz;
 reset (arr, st);
char temp [MAX_LEN], temp2 [MAX_LEN];
  char* buff;
for (i=0; i< len; i++)
{ j = ((int)s[i] + 65));
    strcpy (temp, &NFA [j] [m]);
    if (strcmp (temp, " ") != 0)
       { sz = strlen (temp);
            g=0;
       while (g < sz)
         { k = ((int) (temp[g] - 65));
```

```
strcpy(temp2, &clos_t[k]);
check(arr, temp2);
        gt++; } }
bzero( void * )temp, MAX_LEN);
state( arrr, st, temp );
if (temp[0] != '\0')
  { strcpy(TB, temp); }
else
    { strcpy (TB, " "); }
void Display_DFA (int last_index, struct DFA * dfa
states, char *DFA_TABLE[] [symbols])
{ int i, j;
printf("\n\n * * * * * * ....\n\n");
printf ("\t\t DFA TRANSITION TABLE \t\t\n\n");
printf ("\n STATES OF DFA \t\t");
for (i=1 ; i< last_index ; i++)
    printf( "%s, &dfa_states[i].states);
    printf ("\n");
    printf("\n GIVEN SYMBOLS FOR DFA : \t");

for (i=0 ; i<symbols ; i++)
      printf ("%d ", i);

for (i=1; i< last_index ; i++)
  printf ( " %s, &dfa_states[i].states);
    printf ("\n");
```

```
printf("\n GIVEN SYMBOL FOR DFA: \t");
for (i=0 ; i<symbols; i++)
printf("%d , " , i);
printf("\n \n");
printf("STATES \n");
for (i=0; i< symbols ; i++)
    printf("%d.\t", i);
   printf("\n");
printf(" . . . . . . . . . . \n ");
   for (i=0; i<gg ; i++)
    { printf("\n \t", &dfa_states [i+1]. states);
      for (j=0; j< symbals; j++)
        { printf(" %s \t", & DFA_TABLE [i][j]); }
      printf("\n");    }}
int main ( );
{ int i, j. states;
  char T_but [MAX_LEN];
struct DFA "dfa_states" = malloc [MAX_LEN * (is size of
(dfa)]};
   states = 0, symbales = 2;
printf("\n NFA STATES \n \t");
for (i=0; i<states ; i++)
printf("%c, ", ', (char)(65+i));
```

```
printf("\n");
printf("\n GIVEN SYMBOLS FOR NFA :\t");
for (i=0; i< symbols; i++)
 printf("%d", i);
 printf(" eps");
 printf("\n\n");
char * DFA_TABLE [MAX_LEN] [symbols];
strcpy (&NFA_TABLE [0][0], "FC");
strcpy (&NFA_TABLE [0][1], "=");
strcpy (&NFA_TABLE [0][2], "BF");
strcpy (&NFA_TABLE [1][0], "_");
strcpy (&NFA_TABLE [1][1], "C");
strcpy (&NFA_TABLE [1][2], "=");
strcpy (&NFA_TABLE [1][2], "_");
strcpy (&NFA_TABLE [2][0], "_");
strcpy (&NFA_TABLE [2][1], "D");
strcpy (&NFA_TABLE [2][2], "E");
strcpy (&NFA_TABLE [3][0], "A");
strcpy (&NFA_TABLE [3][1], "_");
strcpy (&NFA_TABLE [3][2], "A");
strcpy (&NFA_TABLE [4][0], "_");
strcpy (&NFA_TABLE [4][1], "BF");
strcpy (&NFA_TABLE [4][2], "_");
strcpy (&NFA_TABLE [5][0], "_");
strcpy (&NFA_TABLE [5][1], "_");
strcpy (&NFA_TABLE [5][2], "=");
```

```
printf("\n NFA STATE TRANSITION TABLE \n\n");
  printf("STATES \t");
printf("\n ..... \n");
 for(i=0;i<states;i++)
   { printf("\n%c \t",(char)(65+i));
  for(j=0; j<=symbols;j++)
   { printf(" %s \t',&NFA_TABLES[i][j]); }
    printf("\n"); }
int_closure_au[states];
char * closure = table[states];
Display_closure(states, closure_au, closure_table,
NFA_TABLE, DFA_TABLE);
strcpy(&dfa_states[last_index++].states," ");
dfa_states(last_index -1] + count = 1;
bzero((void *) buffer, MAX_LEN);
strcpy(buffer, &closure_table[0]);
strcpy(&dfa-states[last_index++].states, buffer);
int sm = 1, ind = 1;
int start_index = 1;
while (ind.i = -1)
 { dfa_states[start_index].comt = 1;
    sm = 0;
  for(i=0; i<symbols;i++)
```

```
trans(buffer, p, closure_table, states, NFA_TABLE, T_buf);
strcpy(&DFA_TABLE[gg][i], Tbuf);
sm = sm + new_states(dfa_states, T_Buf); }

    ind = indexing(dfa_states);
    if(ind != -1)
     strcpy(buffer, &dfa_states[++start_index].
                                        states);
    gg++; }.
Display_DFA(last_index, dfa_states, DFA_TABLE);
return 0;
```

RESULT

The program is implemented & result is obtained successfully

Output

STATES	0	1	eps
A	FC	-	BF
B	-	C	-
C	-	-	D
D	E	A	-
E	A	-	BF
F	-	-	-

e-closure (A): ABF
e-closure (B) : B
e-closure (C) : CD
e-closure (D) : D
e-closure (E) : BEF
e-closure (F) : F

* * * * * * * * * * * * *

DFA TRANSITION TABLE

STATES OF DFA : ABF, CDF, CD, BEF

GIVEN SYMBOLS FOR DFA:

STATES	0	1
ABF	CDF	CD
CDF	BEF	ABF
CD	BEF	ABF
BEF	ABF	CD