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

char mnemonic[5][3][10]=
{
 {"1","START","AD"},
 {"2","EQU","AD"},
 {"3","ORIGIN","AD"},
 {"4","LTORG","AD"},
 {"5","END","AD"}
};

char
symbol_table[10][2][10]={""};
char lit_table[10][2][10]={""};
int pool_table[10][2]={0};
int s1=0,l1=0,p1=0,l_cnt=0;

int main()
{
 int i=0,j;
 int loc=0;
 int start=0,equ=0,origin=0,ltorg=0,end=0;
char *field,record[200],const1[10];  char
symb_loc[25];
```

```
int n;

char op[20];
FILE *fr;

pool_table[0][0]=1;

pool_table[0][1]=0;



fr=fopen("ass_4.txt","r");


while(fgets(record,200,fr))

{

        int fcnt=0; // field counter

        loc++;

        printf("\n");

        field=strtok(record," ");


        while(field!=NULL)

        {

         fcnt++;

         printf("%s \t",field);


         if(fcnt==1)

         {

                 if(strcmp(field,"$")!=0) // if field is not $ then label exist

                 {

                  strcpy(symbol_table[s1][0],field);
         strcpy(op,field);
         sprintf(symb_loc,"%d",loc);
         strcpy(symbol_table[s1][1],symb_loc);
         s1++;
        }//if not '$'
}//if fcnt=1

if(fcnt==2)
{
        int found=0;
        int index;
        for(i=0;i<5;i++)
        {
         if(strcmp(mnemonic[i][1],field)==0)
        {
                found=1;
                index=i;
                break;
         }
        }
        if(found==1)
        {
         char class1[10]="";
         char mnemonic1[10]="";
         strcpy(class1,mnemonic[index][2]);
        strcpy(mnemonic1,mnemonic[index][1]);
```

```
if(strcmp(class1,"AD")==0)
 {
        if(strcmp(mnemonic1,"START")==0)
        {
         start=1;
        }
        if(strcmp(mnemonic1,"EQU")==0)
        {
         equ=1;
         loc--;
        }
        if(strcmp(mnemonic1,"ORIGIN")==0
        ) {
         origin=1;
         loc--;
        }
        if(strcmp(mnemonic1,"LTORG")==0)
        {
         ltorg=1;
         loc--;
         break;
        }
        if(strcmp(mnemonic1,"END")==0)
        {
         end=1;
         loc--;
```

```
                }
            }
           }
}//if cnt=2
if(fcnt==3)
{
           if(start==1)
           {
            strcpy(const1,field);
            loc=atoi(const1);
            loc=loc-1;
            start=0;
           }
           if(equ==1)
           {
            char index_of_symbol[20];
            int find_index=0;
            for(i=0;i<s1;i++)
            {
                      if(strcmp(symbol_table[i][0],field)==0)
                      {
                          if(strcmp(symbol_table[i][1]," ")!=0)
                       {
                               find_index=1;
                               strcpy(index_of_symbol,symbol_table[i][1]);
                               break;
```

```
          }
         }
 }//for complete
 if(find_index==1)
 {
         for(i=0;i<s1;i++)
         {
             if(strcmp(symbol_table[i][0],op)==0)
          {
                   strcpy(symbol_table[i][1],index_of_symbol)
                   ; break;
          }
         }//for complete
         find_index=0;
 }//find_index =1 comlete
 equ=0;
} //if equ=1 complete
if(origin==1)
{
 char origin_str[20];
 char *p;
 char index_of_symbol[20];
 int find_index=0;
 strcpy(origin_str,field);
 p = strtok(origin_str, "+-");
 for(i=0;i<s1;i++)
```

```
{
        if(strcmp(symbol_table[i][0],p)==0)
        {
           if(strcmp(symbol_table[i][1]," ")!=0)
         {
                find_index=1;
                strcpy(index_of_symbol,symbol_table[i][1]);
                break;
         }
        }
} //for complete
if(find_index==1)
{
        for(i=0;i<s1;i++)
        {
         if(strcmp(symbol_table[i][0],op)==0)
        {
                char *ptr = strchr(field, '+');
                p= (strtok(NULL, "+ -")) ;
                if(ptr)
                 loc= atoi(index_of_symbol)+atoi(p);
                else
                   loc=atoi(index_of_symbol)-atoi(p);
                sprintf(symb_loc,"%d",loc);
                break;
         }
```

```
          } // for complete
          find_index=0;
 }//find_index =1 comlete
 origin=0;
 loc--;
}
if(ltorg==1)
{
 l_cnt++;
 if(l_cnt>l1)
 {
          ltorg=0;
          p1++;
          pool_table[p1][0]=l_cnt;
          pool_table[p1][1]=0;
          l_cnt--;
 }
 else
 {
          char *ptr;
          ptr=strchr(field,'=');
          if(ptr)
          {
           for(i=0;i<l1;i++)
           {
                    if(strcmp(lit_table[i][0],field)==0)
                    {
                       if(strcmp(lit_table[i][1]," ")==0)
                     {
                            sprintf(symb_loc,"%d",loc);
                            strcpy(lit_table[i][1],symb_loc);
                            pool_table[p1][1]=pool_table[p1][1]+1;
                     }
                    }
            }
           }
     }
    }
    if(end==1)
    {
     char *ptr;
     ptr=strchr(field,'=');
     if(ptr)
     {
           for(i=0;i<l1;i++)
           {
              if(strcmp(lit_table[i][0],field)==0)
            {
                    if(strcmp(lit_table[i][1]," ")==0)
                    {
                       sprintf(symb_loc,"%d",loc);
                       strcpy(lit_table[i][1],symb_loc);
```

```
                        pool_table[p1][1]=pool_table[p1][1]+1;
                      }
                  }
                 }
          }
         }
 }//if fcnt=3

 if(fcnt==4) // will write literals to littable /////  {
 // int complete=0;
 // int lable_exist=0;
         char *ptr;
         ptr=strchr(field,'=');
         if(ptr)
         {
          strcpy(lit_table[l1][0],field);
         strcpy(lit_table[l1][1]," ");
          l1++;
          // complete=1;
         }
 }
 field=strtok(NULL," ");
 }//while for all fields(tokens)
}//eof while
fclose(fr);
```

```
printf("\n \n \n Symbol table\n");

for(i=0;i<s1;i++)
{
 printf("\n");
 for(j=0;j<2;j++)
 {
         printf("%s
\t",symbol_table[i][j]);  }
}
printf("\n \n \n Literal
table\n"); for(i=0;i<l1;i++)
{
 printf("\n");
 for(j=0;j<2;j++)
 {
         printf("%s \t",lit_table[i][j]);
 }
}
printf("\n \n \n Pool table\n");

for(i=0;i<=p1;i++)
{
 printf("\n");
 for(j=0;j<2;j++)
```

```
           {
 printf("%d \t",pool_table[i][j]);  }
          }
          getch();
          return 0;
}
```

```
# START 101
# MOVEM AREG A
LOOP MOVER AREG A
# MOVER CREG B
# BC ANY NEXT
NEXT SUB AREG A
LAST STOP
# BC LT LOOP
A DS 1
B DS 1
BACK EQU LOOP
# END
```

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


char mnemonic[18][3][10] = {

 {"00", "STOP", "IS"},

 {"01", "ADD", "IS"},

 {"02", "SUB", "IS"},

 {"03", "MULT", "IS"},

 {"04", "MOVER", "IS"},

 {"05", "MOVEM", "IS"},

 {"06", "COMP", "IS"},

 {"07", "BC", "IS"},

 {"08", "DIV", "IS"},

 {"09", "READ", "IS"},

 {"10", "PRINT", "IS"},

 {"11", "DS", "DL"},

 {"12", "DC", "DL"},

 {"13", "START", "AD"},

 {"14", "END", "AD"},

 {"15", "ORIGIN", "AD"},

 {"16", "EQU", "AD"},

 {"17", "LTORG", "AD"}};


char register1[4][10] = {"AREG", "BREG", "CREG", "DREG"};

char condition[6][10] = {"LT", "LE", "GT", "GE", "EQ", "ANY"};

char symbol_table[10][2][10] = {{"LOOP", "102"}, {"NEXT", "107"}, {"LAST", "113"},
{"A", "115"}, {"B", "116"}};

char lit_table[10][2][10] = {{"=5", "108"}, {"=1", "109"}, {"=1", "111"}, {"=2",
"117"}}; int s1 = 10, l1 = 10, p1 = 10, l_cnt = 0, blank_cnt = 0, remain1 = 0;
int main() {
FILE *fr, *fw;
int start = 0, loc = 0, equ = 0, ltorg = 0, end = 0, dl = 0, is = 0;
char *field, record[200], const1[10], left_op[20];
int pool_ptr = 0, i = 0;

fr = fopen("ass_ic.txt", "r");
fw = fopen("icnew.txt", "w");

while (!feof(fr)) {
           int fcnt = 0;
           int found = 0, index;
           loc++;
           if (loc != 1) {
fprintf(fw, "%d%s", loc, "+");
 }
fgets(record, 200, fr);
field = strtok(record, " ");
while (field != NULL) {
fcnt++;
if (fcnt == 2) {
if (ltorg == 1 && strcmp(field, "#") == 0) {
for (i = 0; i < 18; i++) {
if (strcmp(mnemonic[i][1], "DC") == 0) {  fprintf(fw, "%s%s%s\t",
"(DL,", mnemonic[i][0], ")");  }
 }
 }
if (end == 1 && strcmp(field, "#") == 0) {
```

```
for (i = 0; i < 18; i++) {
if (strcmp(mnemonic[i][1], "DC") == 0) {  fprintf(fw, "%s%s%s\t",
"(DL,", mnemonic[i][0], ")");  }
}
}
for (i = 0; i < 18; i++) {
if (strcmp(mnemonic[i][1], field) == 0) {
found = 1;
index = i;
break;
}
}
if (found == 1) {
char class1[10] = "", mnemonic1[10] = "", op_code[10] = "";
strcpy(class1, mnemonic[index][2]);
strcpy(mnemonic1, mnemonic[index][1]);
strcpy(op_code, mnemonic[index][0]);
if (strcmp(class1, "AD") == 0) {
if (strcmp(mnemonic1, "START") == 0) {  start = 1;
fprintf(fw, "%s%s%s", "(AD,", op_code, ")");  }
if (strcmp(mnemonic1, "EQU") == 0) {
equ = 1;
fprintf(fw, "%s%s%s", "(AD,", op_code, ")");  loc--;
}
if (strcmp(mnemonic1, "LTORG") == 0) {  ltorg = 1;
fprintf(fw, "%s%s%s", "(AD,", op_code, ")");
loc--;
pool_ptr++;
break;
}
if (strcmp(mnemonic1, "END") == 0) {  end = 1;
fprintf(fw, "%s%s%s", "(AD,", op_code, ")");  loc--;
break;
}
} else if (strcmp(class1, "DL") == 0) {  dl = 1;
fprintf(fw, "%s%s%s\t", "( DL,", op_code, ")");  } else if
(strcmp(class1, "IS") == 0) {  is = 1;
fprintf(fw, "%s%s%s\t", "(IS,", op_code, ")");  }
}
}
if (fcnt == 3) {
if (dl == 1 && equ != 1 && end != 1) {  fprintf(fw,
"%s%s%s\t", "(C,", field, ")");  }
if (is == 1) {
for (i = 0; i < 4; i++) {
if (strcmp(register1[i], field) == 0) {  fprintf(fw,
"%s%d%s\t", "(", i + 1, ")");  }
}
for (i = 0; i < 6; i++) {
if (strcmp(condition[i], field) == 0) {  fprintf(fw,
"%s%d%s\t", "(", i + 1, ")");  }
}
}
if (start == 1) {
strcpy(const1, field);
loc = atoi(const1);
fprintf(fw, "%s%d%s\t", "(C,", loc, ")");  loc = loc -
```

```
1;
 start = 0;
 }
 if (equ == 1) {
 for (i = 0; i < s1; i++) {
 if (strcmp(symbol_table[i][0], field) == 0) {  fprintf(fw,
"%s%d%s", "(S,", i + 1, ")");  equ = 0;
 break;
 }
 }
 }
 if (ltorg == 1) {
 char *ptr, *s;
 ptr = strchr(field, '=');
 if (ptr) {
 s = strtok(field, "=");
 fprintf(fw, "%s%s%s \t\t ", "(C,", s, ")");  } else {
 ltorg = 0;
 }
 }
 if (end == 1) {
 char *ptr, *s;
 ptr = strchr(field, '=');
 if (ptr) {
 s = strtok(field, "=");
 fprintf(fw, "%s%s%s \t\t", "(C,", s, ")");  } else {
 end = 0;
 }
 }
 }
 if (fcnt == 4) {
 char *ptr;
 ptr = strchr(field, '=');
 if (ptr) {
 int get_lit;
 for (i = 0; i < l1; i++) { // Iterate over lit_table directly  if
(strcmp(lit_table[i][0], field) == 0) {  get_lit = i;
 fprintf(fw, "%s%d%s", "(L,", (get_lit + 1), ")");  break; // Once
found, exit loop
 }
 }
 }
 else {
 int complete = 0;
 for (i = 0; i < s1; i++) {
 if (strcmp(symbol_table[i][0], field) == 0) {  fprintf(fw,
"%s%d%s", "(S,", i + 1, ")");
 complete = 1;
 break;
 }
 }
 if (complete == 0) {
 // Handle undefined symbols here  }
 }
 }
 field = strtok(NULL, " ");
 if (fcnt != 1) {
```

```
 fprintf(fw, "\n");
 }
 }
 }
 fclose(fr);
 fclose(fw);
 return 0;
}
```

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
char mnemonic[3][3][10]=
{
 {"1","START","AD"},
 {"2","EQU","AD"}
};
char
symbol_table[10][2][10]={""}; int
s1=0;
int main()
{
 int i=0,j=0;
 int loc=0;
 int start=0,equ=0,origin=0;  char
*field,record[200],const1[10];  char
symb_loc[25];
 int n;
 char op[20];
 FILE *fr;
 clrscr();

 fr=fopen("ass_2.txt","r");
 while(!feof(fr))
 {
 int fcnt=0;
 loc++;
```

```
fgets(record,200,fr);

field=strtok(record," ");
while(field!=NULL)

{

fcnt++;

printf("%s \t",field);


if(fcnt==1)

{

if(strcmp(field,"$")!=0)

{

            strcpy(symbol_table[s1][0],field);

            strcpy(op,field);

            sprintf(symb_loc,"%d",loc);

            strcpy(symbol_table[s1][1],symb_loc)

            ; s1++;

}

}

if(fcnt==2)

{

int found=0;

int index;

for(i=0;i<3;i++)

{

             if(strcmp(mnemonic[i][1],field)==0

             ) {

             found=1;

             index=i;

             break;

             }
}

if(found==1)
{
            char class1[10]="";

            char mnemonic1[10]="";

            strcpy(class1,mnemonic[index][2]);

            strcpy(mnemonic1,mnemonic[index][1])

            ; if(strcmp(class1,"AD")==0)

            {

            if(strcmp(mnemonic1,"START")==0)

            {

            start=1;

            }

            if(strcmp(mnemonic1,"EQU")==0)

            {

            equ=1;

            loc--;

            }

            }

}

}

if(fcnt==3)

{

           if(start==1)

           {

            strcpy(const1,field);

            loc=atoi(const1);

            loc=loc-1;

            start=0;

           }
```

```
if(equ==1)

{

 char index_of_symbol[20];
 int find_index=0;

 for(i=0;i<s1;i++)

 {

 if(strcmp(symbol_table[i][0],field)==0)

{

 if(strcmp(symbol_table[i][1]," ")!=0)

 {

 find_index=1;

 strcpy(index_of_symbol,symbol_table[i][1]);

break;

 }

 }

 }

 if(find_index==1)

 {

 for(i=0;i<s1;i++)

 {

             if(strcmp(symbol_table[i][0],op)==0)

             {

              strcpy(symbol_table[i][1],index_of_symbol)

              ; break;

             }

 }

 find_index=0;

 }

 equ=0;

 }
```

```
        }

        field=strtok(NULL," ");

        }

        }
        fclose(fr);


        printf("\n \n \n symbol table\n");


        for(i=0;i<s1;i++)

        {

        printf("\n");

        for(j=0;j<2;j++)

        {

                printf("%s \t",symbol_table[i][j]);

        }

        }

        getch();

        return 0;

}
```

```
#include <stdio.h>

#include <conio.h>

#include <ctype.h>

#include <string.h>

#include <stdlib.h>

char mnemonic[1][3][10]={

                    {"1","START","AD"}};

char symbol_table[10][2][10]={""};

int s1=0;


int main()

{

          int i=0,loc=0,j;

          char

          *field,record[200],const1[10],symbol_loc[25]; FILE

          *fp;

clrscr();

          fp=fopen("assem.txt","r");

          while(!feof(fp))

          {

                    int fcnt=0;

                    int start=0;

                    loc++;

                    fgets(record,200,fp);

                    field=strtok(record," ");

                    while(field != NULL)

                    {

                              fcnt++;

                              printf("%s \t",field);

                              if(fcnt==1)
```

```
{
        if(strcmp(field,"#")!=0)
        {
                strcpy(symbol_table[s1][0],field);
                sprintf(symbol_loc,"%d" ,loc);
                strcpy(symbol_table[s1][1],symbol_loc);
                s1++;
        }
}
if(fcnt==2)
{
        int found=0;
        int index;
        for(i=0;i<1;i++)
        {
                if(strcmp(mnemonic[i][1],field)==0)
                {
                        found=1;
                        index=i;
                        break;
                }
        }
        if(found==1)
        {
                char class1[10]="";
                char mnemonic1[10]="";
                strcpy(class1,mnemonic[index][2]);
                strcpy(mnemonic1,mnemonic[index][1]);
                if(strcmp(class1,"AD")==0)
                {
```

```
                              if(strcmp(mnemonic1,"START")==0)
                              {
                                        start=1;
                              }
                    }
          }
     }
     if(fcnt==3)
     {
          if(start==1)
          {
               strcpy(const1,field);
               loc=atoi(const1);
               loc=loc-1;
               start=0;
          }
     }
     field=strtok(NULL," ");
  }
}
fclose(fp);
printf("\n\n Symbol Table: ");
for(i=0;i<s1;i++)
{
     printf("\n");
     for(j=0;j<2;j++)
     {
          printf("%s \t",symbol_table[i][j]);
     }
}
```

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

DFA ID

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


char stat_table[3][3][10] = {

 {"stat", "letter", "digit"},

 {"start", "ID", "ID"},

 {"ID", "ID", "ID"}

};


int main() {

 char input[20], column_stat[10], next_stat[10], current_stat[10];

char ch;

 int i, len, row, column, error = 0;


 clrscr(); // Clear the screen

 printf("Enter Identifier: ");

 gets(input); // Avoid using gets() in production code due to security
vulnerabilities

 len = strlen(input);

 printf("OUTPUT:\n");

 strcpy(current_stat, "start");


 for (i = 0; i < len; i++) {

 ch = input[i];
 if (isalpha(ch)) {
```

```
strcpy(column_stat, "letter");
} else if (isdigit(ch)) {
strcpy(column_stat, "digit");
} else {
strcpy(column_stat, "invalid");
error++;
}

if (error == 0) {
for (row = 1; row < 3; row++) {
if (strcmp(stat_table[row][0], current_stat) == 0) {  break;
}
}

for (column = 1; column < 2; column++) { // Changed the loop range to  2
since there are only 2 columns now
if (strcmp(stat_table[0][column], column_stat) == 0) {  break;
}
}

strcpy(next_stat, stat_table[row][column]);
printf("%s %c %s\n", current_stat, ch, next_stat);
} else {
printf("%s %c %s\n", current_stat, ch, "invalid");
printf("ERROR: Invalid Character\n");
break;
}
```

```
 strcpy(current_stat, next_stat);
 }

 if (error == 0 && strcmp(current_stat, "ID") == 0) {
printf("Valid Identifier\n");
 } else {
 printf("Invalid Identifier\n");
 }

 getch(); // Wait for a key press
 return 0;
}
```

DFA REAL

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

char stat_table[6][4][10] = {
 {"stat", "letter", "digit", "."},
 {"start", "id", "int", "error"},
 {"id", "id", "id", "error"},
 {"int", "error", "int", "s"},
 {"s", "error", "real", "error"},
 {"real", "error", "real", "error"},
};

int main() {
 char input[20], column_stat[10], current_stat[10], next_stat[10];
char ch, choice;
 int error, i, c, r, len;

 do {
        printf("Enter identifier: ");
        scanf("%s", input);
 len = strlen(input);
 strcpy(current_stat, "start");
 error = 0; // Reset the error before each input
```

```
for (i = 0; i < len && !error; i++) {
ch = input[i];


if (isalpha(ch)) {
strcpy(column_stat, "letter");
} else if (isdigit(ch)) {
strcpy(column_stat, "digit");
} else if (ch == '.') {
strcpy(column_stat, ".");
} else {
strcpy(next_stat, "error");
error = 1;
break; // Exit the loop immediately when encountering an error  }


for (r = 1; r < 6; r++) {
if (strcmp(stat_table[r][0], current_stat) == 0) {  for (c =
1; c < 4; c++) {
if (strcmp(stat_table[0][c], column_stat) == 0) {
strcpy(next_stat, stat_table[r][c]);
break;
}
}


if (strcmp(next_stat, "error") == 0) {
error = 1; // Set error to 1 to break out of the loop   break;
}
                    printf("%s %c %s\n", current_stat, ch, next_stat);
strcpy(current_stat, next_stat);
break;
}
}
}

if (error) {
printf("\nInvalid Token");
} else {
printf("\nValid");

if (strcmp(current_stat, "id") == 0) {
printf("\nIt is an identifier");
} else if (strcmp(current_stat, "int") == 0) {
printf("\nIt is an integer");
} else if (strcmp(current_stat, "real") == 0) {
printf("\nIt is a real");
}
}

printf("\n\nDo you want to continue? (enter 'y' for yes and 'n' for no): ");
scanf(" %c", &choice);
} while (choice != 'n');

return 0;
```

```
}
```

DFA INT

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

char stat_table[3][3][10] = {
 {"stat", "digit", "letter"},
 {"start", "INT", "error"},
 {"INT", "INT", "error"}};

int main() {
 int i, len, row, column, r, c, flag = 0, error = 0;
 char input[20], column_stat[10], next_stat[10], current_stat[10];
char ch;

 clrscr(); // Clear the screen

 printf("Enter Integer:");
 gets(input);

 flag = 0;
 error = 0;
 len = strlen(input);

 printf("OUTPUT:\n");
```

```
strcpy(current_stat, "start");
for (i = 0; i < len; i++) {
        for (r = 1; r < 3; r++) {
         if (strcmp(stat_table[r][0], current_stat) == 0) {
                row = r;
                break;
         }
        }

        ch = input[i];

        if (!isdigit(ch)) {
         strcpy(next_stat, "error");
         error++;
        } else {
         strcpy(column_stat, "digit");
        }

        if (error == 0) {
         for (c = 1; c < 3; c++) {
                if (strcmp(stat_table[0][c], column_stat) == 0)
                {  column = c;
                 break;
                }
         }
```

```
            strcpy(next_stat, stat_table[row][column]);
            printf("%s %c %s\n", current_stat, ch, next_stat);
           } else {
            printf("%s %c %s\n", current_stat, ch, next_stat);
           printf("ERROR: Invalid Input\n");
            break;
           }

    strcpy(current_stat, next_stat);
    }

    if (error == 0 && strcmp(current_stat, "INT") == 0) {
   printf("Valid Integer\n");
    } else {
    printf("Invalid Integer\n");
    }

    getch(); // Wait for a key press
    return 0;
   }
```

MACRO-I

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

struct mnt1 {
   char name[15];
   int npp, nkp, nev, mdtp, kpdtp, sstp;
} mnt;

char pntab[5][15]; // PNTAB (Parameter Name
Table) char mdt[50][50]; // MDT (Macro Definition
Table)

void main() {
   FILE *f1;
   char ch[80], *p;
   int i, pntbptr = 0, mdtptr = 1;

   // Initialize MNT values
   mnt.npp = mnt.nkp = mnt.nev = 0;
   mnt.mdtp = 1;
   mnt.kpdtp = 0;
   mnt.sstp = 0;

   clrscr();
   f1 = fopen("MAC.txt", "r");
   fgets(ch, 80, f1);
   p = strtok(ch, " ");
   if (p[strlen(p) - 1] == '\n')
```

```
  p[strlen(p) - 1] = '\0';
if (strcmp(p, "MACRO") ==

  0) { // Read Macro Name

  fgets(ch, 80, f1);

  p = strtok(ch, " ,\n");

  strcpy(mnt.name, p);


  // Read Macro Parameters

  p = strtok(NULL, " ,\n");

  while (p) {

    strcpy(pntab[pntbptr], p);

    pntbptr++;

    mnt.npp++;

    p = strtok(NULL, " ,\n");

  }


  // Process Macro Body

  while (fgets(ch, 80, f1)) {

    if  (strcmp(ch,  "MEND\n")  ==

      0)     {     strcpy(mdt[mdtptr],

      "MEND"); mdtptr++;

      break;

    }


    p = strtok(ch, " ,\n");

    strcpy(mdt[mdtptr], "");

    while (p) {

      int found = 0;

      for (i = 0; i < pntbptr; i++) {

        if (strcmp(p, pntab[i]) == 0) {
            char temp[10];

            sprintf(temp, "(P,%d)", i + 1);
            strcat(mdt[mdtptr], temp);

            found = 1;

            break;

          }

        }

        if (!found) {

          strcat(mdt[mdtptr], p);

        }

        strcat(mdt[mdtptr], " ");

        p = strtok(NULL, " ,\n");

    }

    mdtptr++;

}


// Print PNTAB

printf("\t\tPNTAB\n");

printf("-------------------\n");

for (i = 0; i < pntbptr; i++) {

    printf("%d | %s\n", i + 1, pntab[i]);

}

printf("-------------------\n\n");


// Print MNT

printf("\t\t MNT\n");

printf("-------------------------------------------\n");

printf(" Name #pp #kp #ev MDTP KPDTP SSTP\n");

printf("%s %d %d %d %d %d %d",

      mnt.name, mnt.npp, mnt.nkp, mnt.nev, mnt.mdtp, mnt.kpdtp,
    mnt.sstp); printf("\n------------------------------------------\n");


    // Print MDT
    printf("\n\t\tMDT\n");

    printf("------------------------\n");

    for (i = 1; i < mdtptr; i++) {

      printf("%d | %s\n", i,

    mdt[i]); }

    printf("------------------------\n");

  } else {

    printf("Invalid

  source...\n"); }


  fclose(f1);

  getch();

}
```

MACRO-II

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

char mdt[50][50] = {
    "MOVER (P,3) (P,1)",
    "ADD (P,3) (P,2)",
    "MOVEM (P,3) (P,1)",
    "MEND"
};

char pntab[5][15] = {"", "&MEM_VAL", "&INCR_VAL",
"&REG"}; char actual_params[5][15]; // ANTAB

void main() {
    char call[] = "INCR A, B, AREG";
    char *token;
    int i;
    clrscr();
    // Parse macro call and populate ANTAB
    token = strtok(call, " ,");
    token = strtok(NULL, " ,"); // Skip macro name

    i = 0;
    while (token != NULL) {
    strcpy(actual_params[++i], token);
```

```
token = strtok(NULL, " ,");
}

// Print ANTAB
printf("\tANTAB (Actual Name Table)\n");
printf("------------------------------------\n");
printf("Index | Formal Parameter | Actual
Arg\n");
printf("------------------------------------\n");
for (i = 1; i <= 3; i++) {
printf(" %d | %-15s | %s\n", i, pntab[i], actual_params[i]); }
printf("------------------------------------\n\n");

// Expand macro
printf("Expanded Code:\n-------------------\n");
for (i = 0; strcmp(mdt[i], "MEND") != 0; i++)
{ char line[80] = "+ ", temp[80], *p;
strcpy(temp, mdt[i]);
p = strtok(temp, " \n");

while (p != NULL) {
   if (strncmp(p, "(P,", 3) == 0) {
   int index = p[3] - '0';
   strcat(line, actual_params[index]);
   } else {
   strcat(line, p);
```

```
        }

        strcat(line, " ");
        p = strtok(NULL, " \n");

    }

   printf("%s\n", line);

    }

   printf("-------------------\n");

getch();

}
```

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
 char info;
 struct Node* left;
 struct Node* right;
};

struct Stack {
 char info;
 struct Node* next;
};

struct Stack st[10];
int top = -1, ssm = 0;
int i, j;
char table[9][9] = {
{'_', '+', '*', '-', '/', '^', '(', ')', '$'},
{'+', '>', '<', '>', '<', '<', '<', '>', '>'},
{'*', '>', '>', '>', '>', '<', '<', '>', '>'},
{'-', '<', '<', '>', '<', '<', '<', '>', '>'},
{'/', '>', '<', '>', '>', '<', '<', '>', '>'},
{'^', '>', '>', '>', '>', '>', '<', '>', '>'},
{'(', '<', '<', '<', '<', '<', '<', '=', ' '},
{')', '>', '>', '>', '>', '>', ' ', '>', '>'},
{'$', '<', '<', '<', '<', '<', '<', ' ', '='}
};
char s[30];
```

```
struct Node* makenode(char info, struct Node* l, struct Node* r) {
 struct Node* temp = (struct Node*)malloc(sizeof(struct Node));

temp->info = info;

 temp->left = l;

 temp->right = r;

 return temp;

}


char check() {

 int i, j;

 for (i = 1; i < 9; i++) {

        if (table[i][0] == st[top].info) {

         break;

        }

 }

 for (j = 1; j < 9; j++) {

        if (table[0][j] == s[ssm]) {

         break;

        }

 }

 if (table[i][j] == ' ') {

        printf("Error: Invalid expression");

        getch();

        exit(0);

 }

 return table[i][j];

}


void inorder(struct Node* ptr) {
```

```
if (ptr != NULL) {

        inorder(ptr->left);

        printf("%c ", ptr->info);
        inorder(ptr->right);

}

}
```

```
Pre Order
if(ptr!=null)
{
printf("%c", ptr->info);
preorder(ptr->left);
preorder(ptr->right);
}
```

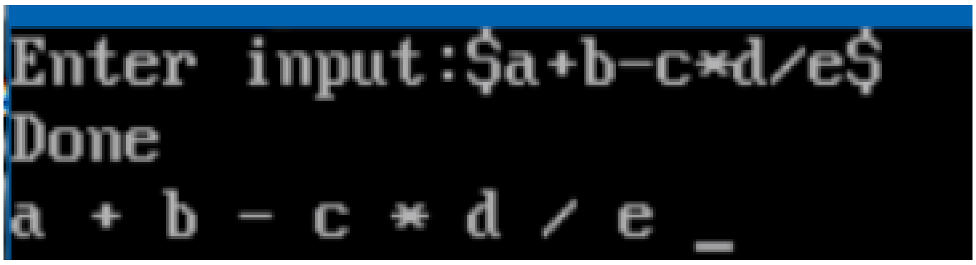
```
PostOrder
if(ptr!=null)
{
postorder(ptr->left);
postorder(ptr->right);
printf("%c", ptr->info);
}
```

```
int parse() {

 char priority;

 st[++top].info = s[ssm];

 while (1) {

        if (s[++ssm] == '$' || s[ssm] == '(' || s[ssm] == ')' || s[ssm] == '+' || s[ssm] == '*' || s[ssm]==
'- '|| s[ssm]== '/' || s[ssm]== '^') {

        if (s[ssm] == ')' && st[top].info == '(') {

                printf("Error: Invalid expression");

                getch();

                exit(0);

        }

        if ((s[ssm] == '+' || s[ssm] == '*' || s[ssm] == '-' || s[ssm] == '/' || s[ssm] == '^') && (s[ssm +
1] == '+' || s[ssm + 1] == '*' || s[ssm + 1] == '-' || s[ssm + 1] == '/' || s[ssm + 1] == '^')) {

                printf("Error: Invalid expression");

                getch();

                exit(0);

        }

        priority = check();

        while (priority == '>') {

                st[--top].next = makenode(st[top + 1].info, st[top].next, st[top + 1].next);

                priority = check();

        }

        if (priority == '<') {

                st[++top].info = s[ssm];
```

```
            }
            else {
                        if (st[top].info == '$' && !top) {
                         return 1;
                        }
                        if (st[top].info == '$' && top) {
                         return 0;
                        }
                        if (st[top].info == '(') {
                         st[--top].next = st[top + 1].next;
                        }
            }
           }
           else {
            st[top].next = makenode(s[ssm], NULL, NULL); }
 }
}

int main() {
 printf("Enter input:");
 scanf("%s", s);
 if (parse()) {
           printf("Done\n");
           inorder(st[top].next);
 }
 else {
           printf("Not done.");
 }
 getch();
 return 0;
```

```
}
```

Output



```
Enter input:$a+b-c*d/e$
Done
a + b - c * d / e _
```

Practical implementation of Scanner	
Step 1	Generate text file for given Input
Step 2	Declare two static table for Operator and Keywords
Step 3	Declare two dynamic table for constant and Symbol
Step 4	Read Input file apply STRTOK () to tokenize given input string (get logic from Help menu)
Step 5	In tokenization While loop, for each token • Check for keywords from keyword table if it exists then print [KW#index], where index is the record number in the respective table. • Else Check for Operator from Operator table if it exists then print [OP#index], where index is the record number in the respective table. • Else check that given token is digit then check whether it exists in constant table then print [CO#index], where index is the record number in the respective table, else store digits in constant table then print [CO#index] • Else check that given token exists in symbol table then print [ID#index], where index is the record number in the respective table, else store symbol in symbol table then print [ID#index]
Ex:	INPUT : INT a , b ; REAL c , d ; a = b + c * 100 ; d = a – 90 ; Static Table: OP , ; = * + - TABLE KW TABLE INT REAL Dynamic table: ID TABLE a b c d CO TABLE 100 90 OUTPUT: [KW #1] [ID #1] [OP #1] [ID#2][OP#2] [KW#2] [ID#3][OP#1][ID#4][OP#2] [ID#1][OP#3][[ID#2][OP#5][ID#3][OP#4][CO#1][OP#2] [ID#4][OP#3][ID#1][OP#6][CO#2][OP#2]

Code:-

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

#include <ctype.h>

#include <conio.h> // TurboC specific header for console I/O


      char kw[32][10] = {"int", "float", "while", "for", "do", "char",
"break",  "auto", "continue", "default", "double", "if", "else",  "enum",
   "goto", "long", "switch", "typedef", "union",  "unsigned", "void",
                              "volatile", "extern", "case",
                  "const", "return", "sizeof", "static", "struct",
                  "register", "signed"};

char op[15] = {'+', '-', '*', '/', '=', ':', ';', '<', '>',','};

char identifiers[20][10]; // Global array to store identifiers char
constants[20][10]; // Global array to store constants int ic = 0, cc =
0; // Global counters for identifiers and constants

void analyzeString(char str[]);

int main() {
 FILE *file;
 char str[100];
 file = fopen("input.txt", "r");

 if (file == NULL) {
         printf("Error opening the file.\n");
         getch(); // Wait for a key press
         return 1; // Return an error code
 }

 while (fgets(str, sizeof(str), file) != NULL) {
         analyzeString(str);
 }

 fclose(file);

 getch(); // Wait for a key press before closing the console window
return 0;
}

void analyzeString(char str[]) {
 char *ptr;
 int i, j;

 ptr = strtok(str, " \n");

 while (ptr != NULL) {
         int flag = 0;
```

```
for (i = 0; i < 32; i++) {
 if (strcmp(ptr, kw[i]) == 0) {
          printf("KW#%d ", i + 1);
          flag = 1;
          break;
 }
}

if (flag == 0) {
 for (j = 0; j < 10; j++) {
          if (ptr[0] == op[j]) {
           printf("OP#%d ", j + 1);
           flag = 1;
           break;
          }
 }
}

if (flag == 0) {
 if (isalpha(ptr[0])) {
          int isRepeated = 0;
          for (i = 0; i < ic; i++) {
            if (strcmp(ptr, identifiers[i]) == 0) {
                     printf("ID#%d ", i + 1);
                    isRepeated = 1;
                    break;
                }
                }
                if (!isRepeated) {
                 strcpy(identifiers[ic++], ptr);
                 printf("ID#%d ", ic);
                }
            } else if (isdigit(ptr[0])) {
                int isRepeated = 0;
                for (i = 0; i < cc; i++) {
                 if (strcmp(ptr, constants[i]) == 0) {
                    printf("CO#%d ", i + 1);
                    isRepeated = 1;
                    break;
                 }
                }
                if (!isRepeated) {
                 strcpy(constants[cc++], ptr);
                 printf("CO#%d ", cc);
                }
            }
            }

            ptr = strtok(NULL, " \n");
        }
        }
```

File—

```
int a , b ;
float c , d ;
a = b + c * 100 ;
d = a – 90 ;
c = 90 ;
```

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct treenode
{
        char info;
        struct treenode *left;
        struct treenode *right;
}*temp,*a,*b,*c,*d,*temp1,*root
; typedef struct treenode node;
node * proc_e(char input[]); node
* proc_t(char input[]); node *
proc_v(char input[]); void
traversal(node *temp); int ssm=0;
void main()
{
        char input[20];
        ssm=0;
clrscr();
        printf("Enter String:");
        gets(input);
        root=proc_e(input);
        printf("Parser Tree: ");
        traversal(root);
getch();
}
```

```
node * proc_e(char input[])
{
        char ch;
        a=proc_t(input);
        while(input[ssm]=='+' ||
        input[ssm]=='-') {
                ch=input[ssm];
                ssm++;
                b=proc_t(input);
                temp=(node *)malloc(sizeof(node));
                temp->info=ch;
                temp->left=a;
                temp->right=b;
                a=temp;
        }
        return a;
}
node * proc_t(char input[])
{
        char ch;
        c=proc_v(input);
        ssm+=1;
        while(input[ssm]=='*' ||
        input[ssm]=='/') {
                ch=input[ssm];
                ssm++;
                d=proc_v(input);
                temp=(node *)malloc(sizeof(node));
                temp->info=ch;
                temp->left=c;
                temp->right=d;
                c=temp;
                ssm+=1;
        }
        return c;
}
node * proc_v(char input[])
{
        if(isalpha(input[ssm]))
        {
                temp=(node *)malloc(sizeof(node));
                temp->info=input[ssm];
                temp->left=NULL;
                temp->right=NULL;
                return temp;
        }
        else
        {
                printf("Error %c",input[ssm]);
                exit(0);
        }
}
```

```
void traversal(node *temp1)
{
        if(temp1!=NULL)
        {
                traversal(temp1->left);
                printf("%c",temp1->info);
                traversal(temp1->right);
        }
}
```

Simple Scanner

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h> // Include ctype.h for isalpha and isdigit functions

char
kw[32][10]={"int","float","while","for","do","char","break","auto","continue",

"default","double","if","else","enum","goto","long","switch","typedef","union"
,  "unsign","void","volatile","extern","case","const"}; char
op[15]={'+','-','*','/','=',':',';','<','>'};
char ip[15];
char identifiers[20];
char constants[20];
char operators[20];
int oc=0,cc=0,ic=0;

void main()
 {
 char str[20];
 char *ptr;
 int i=0,j;

 clrscr();
 printf("\n Enter String");
 scanf("%[^\n]s",str);
 ptr=strtok(str," ");
 printf("Keyword is:- ");
while(ptr!=NULL)
 {
          int flag=0;
          for(i=0;i<32;i++)
           {
           if(strcmp(ptr,kw[i])==0)
          {
                    printf("%s , ",ptr);
                    flag=1;
           }
          }
          if(flag==0)
           {
           strcat(ip,ptr);
          }
          ptr=strtok(NULL," ");
 }
 for(i=0;i<strlen(ip);i++)
{
          if(isalpha(ip[i]))
           {
           identifiers[ic] = ip[i];
           ic++;
          }
          else if(isdigit(ip[i]))
           {
           constants[cc]=ip[i];
           cc++;
           }
          else
           {
           for(j = 0; j < sizeof(op); j++)
{
if(ip[i] == op[j])
{
operators[oc] = ip[i];
oc++;
break; // Exit the loop once the operator is found  }
}
}
}
printf("\n Identifiers : ");
for(i=0;i<ic;i++)
{
printf("%c ",identifiers[i]);
}
printf(" \nConstants : ");
for(i=0;i<cc;i++)
{
printf("%c ",constants[i]);
```

```
}
printf(" \nOperators : ");
for(i=0;i<oc;i++)
{
printf("%c ", operators[i]);
}
getch();
}
```

Scanner with dynamic table.

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

char
kw[32][10]={"int","float","while","for","do","char","break","auto","continue",

"default","double","if","else","enum","goto","long","switch","typedef","union"
,  "unsign","void","volatile","extern","case","const"}; char
op[15]={'+','-','*','/','=',':',';','<','>'};
char ip[100];
char identifiers[100];
char constants[100];
char operators[100];
int oc=0,cc=0,ic=0;
```

```
void main()
 {
 char str[100];
 char *ptr;
 int i=0, j=0, flag=0; // Declare loop variables and flags

 clrscr();
 printf("\n Enter String");
 scanf("%[^\n]s",str);
 ptr=strtok(str," ");
 printf("Keyword is:- ");
 while(ptr!=NULL)
 {
 flag=0;
 for(i=0;i<32;i++)
 {
 if(strcmp(ptr,kw[i])==0)
 {
 printf("%s , ",ptr);
 flag=1;
 }
 }
 if(flag==0)
 {
 strcat(ip,ptr);
 }
 ptr=strtok(NULL," ");
 }
 for(i=0;i<strlen(ip);i++)
 {
 if(isalpha(ip[i]))
 {
 // Check if the identifier already exists  int
exists = 0;
 for(j = 0; j < ic; j++) {
 if(ip[i] == identifiers[j]) {  exists =
1;
 break;
 }
 }
 if(!exists) {
 identifiers[ic] = ip[i];
 ic++;
 }
 }
 else if(isdigit(ip[i]))
 {
 // Check if the constant already exists  int
exists = 0;
 for(j = 0; j < cc; j++) {
 if(ip[i] == constants[j]) {  exists =
1;
 break;
 }
 }
 if(!exists) {
 constants[cc] = ip[i];  cc++;
 }
 }
 else
 {
 for(j = 0; j < sizeof(op); j++)  {
 if(ip[i] == op[j])
 {
 operators[oc] = ip[i];  oc++;
 break;
 }
 }
 }
 }
 printf("\n Identifiers : ");
 for(i=0;i<ic;i++)
 {
 printf("%c ",identifiers[i]);  }
 printf(" \nConstants : ");
 for(i=0;i<cc;i++)
 {
 printf("%c ",constants[i]);  }
 printf(" \nOperators : ");
```

```
 for(i=0;i<oc;i++)
 {
 printf("%c ", operators[i]);
 }

 printf("\n\nDynamic Table of Constants and Identifiers:\n");
printf("Identifier\n");
 for(i=0; i<ic; i++)
 {
 printf("%c\n", identifiers[i]);
 }
 printf("Constant\n");
 for(i=0; i<cc; i++)
 {
 printf("%c\n", constants[i]);
 }
 getch();
}
```