# MADHAV INSTITUTE OF TECHNOLOGY & SCIENCE, GWALIOR (M.P.)


## COMPILER DESIGN

## (150601)

## LAB REPORT FILE

**SUBMITTED TO:**

**PROF. MAHESH PARMAR**

**SUBMITTED BY:**

**ABHINAV CHATURVEDI**

**ROLL NO: 0901CS191003**

**CSE III YEAR (VI SEM.)**

# TABLE OF CONTENTS

# EXPERIMENT – 1

Q: Design a Program to convert NDFA to DFA.

Ans:

Source code:

```python
import pandas as pd
#0901CS191003-Abhinav Chaturvedi
nfa = {}
n = int(input("No. of states : "))
t = int(input("No. of transitions : "))
for i in range(n):
    state = input("\nstate name : ")
    nfa[state] = {}
    for j in range(t):
        path = input("\nTransition path : ")
        print("Enter end state from state {} travelling through path {} : ".format(state,path),end="")
        reaching_state = [x for x in input().split()]
        nfa[state][path] = reaching_state

print("\nNFA :- \n")
print(nfa)
print("\nPrinting NFA table :- ")
nfa_table = pd.DataFrame(nfa)
print(nfa_table.transpose())

print("Enter final state of NFA : ")
nfa_final_state = [x for x in input().split()]

new_states_list = []
dfa = {}
keys_list = list(list(nfa.keys())[0])
```

```
path_list = list(nfa[keys_list[0]].keys())

dfa[keys_list[0]] = {}
for y in range(t):
    var = "".join(nfa[keys_list[0]][path_list[y]])
    dfa[keys_list[0]][path_list[y]] = var
    if var not in keys_list:
        new_states_list.append(var)
        keys_list.append(var)


while len(new_states_list) != 0:
    dfa[new_states_list[0]] = {}
    for _ in range(len(new_states_list[0])):
        for i in range(len(path_list)):
            temp = []
            for j in range(len(new_states_list[0])):
                temp += nfa[new_states_list[0][j]][path_list[i]]
            s = ""
            s = s.join(temp)
            if s not in keys_list:
                new_states_list.append(s)
                keys_list.append(s)
            dfa[new_states_list[0]][path_list[i]] = s

    new_states_list.remove(new_states_list[0])
```

```

print("\nDFA :- \n")
print(dfa)
print("\nPrinting DFA table :- ")
dfa_table = pd.DataFrame(dfa)
print(dfa_table.transpose())

dfa_states_list = list(dfa.keys())
dfa_final_states = []
for x in dfa_states_list:
    for i in x:
        if i in nfa_final_state:
            dfa_final_states.append(x)
            break

print("\nFinal states of the DFA are : ",dfa_final_states)
```

# OUTPUT:

```
No. of states : 4
No. of transitions : 2

state name : A

Transition path : a
Enter end state from state A travelling through path a : A B

Transition path : b
Enter end state from state A travelling through path b : A

state name : B

Transition path : a
Enter end state from state B travelling through path a : C

Transition path : b
Enter end state from state B travelling through path b : C

state name : C

Transition path : a
Enter end state from state C travelling through path a : D

Transition path : b
Enter end state from state C travelling through path b : D
```

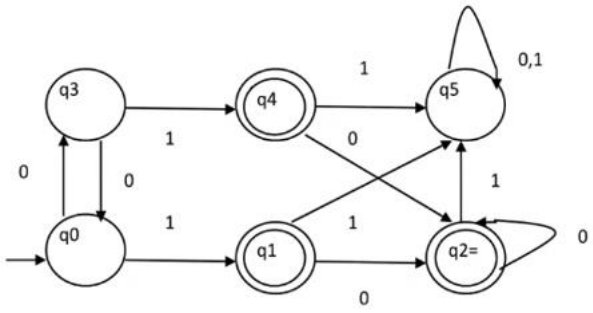
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

state name : D

Transition path : a
Enter end state from state D travelling through path a :

Transition path : b
Enter end state from state D travelling through path b :

NFA :-

{'A': {'a': ['A', 'B'], 'b': ['A']}, 'B': {'a': ['C'], 'b': ['C']}, 'C': {'
a': ['D'], 'b': ['D']}, 'D': {'a': [], 'b': []}}

Printing NFA table :-
        a    b
A  [A, B]  [A]
B     [C]  [C]
C     [D]  [D]
D      []   []
Enter final state of NFA :
D
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
DFA :-

{'A': {'a': 'AB', 'b': 'A'}, 'AB': {'a': 'ABC', 'b': 'AC'}, 'ABC': {'a': 'A
BCD', 'b': 'ACD'}, 'AC': {'a': 'ABD', 'b': 'AD'}, 'ABCD': {'a': 'ABCD', 'b'
: 'ACD'}, 'ACD': {'a': 'ABD', 'b': 'AD'}, 'ABD': {'a': 'ABC', 'b': 'AC'}, '
AD': {'a': 'AB', 'b': 'A'}}

Printing DFA table :-
         a    b
A       AB    A
AB     ABC   AC
ABC   ABCD  ACD
AC     ABD   AD
ABCD  ABCD  ACD
ACD    ABD   AD
ABD    ABC   AC
AD      AB    A

Final states of the DFA are :  ['ABCD', 'ACD', 'ABD', 'AD']
```

# EXPERIMENT – 2

Q: Design a Program for minimization of DFA.




Ans:

Source code:

```c
//0901CS191003
//DFA Minimization
#include <stdio.h>
#include <string.h>
#define STATES  99
#define SYMBOLS 20

int N_symbols;  /* number of input symbols */
int N_DFA_states;   /* number of DFA states */
char *DFA_finals;   /* final-state string */
int DFAtab[STATES][SYMBOLS];

char StateName[STATES][STATES+1];   /* state-name table */

int N_optDFA_states;    /* number of optimized DFA states */
int OptDFA[STATES][SYMBOLS];
char NEW_finals[STATES+1];

/*
    Print state-transition table.
    State names: 'A', 'B', 'C', ...
*/
void print_dfa_table(
    int tab[][SYMBOLS], /* DFA table */
    int nstates,    /* number of states */
    int nsymbols,   /* number of input symbols */
```

```
    char *finals)
{
    int i, j;

    puts("\nDFA: STATE TRANSITION TABLE");

    /* input symbols: '0', '1', ... */
    printf("     | ");
    for (i = 0; i < nsymbols; i++) printf("  %c  ", '0'+i);

    printf("\n-----+--");
    for (i = 0; i < nsymbols; i++) printf("-----");
    printf("\n");

    for (i = 0; i < nstates; i++) {
        printf("  %c  | ", 'A'+i);  /* state */
        for (j = 0; j < nsymbols; j++)
            printf("  %c  ", tab[i][j]);    /* next state */
        printf("\n");
    }
    printf("Final states = %s \n", finals);
}

/*
    Initialize NFA table.
*/
void load_DFA_table()
{

    DFAtab[0][0] = 'B'; DFAtab[0][1] = 'C';
    DFAtab[1][0] = 'A'; DFAtab[1][1] = 'D';
    DFAtab[2][0] = 'E'; DFAtab[2][1] = 'F';
    DFAtab[3][0] = 'E'; DFAtab[3][1] = 'F';
    DFAtab[4][0] = 'E'; DFAtab[4][1] = 'F';
    DFAtab[5][0] = 'F'; DFAtab[5][1] = 'F';

    DFA_finals = "CDE";
    N_DFA_states = 6;
    N_symbols = 2;
}

/*
    Get next-state string for current-state string.
*/
void get_next_state(char *nextstates, char *cur_states,
    int dfa[STATES][SYMBOLS], int symbol)
{
    int i, ch;
```

```
    for (i = 0; i < strlen(cur_states); i++)
        *nextstates++ = dfa[cur_states[i]-'A'][symbol];
    *nextstates = '\0';
}

/*
    Get index of the equivalence states for state 'ch'.
    Equiv. class id's are '0', '1', '2', ...
*/
char equiv_class_ndx(char ch, char stnt[][STATES+1], int n)
{
    int i;

    for (i = 0; i < n; i++)
        if (strchr(stnt[i], ch)) return i+'0';
    return -1;  /* next state is NOT defined */
}

/*
    Check if all the next states belongs to same equivalence class.
    Return value:
        If next state is NOT unique, return 0.
        If next state is unique, return next state --> 'A/B/C/...'
    's' is a '0/1' string: state-id's
*/
char is_one_nextstate(char *s)
{
    char equiv_class;   /* first equiv. class */

    while (*s == '@') s++;
    equiv_class = *s++; /* index of equiv. class */

    while (*s) {
        if (*s != '@' && *s != equiv_class) return 0;
        s++;
    }

    return equiv_class; /* next state: char type */
}

int state_index(char *state, char stnt[][STATES+1], int n, int *pn,
    int cur)    /* 'cur' is added only for 'printf()' */
{
    int i;
    char state_flags[STATES+1]; /* next state info. */

    if (!*state) return -1; /* no next state */
```

```
    for (i = 0; i < strlen(state); i++)
        state_flags[i] = equiv_class_ndx(state[i], stnt, n);
    state_flags[i] = '\0';

    //printf("   %d:[%s]\t--> [%s] (%s)\n",
        //cur, stnt[cur], state, state_flags);

    if (i=is_one_nextstate(state_flags))
        return i-'0';   /* deterministic next states */
    else {
        strcpy(stnt[*pn], state_flags); /* state-division info */
        return (*pn)++;
    }
}

/*
    Divide DFA states into finals and non-finals.
*/
int init_equiv_class(char statename[][STATES+1], int n, char *finals)
{
    int i, j;

    if (strlen(finals) == n) {  /* all states are final states */
        strcpy(statename[0], finals);
        return 1;
    }

    strcpy(statename[1], finals);   /* final state group */

    for (i=j=0; i < n; i++) {
        if (i == *finals-'A') {
            finals++;
        } else statename[0][j++] = i+'A';
    }
    statename[0][j] = '\0';

    return 2;
}

/*
    Get optimized DFA 'newdfa' for equiv. class 'stnt'.
*/
int get_optimized_DFA(char stnt[][STATES+1], int n,
    int dfa[][SYMBOLS], int n_sym, int newdfa[][SYMBOLS])
{
    int n2=n;       /* 'n' + <num. of state-division info> */
    int i, j;
```

```
    char nextstate[STATES+1];

    for (i = 0; i < n; i++) {    /* for each pseudo-DFA state */
        for (j = 0; j < n_sym; j++) {    /* for each input symbol */
            get_next_state(nextstate, stnt[i], dfa, j);
            newdfa[i][j] = state_index(nextstate, stnt, n, &n2, i)+'A';
        }
    }

    return n2;
}

/*
    char 'ch' is appended at the end of 's'.
*/
void chr_append(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);
            }
}

/*
    Divide first equivalent class into subclasses.
        stnt[i1] : equiv. class to be segmented
        stnt[i2] : equiv. vector for next state of stnt[i1]
    Algorithm:
        - stnt[i1] is splitted into 2 or more classes 's1/s2/...'
        - old equiv. classes are NOT changed, except stnt[i1]
        - stnt[i1]=s1, stnt[n]=s2, stnt[n+1]=s3, ...
    Return value: number of NEW equiv. classses in 'stnt'.
*/
int split_equiv_class(char stnt[][STATES+1],
```

```
    int i1, /* index of 'i1'-th equiv. class */
    int i2, /* index of equiv. vector for 'i1'-th class */
    int n,  /* number of entries in 'stnt' */
    int n_dfa)  /* number of source DFA entries */
{
    char *old=stnt[i1], *vec=stnt[i2];
    int i, n2, flag=0;
    char newstates[STATES][STATES+1];   /* max. 'n' subclasses */

    for (i=0; i < STATES; i++) newstates[i][0] = '\0';

    for (i=0; vec[i]; i++)
        chr_append(newstates[vec[i]-'0'], old[i]);

    for (i=0, n2=n; i < n_dfa; i++) {
        if (newstates[i][0]) {
            if (!flag) {    /* stnt[i1] = s1 */
                strcpy(stnt[i1], newstates[i]);
                flag = 1;   /* overwrite parent class */
            } else  /* newstate is appended in 'stnt' */
                strcpy(stnt[n2++], newstates[i]);
        }
    }

    sort(stnt, n2); /* sort equiv. classes */

    return n2;  /* number of NEW states(equiv. classes) */
}

/*
    Equiv. classes are segmented and get NEW equiv. classes.
*/
int set_new_equiv_class(char stnt[][STATES+1], int n,
    int newdfa[][SYMBOLS], int n_sym, int n_dfa)
{
    int i, j, k;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n_sym; j++) {
            k = newdfa[i][j]-'A';   /* index of equiv. vector */
            if (k >= n)  /* equiv. class 'i' should be segmented */
                return split_equiv_class(stnt, i, k, n, n_dfa);
        }
    }

    return n;
}
/*
```

```
void print_equiv_classes(char stnt[][STATES+1], int n)
{
    int i;

    printf("\nEQUIV. CLASS CANDIDATE ==>");
    for (i = 0; i < n; i++)
        printf(" %d:[%s]", i, stnt[i]);
    printf("\n");
}
 */
/*
    State-minimization of DFA: 'dfa' --> 'newdfa'
    Return value: number of DFA states.
*/
int optimize_DFA(
    int dfa[][SYMBOLS], /* DFA state-transition table */
    int n_dfa,  /* number of DFA states */
    int n_sym,  /* number of input symbols */
    char *finals,   /* final states of DFA */
    char stnt[][STATES+1],  /* state name table */
    int newdfa[][SYMBOLS])  /* reduced DFA table */
{
    char nextstate[STATES+1];
    int n;  /* number of new DFA states */
    int n2; /* 'n' + <num. of state-dividing info> */

    n = init_equiv_class(stnt, n_dfa, finals);

    while (1) {
        //print_equiv_classes(stnt, n);
        n2 = get_optimized_DFA(stnt, n, dfa, n_sym, newdfa);
        if (n != n2)
            n = set_new_equiv_class(stnt, n, newdfa, n_sym, n_dfa);
        else break; /* equiv. class segmentation ended!!! */
    }

    return n;   /* number of DFA states */
}

/*
    Check if 't' is a subset of 's'.
*/
int is_subset(char *s, char *t)
{
    int i;

    for (i = 0; *t; i++)
        if (!strchr(s, *t++)) return 0;
```

```
    return 1;
}

/*
    New finals states of reduced DFA.
*/
void get_NEW_finals(
    char *newfinals,    /* new DFA finals */
    char *oldfinals,    /* source DFA finals */
    char stnt[][STATES+1],  /* state name table */
    int n)  /* number of states in 'stnt' */
{
    int i;

    for (i = 0; i < n; i++)
        if (is_subset(oldfinals, stnt[i])) *newfinals++ = i+'A';
    *newfinals++ = '\0';
}

void print_min_dfa_table(
    int tab[][SYMBOLS], /* DFA table */
    int nstates,    /* number of states */
    int nsymbols,   /* number of input symbols */
    char *finals)
{
    int i, j;

    puts("\nMINIMISED DFA: STATE TRANSITION TABLE");

    /* input symbols: '0', '1', ... */
    printf("     |      ");
    for (i = 0; i < nsymbols; i++) printf(" %c  ", '0'+i);

    printf("\n-----+-----");
    for (i = 0; i < nsymbols; i++) printf("-----");
    printf("\n");
    for (i = 0; i < nstates; i++) {
        printf(" Q%c  | ", '0'+i);  /* state */
        for (j = 0; j < nsymbols; j++){
            if(tab[i][j]=='A')  printf("    Q%c  ", '0');
            if(tab[i][j]=='B')  printf("    Q%c  ", '1');
            if(tab[i][j]=='C')  printf("    Q%c  ", '2');
            //printf("    %c  ", tab[i][j]);    /* next state */
        }
        printf("\n");
    }
    if(*finals=='A') printf("Final states = %s\n", "Q0");
    if(*finals=='B') printf("Final states = %s\n", "Q1");
```

```
    if(*finals=='C') printf("Final states = %s\n", "Q2");
}


int main()
{
    load_DFA_table();
    print_dfa_table(DFAtab, N_DFA_states, N_symbols, DFA_finals);

    N_optDFA_states = optimize_DFA(DFAtab, N_DFA_states,
            N_symbols, DFA_finals, StateName, OptDFA);
    get_NEW_finals(NEW_finals, DFA_finals, StateName, N_optDFA_states);

    print_min_dfa_table(OptDFA, N_optDFA_states, N_symbols, NEW_finals);
    return 0;
}
```

# OUTPUT:

```
DFA: STATE TRANSITION TABLE
      |   0    1
------+------------
  A   |   B    C
  B   |   A    D
  C   |   E    F
  D   |   E    F
  E   |   E    F
  F   |   F    F
Final states = CDE

MINIMISED DFA: STATE TRANSITION TABLE
      |        0   1
------+---------------
 Q0   |      Q0      Q1
 Q1   |      Q1      Q2
 Q2   |      Q2      Q2
Final states = Q1
```

# EXPERIMENT – 3

Q: Design a Program for Lexical Analyzer.

Ans:

Source code:

```
#include <bits/stdc++.h>
using namespace std;

//0901CS191003 Abhinav Chaturvedi

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
    return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
```

```
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
         !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
```

```
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
                  sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("'%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
                   || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' IS A REAL NUMBER\n", subStr);
```

```
            else if (validIdentifier(subStr) == true
                     && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                     && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}

// DRIVER FUNCTION
int main()
{
     // maximum length of string is 100 here
    char str[100] = "int a = b + 1c; ";

    cout<<"Output: \n\n ";
    parse(str); // calling the parse function

    return 0;
}
```

# OUTPUT:

```
 "f:\abhinav chaturvedi\compiler_design\assignment_3\"assignment_3
Output:

 'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'1c' IS NOT A VALID IDENTIFIER
```

# EXPERIMENT – 4

Q: Write a program to parse using the Brute force technique of Top-down parsing.

Ans:

Source code:

OUTPUT:

# EXPERIMENT – 5

Q: Develop LL (1) parser (Construct parse table also).

Ans:

Source code:

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
// table[i][j] stores the index of production that must be applied on
// ith varible if the input is jth nonterminal
int table[100][TSIZE];
// stores all list of terminals the ASCII value if use to index terminals
// terminal[i] = 1 means the character with ASCII value is a terminal
char terminal[TSIZE];
// stores all list of terminals only Upper case letters from 'A' to 'Z'
// can be nonterminals nonterminal[i] means ith alphabet is present as
// nonterminal is the grammar
char nonterminal[26];
// structure to hold each production str[] stores the production
// len is the length of production
struct product {
    char str[100];
    int len;
}pro[20];
// no of productions in form A->ß
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
// stores first of each production in form A->ß
char first_rhs[100][TSIZE];
// check if the symbol is nonterminal
int isNT(char c) {
    return c >= 'A' && c <= 'Z';
}
// reading data from the file
void readFromFile() {
    FILE* fptr;
    fptr = fopen("text.txt", "r");
    char buffer[255];
    int i;
    int j;
    while (fgets(buffer, sizeof(buffer), fptr)) {
        printf("%s", buffer);
        j = 0;
        nonterminal[buffer[0] - 'A'] = 1;
```

```
        for (i = 0; i < strlen(buffer) - 1; ++i) {
            if (buffer[i] == '|') {
                ++no_pro;
                pro[no_pro - 1].str[j] = '\0';
                pro[no_pro - 1].len = j;
                pro[no_pro].str[0] = pro[no_pro - 1].str[0];
                pro[no_pro].str[1] = pro[no_pro - 1].str[1];
                pro[no_pro].str[2] = pro[no_pro - 1].str[2];
                j = 3;
            }
            else {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>')
{
                    terminal[buffer[i]] = 1;
                }
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}
void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}
void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}
void FOLLOW() {
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro) {
        for (k = 0; k < 26; ++k) {
            if (!nonterminal[k])    continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i) {
                for (j = 3; j < pro[i].len; ++j) {
                    if (nt == pro[i].str[j]) {
                        for (x = j + 1; x < pro[i].len; ++x) {
```

```
                            char sc = pro[i].str[x];
                            if (isNT(sc)) {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['^'])
                                    continue;
                            }
                            else {
                                follow[nt - 'A'][sc] = 1;
                            }
                            break;
                        }
                        if (x == pro[i].len)
                            add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
                    }
                }
            }
        }
    }
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^') {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
        }
    }
}
void FIRST() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first[pro[i].str[0] - 'A']['^'] = 1;
        }
        ++t;
```

```
    }
}
void add_FIRST_A_to_FIRST_RHS__B(char A, int B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}
// Calculates FIRST(ß) for each A->ß
void FIRST_RHS() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first_rhs[i][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first_rhs[i]['^'] = 1;
        }
        ++t;
    }
}
int main() {
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
    FIRST_RHS();
    int i, j, k;

    // display first of each variable
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            char c = pro[i].str[0];
            printf("FIRST OF %c: ", c);
            for (j = 0; j < TSIZE; ++j) {
```

```
                if (first[c - 'A'][j]) {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }

    // display follow of each variable
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            char c = pro[i].str[0];
            printf("FOLLOW OF %c: ", c);
            for (j = 0; j < TSIZE; ++j) {
                if (follow[c - 'A'][j]) {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }
    // display first of each variable ß
    // in form A->ß
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
        printf("FIRST OF %s: ", pro[i].str);
        for (j = 0; j < TSIZE; ++j) {
            if (first_rhs[i][j]) {
                printf("%c ", j);
            }
        }
        printf("\n");
    }

    // the parse table contains '$'
    // set terminal['$'] = 1
    // to include '$' in the parse table
    terminal['$'] = 1;

    // the parse table do not read '^'
    // as input
    // so we set terminal['^'] = 0
    // to remove '^' from terminals
    terminal['^'] = 0;

    // printing parse table
    printf("\n");
```

```
    printf("\n\t*************** LL(1) PARSING TABLE *****************\n");
    printf("\t-----------------------------------------------------\n");
    printf("%-10s", "");
    for (i = 0; i < TSIZE; ++i) {
        if (terminal[i])    printf("%-10c", i);
    }
    printf("\n");
    int p = 0;
    for (i = 0; i < no_pro; ++i) {
        if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
            p = p + 1;
        for (j = 0; j < TSIZE; ++j) {
            if (first_rhs[i][j] && j != '^') {
                table[p][j] = i + 1;
            }
            else if (first_rhs[i]['^']) {
                for (k = 0; k < TSIZE; ++k) {
                    if (follow[pro[i].str[0] - 'A'][k]) {
                        table[p][k] = i + 1;
                    }
                }
            }
        }
    }
    k = 0;
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            printf("%-10c", pro[i].str[0]);
            for (j = 0; j < TSIZE; ++j) {
                if (table[k][j]) {
                    printf("%-10s", pro[table[k][j] - 1].str);
                }
                else if (terminal[j]) {
                    printf("%-10s", "");
                }
            }
            ++k;
            printf("\n");
        }
    }
}
```

# INPUT:

text.txt

assignment_5 > text.txt

```
1    E->TA
2    A->+TA|^
3    T->FB
4    B->*FB|^
5    F->t|(E)
```

# OUTPUT:

```
\abhinav chaturvedi\compiler_design\assignment_5\"assignment_5
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)
FIRST OF E: ( t
FIRST OF A: + ^
FIRST OF T: ( t
FIRST OF B: * ^
FIRST OF F: ( t
```

```
FOLLOW OF E: $ * +
FOLLOW OF A: $ * +
FOLLOW OF T: $ * +
FOLLOW OF B: $ * +
FOLLOW OF F: $ * +

FIRST OF E->TA: ( t
FIRST OF A->+TA: +
FIRST OF A->^: ^
FIRST OF T->FB: ( t
FIRST OF B->*FB: *
FIRST OF B->^: ^
FIRST OF F->t: t
FIRST OF F->(E: (
```

```
        ********** LL(1) PARSING TABLE **********
        ------------------------------------------------------
          $         (         *         +         t
 E                  E->TA                         E->TA
 A        A->^                 A->^      A->^
 T                  T->FB                         T->FB
 B        B->^                 B->^      B->^
 F                  F->(E                         F->t

 f:\abhinav chaturvedi\compiler_design\assignment_5>
```

# EXPERIMENT – 6

Q: Develop an operator precedence parser (Construct parse table also).

Ans:

Source code:

```
import numpy as np
def stringcheck():
    a=list(input("Enter the operator used in the given grammar including the
terminals\n non-treminals should be in cursive(small)letter \n"))
    a.append('$')
    print(a)
    l=list("abcdefghijklmnopqrstuvwxyz")
    o=list('(/*%+-)')
    p=list('(/*%+-)')
    n=np.empty([len(a)+1,len(a)+1],dtype=str,order="C")
    for j in range(1,len(a)+1):
        n[0][j]=a[j-1]
        n[j][0]=a[j-1]
    for i in range(1,len(a)+1):
        for j in range(1,len(a)+1):
            if((n[i][0] in l)and(n[0][j] in l)):
                n[i][j]=""
            elif((n[i][0] in l)):
                n[i][j]=">"
            elif((n[i][0] in o) and (n[0][j] in o)):
                if(o.index(n[i][0])<=o.index(n[0][j])):
                   n[i][j]=">"
                else:
                    n[i][j]="<"
            elif((n[i][0] in o)and n[0][j]in l):
                n[i][j]="<"
            elif(n[i][0]=="$" and n[0][j]!="$"):
                n[i][j]="<"
            elif(n[0][j]=="$" and n[i][0]!="$" ):
                 n[i][j]=">"
            else:
                break
    print("The Operator Precedence Relational
Table\n==========================================")
    print(n)
    i=list(input("Enter the string want to be checked(non-terminals should be
in cursive(small) letter..."))
    i.append("$")
```

```
    s=[None]*len(i)
    q=0
    s.insert(q,"$")
    x=[row[0] for row in n]
    y=list(n[0])
    h=0
    while(s[0]!=s[1]):
        if((i[len(i)-2] in p)):
            break
        elif((s[q] in x)and(i[h]in y )):
            if(n[x.index(s[q])][y.index(i[h])]=="<"):
                q+=1
                s.insert(q,i[h])
                h+=1
            elif(n[x.index(s[q])][y.index(i[h])]==">"):
                s.pop(q)
                q-=1
            elif((n[x.index(s[q])][y.index(i[h])]=='')and ((s[q]=="$") and
(i[h]=="$"))):
                s[1]=s[0]
        else:
            break
    if(s[0]!=s[1]):
        return False
    else:
        return True
def grammarcheck(i):
    print("Enter the",str(i+1)+"th grammar(production) want to be checked\n
For null production please enter any special symbol or whitespace...")
    b=list(input().split("->"))
    f=list("abcdefghijklmnopqrstuvwxyz")
    if(b[0]==" " or b[0]=="" or b[0] in f or len(b)==1):
        return False
    else:
        b.pop(0)
        b=list(b[0])
        s=list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
        o=list("(abcdefghijklmnopqrstuvwxyz^/*+-|)")
        sp=['!','@','#','$','?','~','`',',',';',':','"','=','_','&',"'","","
"]
        for i in range(0,len(b),2):
            if(b[i]==" "):
                g=False
            elif(b[i] in sp):
                g=False
                break
            elif(b[len(b)-1] in o and ((b[0]=="(" and b[len(b)-1]==")" )or
(b.count("(")==b.count(")")))):
```

```
                g=True
            elif(b[i] in f):
                g=True
            elif(b[len(b)-1] in o):
                g=False
            elif((i==len(b)-1) and (b[i] in s)):
                g=True
            elif((i==len(b)-1) and (b[i] not in s) and (b[i] in o)and b[i-1]
in o):
                g=True
            elif((b[i] in s) and(b[i+1]in o)):
                g=True
            elif((b[i] in s) and (b[i+1] in s)):
                g=False
                break
            else:
                g=False
                break
        if(g==True):
            return True
        else:
            return False
c=int(input("Enter the number of LHS variables..\n"))
for i in range(c):
    if(grammarcheck(i)):
        t=True
    else:
        t=False
        break
if(t):
    print("Grammar is accepted")
    if(stringcheck()):
        print("String is accepted")
    else:
        print("String is not accepted")

else:
    print("Grammar is not accepted ")
```

# OUTPUT:

```
Enter the number of LHS variables..
2
Enter the 1th grammar(production) want to be checked
 For null production please enter any special symbol or whitespace...
A->a
Enter the 2th grammar(production) want to be checked
 For null production please enter any special symbol or whitespace...
A->A+A
Grammar is accepted
Enter the operator used in the given grammar including the terminals
 non-treminals should be in cursive(small)letter
a+
['a', '+', '$']
The Operator Precedence Relational Table
```

```
The Operator Precedence Relational Table
=========================================
[['' 'a' '+' '$']
 ['a' '' '>' '>']
 ['+' '<' '>' '>']
 ['$' '<' '<' '']]
Enter the string want to be checked(non-terminals should be in cursive(small) letter...a+a
String is accepted

F:\abhinav chaturvedi\compiler_design\assignment_6>
```

# EXPERIMENT – 7

Q: Develop a recursive descent parser.

Ans:

Source code:

```
#include"stdio.h"
#include"conio.h"
#include"string.h"
#include"stdlib.h"
#include"ctype.h"

char ip_sym[15],ip_ptr=0,op[50],tmp[50];
void e_prime();
void e();
void t_prime();
void t();
void f();
void advance();
int n=0;
void e()
{
 strcpy(op,"TE'");
 printf("E=%-25s",op);
 printf("E->TE'\n");
 t();
 e_prime();
}

void e_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
    if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='E';n++);
if(ip_sym[ip_ptr]=='+')
 {
     i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
```

```
  op[n++]='+';
  op[n++]='T';
  op[n++]='E';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("E'->+TE'\n");
  advance();
  t();
  e_prime();
 }
 else
 {
     op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("E'->e");
 }
}
void t()
{
 int i,n=0,l;
 for(i=0;i<=strlen(op);i++)
  if(op[i]!='e')
   tmp[n++]=op[i];
 strcpy(op,tmp);
 l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);

 i=n+1;
 do
 {
  op[i+2]=op[i];
  i++;
 }while(i < l);
 op[n++]='F';
 op[n++]='T';
 op[n++]=39;
 printf("E=%-25s",op);
 printf("T->FT'\n");
 f();
 t_prime();
}

void t_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
```

```
    if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
if(ip_sym[ip_ptr]=='*')
 {
     i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i < l);
  op[n++]='*';
  op[n++]='F';
  op[n++]='T';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("T'->*FT'\n");
  advance();
  f();
  t_prime();
 }
 else
 {
   op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("T'->e\n");
 }
}

void f()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
    if(op[i]!='e')
 tmp[n++]=op[i];
 strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='F';n++);
 if((ip_sym[ip_ptr]=='i')||(ip_sym[ip_ptr]=='I'))
 {
 op[n]='i';
 printf("E=%-25s",op);
 printf("F->i\n");
 advance();
```

```
 }
 else
 {
  if(ip_sym[ip_ptr]=='(')
  {
   advance();
   e();
   if(ip_sym[ip_ptr]==')')
   {
    advance();
     i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
  op[n++]='(';
  op[n++]='E';
  op[n++]=')';
  printf("E=%-25s",op);
  printf("F->(E)\n");
   }
  }
  else
  {
   printf("\n\t syntax error");
   getch();
   exit(1);
  }
 }
}

void advance()
{
 ip_ptr++;
}

void main()
{
 int i;
//  clrscr();
 printf("\nGrammar without left recursion");
 printf("\n\t\t E->TE' \n\t\t E'->+TE'|e \n\t\t T->FT' ");
 printf("\n\t\t T'->*FT'|e \n\t\t F->(E)|i");
 printf("\n Enter the input expression:");
 gets(ip_sym);
 printf("Expressions");
 printf("\t Sequence of production rules\n");
```

```
 e();
 for(i=0;i < strlen(ip_sym);i++)
{
 if(ip_sym[i]!='+'&&ip_sym[i]!='*'&&ip_sym[i]!='('&&
    ip_sym[i]!=')'&&ip_sym[i]!='i'&&ip_sym[i]!='I')
 {
  printf("\nSyntax error");
  break;
 }
 for(i=0;i<=strlen(op);i++)
   if(op[i]!='e')
tmp[n++]=op[i];
   strcpy(op,tmp);
   printf("\nE=%-25s",op);
}
 getch();
}
```

# OUTPUT:

```
Grammar without left recursion
                E->TE'
                E'->+TE'|e
                T->FT'
                T'->*FT'|e
                F->(E)|i
 Enter the input expression:i+i*i
```

```
 Expressions      Sequence of production rules
 E=TE'                      E->TE'
 E=FT'E'                    T->FT'
 E=iT'E'                    F->i
 E=ieE'                     T'->e
 E=i+TE'                    E'->+TE'
 E=i+FT'E'                  T->FT'
 E=i+iT'E'                  F->i
 E=i+i*FT'E'                T'->*FT'
 E=i+i*iT'E'                F->i
 E=i+i*ieE'                 T'->e
 E=i+i*ie                   E'->e
 E=i+i*i
 f:\abhinav chaturvedi\compiler_design\assignment_7>
```

# EXPERIMENT – 8

Q: Write a program for generating various intermediate code forms.

i) Three address code
ii) Polish notation

Ans:

Source code:

```
#include <stdio.h>
#include <string.h>
void pm();
void plus();
void div();
int i, ch, j, l, addr = 100;
char ex[10], exp[10], exp1[10], exp2[10], id1[5], op[5], id2[5];
void main()
{
    // clrscr();
    while (1)
    {
        printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the
choice:");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("\nEnter the expression with assignment operator:");
            scanf("%s", exp);
            l = strlen(exp);
            exp2[0] = '\0';
            i = 0;
            while (exp[i] != '=')
            {
                i++;
            }
            strncat(exp2, exp, i);
            strrev(exp);
            exp1[0] = '\0';
            strncat(exp1, exp, l - (i + 1));
            strrev(exp1);
            printf("Three address code:\ntemp=%s\n%s=temp\n", exp1, exp2);
            break;
```

```
        case 2:
            printf("\nEnter the expression with arithmetic operator:");
            scanf("%s", ex);
            strcpy(exp, ex);
            l = strlen(exp);
            exp1[0] = '\0';

            for (i = 0; i < l; i++)
            {
                if (exp[i] == '+' || exp[i] == '-')
                {
                    if (exp[i + 2] == '/' || exp[i + 2] == '*')
                    {
                        pm();
                        break;
                    }
                    else
                    {
                        plus();
                        break;
                    }
                }
                else if (exp[i] == '/' || exp[i] == '*')
                {
                    div();
                    break;
                }
            }
            break;

        case 3:
            printf("Enter the expression with relational operator");
            scanf("%s%s%s", &id1, &op, &id2);
            if (((strcmp(op, "<") == 0) || (strcmp(op, ">") == 0) ||
(strcmp(op, "<=") == 0) || (strcmp(op, ">=") == 0) || (strcmp(op, "==") == 0)
|| (strcmp(op, "!=") == 0)) == 0)
                printf("Expression is error");
            else
            {
                printf("\n%d\tif %s%s%s goto %d", addr, id1, op, id2, addr +
3);
                addr++;
                printf("\n%d\t T:=0", addr);
                addr++;
                printf("\n%d\t goto %d", addr, addr + 2);
                addr++;
                printf("\n%d\t T:=1", addr);
            }
```

```
            break;
        case 4:
            exit(0);
        }
    }
}
void pm()
{
    strrev(exp);
    j = l - i - 1;
    strncat(exp1, exp, j);
    strrev(exp1);
    printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n", exp1, exp[j + 1],
exp[j]);
}
void div()
{
    strncat(exp1, exp, i + 2);
    printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n", exp1, exp[i + 2],
exp[i + 3]);
}
void plus()
{
    strncat(exp1, exp, i + 2);
    printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n", exp1, exp[i + 2],
exp[i + 3]);
}
```

## OUTPUT:

```
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2

Enter the expression with arithmetic operator:a+b-c
Three address code:
temp=a+b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4

F:\abhinav chaturvedi\compiler_design\assignment_8>
```

# EXPERIMENT – 9

Q: Write a program to simulate Heap storage allocation strategy.

Ans:

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0
typedef struct Heap
{
    int data;
    struct Heap *next;
} node;
node *create();
void main(){
    int choice, val;
    char ans;
    node *head;
    void display(node *);
    node *search(node *, int);
    node *insert(node *);
    void dele(node **);
    head = NULL;
    do
    {
        printf("\nprogram to perform various operations on heap using dynamic
memory management");
        printf("\n1.create");
        printf("\n2.display");
        printf("\n3.insert an element in a list");
        printf("\n4.delete an element from list");
        printf("\n5.quit");
        printf("\nenter your chioce(1-5): ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            head = create();
            break;
        case 2:
            display(head);
            break;
```

```
        case 3:
            head = insert(head);
            break;
        case 4:
            dele(&head);
            break;
        case 5:
            exit(0);
        default:
            printf("invalid choice,try again");
        }
    } while (choice != 5);
}
node *create()
{
    node *temp, *New, *head;
    int val, flag;
    char ans = 'y';
    node *get_node();
    temp = NULL;
    flag = TRUE;
    do
    {
        printf("\n enter the element:");
        scanf("%d", &val);
        New = get_node();
        if (New == NULL)
            printf("\nmemory is not allocated");
        New->data = val;
        if (flag == TRUE)
        {
            head = New;
            temp = head;
            flag = FALSE;
        }
        else
        {
            temp->next = New;
            temp = New;
        }
        printf("\ndo you want to enter more elements?(y/n)");
    } while (ans == 'y');
    printf("\nthe list is created\n");
    return head;
}
node *get_node()
{
    node *temp;
```

```
    temp = (node *)malloc(sizeof(node));
    temp->next = NULL;
    return temp;
}
void display(node *head)
{
    node *temp;
    temp = head;
    printf("\n");
    if (temp == NULL)
    {
        printf("\nthe list is empty\n");
        return;
    }
    while (temp != NULL)
    {
        printf("%d->", temp->data);
        temp = temp->next;
    }
    printf("NULL");
    printf("\n");

}
node *search(node *head, int key)
{
    node *temp;
    int found;
    temp = head;
    if (temp == NULL)
    {
        printf("the linked list is empty\n");
        return NULL;
    }
    found = FALSE;
    while (temp != NULL && found == FALSE)
    {
        if (temp->data != key)
            temp = temp->next;
        else
            found = TRUE;
    }
    if (found == TRUE)
    {
        printf("\nthe element is present in the list\n");
        return temp;
    }
    else
    {
```

```
        printf("the element is not present in the list\n");
        return NULL;
    }
}
node *insert(node *head)
{
    int choice;
    node *insert_head(node *);
    void insert_after(node *);
    void insert_last(node *);
    printf("\n1. insert a node as a head node");
    printf("\n2. insert a node as a last node");
    printf("\n3. insert a node at intermediate position in the list");
    printf("\nenter your choice for insertion of node: ");
    scanf("%d", &choice);
    switch (choice)
    {
    case 1:
        head = insert_head(head);
        break;
    case 2:
        insert_last(head);
        break;
    case 3:
        insert_after(head);
        break;
    }
    return head;
}
node *insert_head(node *head)
{
    node *New, *temp;
    New = get_node();
    printf("\nEnter the element which you want to insert: ");
    scanf("%d", &New->data);
    if (head == NULL)
        head = New;
    else
    {
        temp = head;
        New->next = temp;
        head = New;
    }
    return head;
}
void insert_last(node *head)
{
    node *New, *temp;
```

```
    New = get_node();
    printf("\nenter the element which you want to insert");
    scanf("%d", &New->data);
    if (head == NULL)
        head = New;
    else
    {
        temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = New;
        New->next = NULL;
    }
}
void insert_after(node *head)
{
    int key;
    node *New, *temp;
    New = get_node();
    printf("\nenter the elements which you want to insert");
    scanf("%d", &New->data);
    if (head == NULL)
    {
        head = New;
    }
    else
    {
        printf("\enter the element which you want to insert the node");
        scanf("%d", &key);
        temp = head;
        do
        {
            if (temp->data == key)
            {
                New->next - temp->next;
                temp->next = New;
                return;
            }
            else
                temp = temp->next;
        } while (temp != NULL);
    }
}
node *get_prev(node *head, int val)
{
    node *temp, *prev;
    int flag;
    temp = head;
```

```
    if (temp == NULL)
        return NULL;
    flag = FALSE;
    prev = NULL;
    while (temp != NULL && !flag)
    {
        if (temp->data != val)
        {
            prev = temp;
            temp = temp->next;
        }
        else
            flag = TRUE;
    }
    if (flag)
        return prev;
    else
        return NULL;
}
void dele(node **head)
{
    node *temp, *prev;
    int key;
    temp = *head;
    if (temp == NULL)
    {
        printf("\nthe list is empty\n");
        return;
    }
    printf("\nenter the element you want to delete:");
    scanf("%d", &key);
    temp = search(*head, key);
    if (temp != NULL)
    {
        prev = get_prev(*head, key);
        if (prev != NULL)
        {
            prev->next = temp->next;
            free(temp);
        }
        else
        {
            *head = temp->next;
            free(temp);
        }
        printf("\nthe element is deleted\n");
    }
}
```

# OUTPUT:

```
program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your chioce(1-5): 2


the list is empty
```

```
program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your chioce(1-5): 3

1. insert a node as a head node
2. insert a node as a last node
3. insert a node at intermediate position in the list
enter your choice for insertion of node: 1

Enter the element which you want to insert: 10

program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your chioce(1-5): 3

1. insert a node as a head node
2. insert a node as a last node
3. insert a node at intermediate position in the list
enter your choice for insertion of node: 2

enter the element which you want to insert20
```

```
program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your chioce(1-5): 2

10->20->NULL

program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your chioce(1-5): 5

F:\abhinav chaturvedi\compiler_design\assignment_9>
```

# EXPERIMENT – 10

Q: Generate Lexical analyzer using LEX.

Ans:

Source code:

OUTPUT:

# EXPERIMENT – 11

Q: Generate YACC specification for a few syntactic categories.

Ans:

Source code:

OUTPUT:

# EXPERIMENT – 12

Q: Given any intermediate code form implement code optimization techniques.

Ans:

Source code:

```
#include <stdio.h>
#include <string.h>
struct op
{
    char l;
    char r[20];
} op[10], pr[10];
void main()
{
    int a, i, k, j, n, z = 0, m, q;
    char *p, *l;
    char temp, t;
    char *tem;
    printf("Enter the Number of Values:");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("left: ");
        scanf(" %c", &op[i].l);
        printf("right: ");
        scanf(" %s", &op[i].r);
    }
    printf("Intermediate Code\n");
    for (i = 0; i < n; i++)
    {
        printf("%c=", op[i].l);
        printf("%s\n", op[i].r);
    }
    for (i = 0; i < n - 1; i++)
    {
        temp = op[i].l;
        for (j = 0; j < n; j++)
        {
            p = strchr(op[j].r, temp);
            if (p)
            {
                pr[z].l = op[i].l;
                strcpy(pr[z].r, op[i].r);
```

```
                z++;
            }
        }
    }
    pr[z].l = op[n - 1].l;
    strcpy(pr[z].r, op[n - 1].r);
    z++;
    printf("\nAfter Dead Code Elimination\n");
    for (k = 0; k < z; k++)
    {
        printf("%c\t=", pr[k].l);
        printf("%s\n", pr[k].r);
    }
    for (m = 0; m < z; m++)
    {
        tem = pr[m].r;
        for (j = m + 1; j < z; j++)
        {
            p = strstr(tem, pr[j].r);
            if (p)
            {
                t = pr[j].l;
                pr[j].l = pr[m].l;
                for (i = 0; i < z; i++)
                {
                    l = strchr(pr[i].r, t);
                    if (l)
                    {
                        a = l - pr[i].r;
                        printf("pos: %d\n", a);
                        pr[i].r[a] = pr[m].l;
                    }
                }
            }
        }
    }
    printf("Eliminate Common Expression\n");
    for (i = 0; i < z; i++)
    {
        printf("%c\t=", pr[i].l);
        printf("%s\n", pr[i].r);
    }
    for (i = 0; i < z; i++)
    {
        for (j = i + 1; j < z; j++)
        {
            q = strcmp(pr[i].r, pr[j].r);
            if ((pr[i].l == pr[j].l) && !q)
```

```
            {
                pr[i].l = '\0';
            }
        }
    }
    printf("Optimized Code\n");
    for (i = 0; i < z; i++)
    {
        if (pr[i].l != '\0')
        {
            printf("%c=", pr[i].l);
            printf("%s\n", pr[i].r);
        }
    }
}
```

## OUTPUT:

```
&& "f:\abhinav chaturvedi\compiler_design\assignment_12\"assignment_12
Enter the Number of Values:5
left: a
right: 9
left: b
right: c+d
left: e
right: c+d
left: f
right: b+e
left: r
right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f
```

```
After Dead Code Elimination
b       =c+d
e       =c+d
f       =b+e
r       =f
pos: 2
Eliminate Common Expression
b       =c+d
b       =c+d
f       =b+b
r       =f
Optimized Code
b=c+d
f=b+b
r=f
```

# EXPERIMENT - 13

Q: Study of an Object-Oriented Compiler.

Ans: A compiler takes a program in a source language, creates some internal representation while checking the syntax of the program, performs semantic checks, and finally generates something that can be executed to produce the intended effect of the program.

The obvious candidate for object technology in a compiler is the symbol table: a mapping from user-defined names to their properties as expressed in the program.

If the internal representation is a tree of objects, semantic checking and generation can be accomplished by sending a message to these objects or by visiting each object. If the result of generation is a set of persistent objects, program execution can consist of sending a message to a distinguished object in this set.

A parser-generator takes a grammar, specified in a language such as BNF or EBNF, checks it, and constructs a representation (the parser) that will execute semantic actions as phrases over the grammar is recognized.

Symbol table and descriptions are obvious candidates for OOP within a compiler: a table is a container object where each description object is held. The descriptions share at least the ability to be located by key. If a base class is used to hold the key, inheritance helps to encapsulate and share the lookup mechanism while keeping it separate from the information constituting the actual description

Using OOP a compiler can transform a program source into a tree of persistent objects as an image. Execution is accomplished by sending a message to the root node of that tree which results in partial traversal as the message is passed along the tree.

Semantic analysis decides if a syntactically acceptable program is meaningful. For a large part it is concerned with the interaction of various data types in expressions: during a postorder traversal of the parse tree, result types are computed for each part of an expression and stored in the nodes for the benefit of code generation. Parse tree nodes are objects, their classes must implement a method to perform semantic checking of the node. If necessary, the method can augment the parse tree with conversion nodes. Types are modeled as unique objects. They have methods informing the semantic analysis about available operations for the type and about permissible interaction with other types. Other type methods generate a simplified, persistent runtime tree which can be used as an interpreter or traversed for code generation.