Register No. : **Name :**

Ex. No : 1 **CONVERSION OF NON-DETERMINISTIC FINITE AUTOMATON (NFA)**

Date : **TO DETERMINISTIC FINITE AUTOMATON (DFA)**

AIM

To write a C program for Conversion of Non-Deterministic Finite Automaton (NFA) To Deterministic Finite Automaton (DFA).

ALGORITHM

Step 1 : Take ∈ closure for the beginning state of NFA as beginning state of DFA.

Step 2 : Find the states that can be traversed from the present for each input symbol (union of transition value and their closures for each states of NFA present in current state of DFA).

Step 3 : If any new state is found take it as current state and repeat step 2.

Step 4 : Do repeat Step 2 and Step 3 until no new state present in DFA transition table.

Step 5 : Mark the states of DFA which contains final state of NFA as final states of DFA.

PROGRAM

```
// C Program to illustrate how to convert e-nfa to DFA

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

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

// Structure to store DFA states and their
// status ( i.e new entry or already present)
struct DFA {
  char *states;
  int count;
} dfa;

int last_index = 0;
FILE *fp;
```

```
int symbols;

/* reset the hash map*/
void reset(int ar[], int size) {
  int i;

  // reset all the values of
  // the mapping array to zero
  for (i = 0; i < size; i++) {
    ar[i] = 0;
  }
}

// Check which States are present in the e-closure

/* map the states of NFA to a hash set*/
void check(int ar[], char S[]) {
  int i, j;

  // To parse the individual states of NFA
  int len = strlen(S);
  for (i = 0; i < len; i++) {

    // Set hash map for the position
    // of the states which is found
    j = ((int)(S[i]) - 65);
    ar[j]++;
  }
}

// To find new Closure States
void state(int ar[], int size, char S[]) {
  int j, k = 0;

  // Combine multiple states of NFA
  // to create new states of DFA
  for (j = 0; j < size; j++) {
    if (ar[j] != 0)
      S[k++] = (char)(65 + j);
  }

  // mark the end of the state
  S[k] = '\0';
}

// To pick the next closure from closure set
```

```
int closure(int ar[], int size) {
  int i;

  // check new closure is present or not
  for (i = 0; i < size; i++) {
    if (ar[i] == 1)
      return i;
  }


  return (100);
}

// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
  int i;

  for (i = 0; i < last_index; i++) {
    if (dfa[i].count == 0)
      return 1;
  }
  return -1;
}

/* To Display epsilon closure*/
void Display_closure(int states, int closure_ar[],
                     char *closure_table[],
                     char *NFA_TABLE[][symbols + 1],
                     char *DFA_TABLE[][symbols]) {
  int i;
  for (i = 0; i < states; i++) {
    reset(closure_ar, states);
    closure_ar[i] = 2;

    // to neglect blank entry
    if (strcmp(&NFA_TABLE[i][symbols], "-") != 0) {

      // copy the NFA transition state to buffer
      strcpy(buffer, &NFA_TABLE[i][symbols]);
      check(closure_ar, buffer);
      int z = closure(closure_ar, states);

      // till closure get completely saturated
      while (z != 100)
      {
```

```
        if (strcmp(&NFA_TABLE[z][symbols], "-") != 0) {
          strcpy(buffer, &NFA_TABLE[z][symbols]);

          // call the check function
          check(closure_ar, buffer);
        }
        closure_ar[z]++;
        z = closure(closure_ar, states);
      }
    }

    // print the e closure for every states of NFA
    printf("\n e-Closure (%c) :\t", (char)(65 + i));

    bzero((void *)buffer, MAX_LEN);
    state(closure_ar, states, buffer);
    strcpy(&closure_table[i], buffer);
    printf("%s\n", &closure_table[i]);
  }
}

/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {

  int i;

  // To check the current state is already
  // being used as a DFA state or not in
  // DFA transition table
  for (i = 0; i < last_index; i++) {
    if (strcmp(&dfa[i].states, S) == 0)
      return 0;
  }

  // push the new
  strcpy(&dfa[last_index++].states, S);

  // set the count for new states entered
  // to zero
  dfa[last_index - 1].count = 0;
  return 1;
}

// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int st,
```

```
            char *NFT[][symbols + 1], char TB[]) {
  int len = strlen(S);
  int i, j, k, g;
  int arr[st];
  int sz;
  reset(arr, st);
  char temp[MAX_LEN], temp2[MAX_LEN];
  char *buff;

  // Transition function from NFA to DFA
  for (i = 0; i < len; i++) {

    j = ((int)(S[i] - 65));
    strcpy(temp, &NFT[j][M]);

    if (strcmp(temp, "-") != 0) {
      sz = strlen(temp);
      g = 0;

      while (g < sz) {
        k = ((int)(temp[g] - 65));
        strcpy(temp2, &clsr_t[k]);
        check(arr, temp2);
        g++;
      }
    }
  }

  bzero((void *)temp, MAX_LEN);
  state(arr, st, temp);
  if (temp[0] != '\0') {
    strcpy(TB, temp);
  } else
    strcpy(TB, "-");
}

/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA *dfa_states,
              char *DFA_TABLE[][symbols]) {
  int i, j;
  printf("\n\n*****************************************************\n\n");
  printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");
  printf("\n STATES OF DFA :\t\t");

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

```
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR DFA: \t");

  for (i = 0; i < symbols; i++)
    printf("%d, ", i);
  printf("\n\n");
  printf("STATES\t");

  for (i = 0; i < symbols; i++)
    printf("|%d\t", i);
  printf("\n");

  // display the DFA transition state table
  printf("--------+----------------------\n");
  for (i = 0; i < zz; i++) {
    printf("%s\t", &dfa_states[i + 1].states);
    for (j = 0; j < symbols; j++) {
      printf("|%s \t", &DFA_TABLE[i][j]);
    }
    printf("\n");
  }
}



// Driver Code
int main() {
  int i, j, states;
  char T_buf[MAX_LEN];

  // creating an array dfa structures
  struct DFA *dfa_states = malloc(MAX_LEN * (sizeof(dfa)));
  states = 6, symbols = 2;

  printf("\n STATES OF NFA :\t\t");
  for (i = 0; i < states; i++)

    printf("%c, ", (char)(65 + i));
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR NFA: \t");

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

    printf("%d, ", i);
  printf("eps");
  printf("\n\n");
```

```
char *NFA_TABLE[states][symbols + 1];

// Hard coded input for NFA table
char *DFA_TABLE[MAX_LEN][symbols];
strcpy(&NFA_TABLE[0][0], "FC");
strcpy(&NFA_TABLE[0][1], "-");
strcpy(&NFA_TABLE[0][2], "BF");
strcpy(&NFA_TABLE[1][0], "-");
strcpy(&NFA_TABLE[1][1], "C");
strcpy(&NFA_TABLE[1][2], "-");
strcpy(&NFA_TABLE[2][0], "-");
strcpy(&NFA_TABLE[2][1], "-");
strcpy(&NFA_TABLE[2][2], "D");
strcpy(&NFA_TABLE[3][0], "E");
strcpy(&NFA_TABLE[3][1], "A");
strcpy(&NFA_TABLE[3][2], "-");
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE \n\n\n");
printf("STATES\t");

for (i = 0; i < symbols; i++)
  printf("|%d\t", i);
printf("eps\n");
 // Displaying the matrix of NFA transition table
printf("--------+-----------------------------------\n");
for (i = 0; i < states; i++) {
  printf("%c\t", (char)(65 + i));
   for (j = 0; j <= symbols; j++) {
    printf("|%s \t", &NFA_TABLE[i][j]);
  }
  printf("\n");
}
int closure_ar[states];
char *closure_table[states];
 Display_closure(states, closure_ar, closure_table, NFA_TABLE, DFA_TABLE);
strcpy(&dfa_states[last_index++].states, "-");
 dfa_states[last_index - 1].count = 1;
bzero((void *)buffer, MAX_LEN);
 strcpy(buffer, &closure_table[0]);
strcpy(&dfa_states[last_index++].states, buffer);
 int Sm = 1, ind = 1;
```

```
  int start_index = 1;
   // Filling up the DFA table with transition values
  // Till new states can be entered in DFA table
  while (ind != -1) {
    dfa_states[start_index].count = 1;
    Sm = 0;
    for (i = 0; i < symbols; i++) {

      trans(buffer, i, closure_table, states, NFA_TABLE, T_buf);

      // storing the new DFA state in buffer
      strcpy(&DFA_TABLE[zz][i], T_buf);

      // parameter to control new states
      Sm = Sm + new_states(dfa_states, T_buf);
    }
    ind = indexing(dfa_states);
    if (ind != -1)
      strcpy(buffer, &dfa_states[++start_index].states);
    zz++;
  }
  // display the DFA TABLE
  Display_DFA(last_index, dfa_states, DFA_TABLE);

  return 0;
}
```

INPUT : 6
2
FC - BF
- C -
- - D
E A -
A - BF
- - -

OUTPUT

```
STATES OF NFA :       A, B, C, D, E, F,

GIVEN SYMBOLS FOR NFA:     0, 1, eps

NFA STATE TRANSITION TABLE

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

e-Closure (A) :    ABF

e-Closure (B) :    B

e-Closure (C) :    CD

e-Closure (D) :    D

e-Closure (E) :    BEF

e-Closure (F) :    F

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

      DFA TRANSITION STATE TABLE

STATES OF DFA :       ABF, CDF, CD, BEF,

GIVEN SYMBOLS FOR DFA:     0, 1,

STATES   |0    |1
--------+----------------------
ABF    |CDF     |CD
CDF    |BEF     |ABF
CD    |BEF     |ABF
BEF    |ABF     |CD
```

RESULT

Program to convert NFA to DFA has been implemented.

Ex.No : 2 **IMPLEMENTATION OF SYMBOL TABLE**
Date :

AIM

To write a C program to implement a symbol table.

ALGORITHM

1. Start the program.
2. Get the input from the user with the terminating symbol '$'.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading, the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till '$' is reached.
7. To reach a variable, enter the variable to be searched and symbol table has been checked for corresponding variable, the variable along with its address is displayed as result.
8. Stop the program.

PROGRAM

```
 #include<stdio.h>
 #include<conio.h>
#include<ctype.h>
#include<alloc.h>
#include<string.h>
#include<math.h>
void main()

{

int i=0,j=0,x=0,n,flag=0; void *p,*add[5];
char ch,srch,b[15],d[15],c; clrscr();
printf("Enter the Expression terminated by $:");
while((c=getchar())!='$')

{

b[i]=c; i++;
}

n=i-1;

printf("Given Expression:"); i=0;
```

```
while(i<=n)

{

printf("%c",b[i]); i++;
}

printf("\n Symbol Table\n"); printf("Symbol\taddr\ttype"); while(j<=n)
{

c=b[j]; if(isalpha(toascii(c)))
{

if(j==n)

{

p=malloc(c); add[x]=p;
d[x]=c;
printf("%c\t%d\tidentifier",c,p);

}

else

{

ch=b[j+1];

if(ch=='+'||ch=='-'||ch=='*'||ch=="=")

{

p=malloc(c); add[x]=p;
d[x]=c; printf("\n%c\t%d\tidentifier\n",c,p); x++;
}}} j++;
}

printf("\n The symbol is to be searched"); srch=getch();
for(i=0;i<=x;i++)

{

if(srch==d[i])

{
```

```
printf("\n Symbol Found"); printf("\n%c%s%d\n",srch,"@address",add[i]); flag=1;
}
}

if(flag==0)

printf("\nSymbol Not Found"); getch();
}
```

OUTPUT

```
Enter the Expression terminated by $ : A+B*C$ Given Expression : A+B*C
Symbol Table

Symbol  addr type A     1900            identifier
1970    identifier

2040    identifier

The symbol is to be searched B Symbol Found
B @address 1970
```

RESULT

The program to implement a symbol table is executed and the output is verified.

Register No. : **Name :**

Ex. No: 3 **GENERATION OF LEXICAL TOKENS USING C**
Date :

AIM

To write a C program to implement lexical analyzer to recognize a few patterns.

ALGORITHM

1) Start the program.
2) Get the input from the user with the terminating symbol ';'.
3) Until the symbol is ';', do the following:
    a. If the next character of the symbol is an operator then print it as an operator.
    b. If the next character of the symbol is a variable, then insert it into symbol table and print it as an identifier.
    c. If the next character of the symbol is a bracket, then print it as parenthesis.
4) Stop the program.

PROGRAM

```
/*Lexical Analyser*/


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

#define SIZE 128
#define NONE -1
#define EOS '\0'
#define NUM 256
#define KEYWORD 257
#define PAREN_OPEN 296
#define PAREN_CLOSE 297
#define ID 259
#define ASSIGN 317
#define DONE 262
#define MAX 999

char lexemes[MAX];
char buffer[SIZE];
int lastchar=-1;
```

```
int lastentry=0;
int tokenval=NONE;

int lineno=1;


struct entry

{

        char *lexptr;

        int token;

}symtable[100];


struct entry
keywords[]={"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,"int",KEYWORD,"float",
KEYWORD,"double",KEYWORD,"char",KEYWORD,"struct",KEYWORD,"return",KEYWO
RD,0,0};

void errormsg(char *m)

{

        fprintf(stderr,"line %d:%s\n",lineno,m);

        exit(1);

}
int lookup(char s[])

{

        int k;

        for(k=lastentry;k>0;k=k-1)

                if(strcmp(symtable[k].lexptr,s)==0)

                        return k;
```

```
        return 0;
}
int insert(char s[],int tok)
{
        int len;
        len=strlen(s);
        if(lastentry+1>=MAX)
                errormsg("symtable is full");
        if(lastentry+len+1>=MAX)
                errormsg("lexemes array is full");
        lastentry=lastentry+1;
        symtable[lastentry].token=tok;
        symtable[lastentry].lexptr=&lexemes[lastchar+1];
        lastchar=lastchar+len+1;
        strcpy(symtable[lastentry].lexptr,s);
        return lastentry;
}
void initialise()
{
        struct entry *ptr;
        for(ptr=keywords;ptr->token;ptr++)
                insert(ptr->lexptr,ptr->token);
}
```

```
int lexer()
{
        int t;
        int val,i=0;
        while(1)
        {
                t=getchar();
                if(t==' '||t=='\t');
                else if(t=='\n')
                        lineno=lineno+1;
                else if(t=='(')
                        return PAREN_OPEN;
                else if(t==')')
                        return PAREN_CLOSE;
                else if(isdigit(t))
                {
                        ungetc(t,stdin);
                        scanf("%d",&tokenval);
                        return NUM;
                }
                else if(isalpha(t))
                {
                        while(isalnum(t))
```

```
                {
                        buffer[i]=t;
                        t=getchar();
                        i=i+1;
                        if(i>=SIZE)
                                errormsg("compiler error");
                }
                buffer[i]=EOS;
                if(t!=EOF)
                        ungetc(t,stdin);
                val=lookup(buffer);
                if(val==0)
                        val=insert(buffer,ID);
                tokenval=val;
                return symtable[val].token;
        }
        else if(t==EOF)
                return DONE;
        else    {
            tokenval := NONE;
        }
        return t;
    }
}

void  main()
```

```
{
    char ans;
    int lookahead;
    clrscr();
    printf("\n \t \t Program for lexical analysis \n\n");
    initialise();
    printf("Enter the expression & place;at the end \n ");
    lookahead=lexer();
    while(lookahead!=DONE)
    {
        if(lookahead==NUM)
        {
            printf("\n Number :%d",tokenval);
        }
        if(lookahead=='+'||lookahead=='-'||lookahead=='*'||lookahead=='/')
        {
            printf("\n Operator:%c",lookahead);
        }
        if(lookahead==PAREN_OPEN)
            printf("\n Parenthesis:%c",lookahead);
        if(lookahead==PAREN_CLOSE)
            printf("\n Parenthesis:%c",lookahead);
        if(lookahead==ID)
```

```
                printf("\n Identifier:%s",symtable[tokenval].lexptr);
            if(lookahead==KEYWORD)
                printf("\n Keyword");
            if(lookahead==ASSIGN)
                printf("\nAssignment Operator:%c",lookahead);
            if(lookahead=='<'||lookahead=='>'||lookahead=='<='||lookahead=='>='||
lookahead=='!=')
                printf("\nRelational Operator");
            lookahead=lexer();
        }
        getch();
}
```

OUTPUT

Program for lexical analysis

Enter the expression & place;at the end

a+(b*c);

Identifier:a

Operator:+

Parenthesis:(

Identifier:b

Operator:*

Identifier:c

Parenthesis:)

RESULT

The program to implement lexical analyzer is executed and the output is verified.

Register No. : **Name :**

Ex.No: 4 **GENERATION OF LEXICAL TOKENS LEX/FLEX TOOL**

Date :

AIM

To write a lex program to implement lexical analyzer to recognize a few patterns.

ALGORITHM

1. Start the program.
2. Lex program consists of three parts.
    a. Declaration %%
    b. Translation rules %%
    c. Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Translation rule of lex program are statements of the form
    a. P1 {action}
    b. P2 {action}
    c. …
    d. …
    e. Pn {action}
5. Write a program in the vi editor and save it with .l extension.
6. Compile the lex program with lex compiler to produce output file as lex.yy.c.

    eg $ lex filename.l $ cc lex.yy.c
7. Compile that file with C compiler and verify the output.

Implement the Lexical Analyzer Using LEX Tool.

```
/* program name is lexp.l */
%{
/* program to recognize a c program */
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
```

```
"*/" {COMMENT = 0;}
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file = fopen(argv[1],"r");
if(!file)
{
printf("could not open %s \n",argv[1]);
exit(0);
}
yyin = file;
}
yylex();
printf("\n\n");
```

```
return 0;
} int yywrap()
{
return 0;
}
```

INPUT

```
$vi var.c
#include<stdio.h>
main()
{
int a,b;
}
```

OUTPUT

```
$lex lex.l
$cc lex.yy.c
$./a.out var.c
#include<stdio.h> is a PREPROCESSOR DIRECTIVE
FUNCTION
main (
)
BLOCK BEGINS
int is a KEYWORD
a IDENTIFIER
b IDENTIFIER
BLOCK ENDS
```

RESULT

The lexical analyzer is implemented using lex and the output is verified.

Register No. : **Name :**

Ex.No: 5 **RECOGNITION OF A VALID ARITHMETIC EXPRESSION THAT USES**

Date : **OPERATOR +, -, * AND / USING YACC**

AIM

To write a yacc program to recognize a valid arithmetic expression that uses operator +,-,* and /.

ALGORITHM

1. Start the program.
2. Write a program in the vi editor and save it with .l extension.
3. In the lex program, write the translation rules for the operators =,+,-,*,/ and for the identifier.
4. Write a program in the vi editor and save it with .y extension.
5. Compile the lex program with lex compiler to produce output file as lex.yy.c.
   eg $ lex filename.l
6. Compile the yacc program with yacc compiler to produce output file as y.tab.c.
   eg $ yacc –d arith_id.y
7. Compile these with the C compiler as

   gcc lex.yy.c y.tab.c
8. Enter an arithmetic expression as input and the tokens are identified as output.

PROGRAM

Program name: arith_id.l

```
%{
/* This LEX program returns the tokens for the expression */
#include "y.tab.h"
%}
%%
"=" {printf("\n Operator is EQUAL");}
"+" {printf("\n Operator is PLUS");}
"-" {printf("\n Operator is MINUS");}
"/" {printf("\n Operator is DIVISION");}
"*" {printf("\n Operator is MULTIPLICATION");}
[a-zA-Z]*[0-9]* {
printf("\n Identifier is %s",yytext);
return ID; }
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
```

Program Name : arith_id.y

```
%{
```

```
#include<stdio.h>
/* This YACC program is for recognizing the Expression */
%}
%token A ID
%%
statement: A'='E
| E {
printf("\n Valid arithmetic expression");
$$=$1;
}
;
E: E'+'ID
| E'-'ID
| E'*'ID
| E'/'ID
| ID
;
%%
extern FILE*yyin;
main() {
do {
yyparse();
}while(!feof(yyin)); }
```

```
yyerror(char*s)
{
}
```

OUTPUT

```
[root@localhost]# lex arith_id.1
[root@localhost]# yacc –d arith_id.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
x=a+b;
Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```

RESULT

A YACC program to recognize a valid arithmetic expression that uses operator +,-,*and / is executed successfully and the output is verified.

Register No. : **Name :**

Ex.No: 6 **RECOGNITION OF A VALID VARIABLE WHICH STARTS WITH A**
Date : **LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS USING YACC**

AIM

To write a YACC program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

ALGORITHM

1. Start the program.
2. Write a program in the vi editor and save it with .l extension.
3. In the lex program, write the translation rules for the keywords int, float and double and for the identifier.
4. Write a program in the vi editor and save it with .y extension.
5. Compile the lex program with lex compiler to produce output file as lex.yy.c.
   eg $ lex filename.l
6. Compile the yacc program with YACC compiler to produce output file as
   y.tab.c.eg $ yacc –d arith_id.y
7. Compile these with the C compiler as

   gcc lex.yy.c y.tab.c
8. Enter a statement as input and the valid variables are identified as output.

PROGRAM

Program name: variable_test.l

```
%{
/* This LEX program returns the tokens for the Expression */
#include"y.tab.h"
%}
%%
"int" {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]* {printf("\nIdentifier is %s",yytext);
return ID;
}
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
```

Program name: variable_test.y

```
%{
#include<stdio.h>
/* This YACC program is for recognising the Expression*/
```

```
%}
%token ID INT FLOAT DOUBLE

%%

D: T L

;

L: L,ID

| ID

;

T: INT

| FLOAT

| DOUBLE

;

%%

extern FILE*yyin;

main()

{

do

{

yyparse();
}while(!feof(yyin));

}
yyerror(char*s)
{

}
```

OUTPUT

```
[root@localhost]# lex variable_test.I
[root@localhost]# yacc –d variable_test.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
int a,b;
Identifier is a
Identifier is b
```

RESULT

A YACC program to recognize a valid variable which starts with a letter followed by any number of letters or digits is executed successfully and the output is verified.

Register No. : **Name :**

Ex.No: 7 **CONVERSION OF THE BNF RULES INTO YACC FORM AND**

Date : **GENERATION OF ABSTRACT SYNTAX TREE**

AIM

To write a program to convert the BNF rules into YACC form and to generate Abstract Syntax Tree.

ALGORITHM

1. Start the program.
2. Write a program in the vi editor and save it with .l extension.
3. In the lex program, write the translation rules for the keywords, identifiers and relational operators.
4. Write a program in the vi editor and save it with .y extension.
5. In the YACC program, the following tasks are performed:
   a. Reading an input file line by line.
   b. Convert it into abstract syntax tree using three address code.
   c. Represent three address code in the form of quadruple tabular form.
6. Compile the lex program with lex compiler to produce output file as lex.yy.c.
   eg $ lex filename.l
7. Compile the yacc program with yacc compiler to produce output file as y.tab.c.
   eg $ yacc –d arith_id.y
8. Compile these with the C compiler as

   gcc lex.yy.c y.tab.c
9. A C program is given as input and the abstract syntax tree is generated as output.

PROGRAM

<int.l>

```
%{
#include"y.t
ab.h"
#include<st
dio.h>
#include<str
ing.h>int
LineNo=1;
%}
 identifier [a-zA-Z][_a-zA-
Z0-9]*number [0-9]+|([0-
9]*\.[0-9]+)
%%
main\(\) return
MAIN;if return
IF;
else return
ELSE; while
return
WHILE;int |
char |
float return TYPE;
```

{identifier}

{strcpy(yylval.var,yytext);

return VAR;}

{number}

{strcpy(yylval.var,yytext);

return NUM;}

< |> |>= |<= |==

{strcpy(yylval.var,yytext);return

RELOP;}

[\t] ;
\n LineNo++;

. return yytext[0];

%%

<**int.y**>

%{

#include<stri

ng.h>

#include<std

io.h> struct

quad{

char

op[5];

char

arg1[10]

```
; char
arg2[10]
; char
result[10
];
}QUAD[30];
struct
stack{
int
items[1
00];int
top;
}stk;
int
Index=0,tIndex=0,StNo,Ind,tI
nd;extern int LineNo;
%}
%unio
n{
char
var[10
];
}
%token <var> NUM VAR RELOP
```

```
%token MAIN IF ELSE WHILE TYPE

%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+' %left '*' '/'

%%

PROGRAM : MAIN BLOCK

;

BLOCK: '{' CODE '}'

;

CODE: BLOCK

| STATEMENT CODE

| STATEMENT

;

STATEMENT: DESCT ';'

| ASSIGNMENT ';'

| CONDST

| WHILEST

;

DESCT: TYPE VARLIST

;

VARLIST: VAR ',' VARLIST

| VAR

;

ASSIGNMENT: VAR '=' EXPR{
```

```
strcpy(QUAD[Index].op,
"=");
strcpy(QUAD[Index].arg
1,$3);
strcpy(QUAD[Index].arg
2,"");
strcpy(QUAD[Index].res
ult,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
 | EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
| EXPR '*' EXPR {AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
 | '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
 Ind=pop();
sprintf(QUAD[Ind].result,"%d",I
```

```
ndex);Ind=pop();
sprintf(QUAD[Ind].result,"%d",I
ndex);
}
 | IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FA
LSE");
strcpy(QUAD[Index].result,"-
1");
push(I
ndex);
Index
++;
}
BLOCK {
 strcpy(QUAD[Index].op,"GOT
O");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result
```

```
,"-1");push(Index);
Index++;
};
ELSEST: ELSE{
 tInd
=pop
();
Ind=
pop()
;
push(
tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BL
O
C
K{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",I
ndex);
};
CONDITION: VAR RELOP VAR {
AddQuadruple($2,$1,$3,$$); StNo=Index-1;
```

```
}
 | VAR
| NUM ;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",
StNo);Ind=pop();
sprintf(QUAD[Ind].result,"%d",I
ndex);
} ;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FA
LSE");
strcpy(QUAD[Index].result,"-
1"); push(Index); Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO
");strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
 strcpy(QUAD[Index].result
,"-1");push(Index);
```

```
Index++;
} ;
%%
extern FILE *yyin;
int main(int argc,char
*argv[]) {FILE *fp;
int i;
if(ar
gc>1
){
fp=fopen(argv[1],
"r");if(!fp) {
printf("\n File not
found");exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t ----------
----
.....");
for(i=0;i<Index;i++) {
printf("\n\t\t %d\t %s\t
%s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
```

```
}
printf("\n\t\t----------");printf("\n\n"); return 0;
}
void push(int
data){
stk.top++;
if(stk.top==1
00) {
printf("\n Stack
overflow\n");exit(0);
}
stk.items[stk.top]=data;
}
int
pop
() {
int
data
;
if(stk.top==-1){
printf("\n Stack
underflow\n");exit(0);
}
data=stk.items[stk.
```

```
top--];return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op); s
trcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex
++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

INPUT

```
$vi test.cmain() {
    int a, b, c;
            if ( a < b)    {
                    a = a + b;
                    }
while(a<b)
{
a=a+b;
```

```
}
if(a<=b)
{
c=a-b;
}
else
 {
c=a+b;
}
}
```

INPUT

```
 $lex int.l
$yacc –d int.y
$gcc lex.yy.c y.tab.c –ll –lm$./a.out test.c
```

OUTPUT

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	t0
1	==	t0	FALSE	5
2	+	a	b	t1
3	==	t1		5
4	GOTO			
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t6
16	=	t6		c

RESULT

Conversion of the BNF rules into YACC form and to generate Abstract Syntax Tree is implemented.

Ex.No: 8 RECOGNITION OF THE GRAMMAR($a^n b$ where n>=10) USING YACC
Date :

AIM

To write a YACC program to recognize the grammar $a^n b$ where n>=10.

ALGORITHM

1. Start the program.
2. Write a program in the vi editor and save it with .l extension.
3. In the lex program, write the translation rules for the variables a and b.
4. Write a program in the vi editor and save it with .y extension.
5. Compile the lex program with lex compiler to produce output file as lex.yy.c.eg $ lex filename.l
6. Compile the yacc program with yacc compiler to produce output file as y.tab.c.eg $ yacc –d arith_id.y
7. Compile these with the C

   compiler asgcc lex.yy.c

   y.tab.c
8. Enter a string as input and it is identified as valid or invalid.

PROGRAM:

Program name: anb.l

```
%{
/*Lex Program for
anb(n>=10)*/#include
"y.tab.h"
%}
%%
a
{retur
n A;}b
{retur
n B;}
. {return yytext[10];}
\n return('\n');
%%
int yywrap()
{
return 1;
}
```

Program name:anb.y

```
%{
/*YACC program for recognising anb(n>=10)*/
```

```
%}
%token A B
%%
stmt: A A A A A A A A A
A anb'\n'{printf("\n Valid
string"); return 0;
}

;
anb:A anb
|A B
;
%%
main()
{
printf("\nEnter some valid
string\n");yyparse();
}
int yyerror(char*s)
{
printf("\nInvalid string\n");
}
```

OUTPUT

```
[root@localhost]# lex anb.1
[root@localhost]# yacc -d
anb.y [root@localhost]# gcc
lex.yy.c y.tab.c
[root@localhost]# ./a.out
Enter some valid
string
aaaaaaaaab
Invalid string
[root@localhost]#
./a.outEnter some
valid string
aaaaaaaaaaab
Valid string
```

RESULT

The YACC program to recognize the grammar a^nb where n>=10 is executed successfully and the output is verified.

Ex.No: 9 IMPLEMENTATION OF THE BACK END OF THE COMPILER
Date :

AIM

To write a program to implement the back end of the compiler.

ALGORITHM

Start the program.

Get the three variables from statements and stored in the text file k.txt.

Compile the program and give the path of the source file.

Execute the program.

Target code for the given statement is produced.

Stop the program.

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<stdlib.h>
void main()
{
int i=2,j=0,k=2,k1=0; char ip[10],kk[10]; FILE *fp;
clrscr();
printf("\nEnter the filename of the intermediate code");
scanf("%s",kk);
fp=fopen(kk,"r"); if(fp==NULL)
{
printf("\nError in opening the file");
getch();
}

clrscr();

while(!feof(fp))
{
fscanf(fp,"%s\n",ip);
```

```
printf("\t\t%s\n",ip);
}

rewind(fp);

printf("\n          \n");
printf("\tStatement\t\tTarget Code\n");
printf("\n          \n");

while(!feof(fp))
{

fscanf(fp,"%s",ip);

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

printf("\t\tMOV %c,R%d\n\t",ip[i+k],j);

if(ip[i+1]=='+')
printf("\t\tADD); else printf("\t\tSUB");

if(islower(ip[i]))
  printf("%c,R%d\n\n",ip[i+k1],j);
else
  printf("%c,%c\n",ip[i],ip[i+2]);
  j++;
k1=2; k=0;
}

printf("\n          \n");
getch();
fclose(fp);

}
```

OUTPUT

Enter the filename of the intermediate code k.txt

X=a-b Y=a-c Z=a+b C=a-b C=a-b

Statement Target Code

X=a-b MOV b,R0 SUB a,R0
Y=a-c MOV a,R1 SUB c,R1
Z=a+b MOV a,R2 ADD b,R2
C=a-b MOV a,R3 SUB b,R3
C=a-b MOV a,R4 SUB b,R4

RESULT

The back end of the compiler is implemented successfully, and the output is verified.

Ex.No: 10 **IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION**
Date : **TECHNIQUES**

AIM

To write a C program to implement simple code optimization techniques such as Common subexpression elimination and Dead Code elimination.

ALGORITHM

Start the program.

The 'L'values and their corresponding 'R' values are given as input.
After common subexpression elimination, the subexpressions that are common are identified and are removed.
After Dead code elimination, the subexpression that is of no use can be identified and is eliminated.
Stop the program.

PROGRAM

```
#include<stdio.h>
#include<conio.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;
clrscr();
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
op[i].l=getche();
printf("\tright: ");
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",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';
strcpy(pr[i].r,'\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);
}
}
getch();
}
```

OUTPUT

Enter the Number of Values:3 left: a right: b+c
left: d right: a+e left: f right: b+c Intermediate Code a=b+c
d=a+e f=b+c
After Dead Code Elimination a =b+c
f =b+c

Eliminate Common Expression a =b+c
a =b+c Optimized Code a=b+c

RESULT

The simple code optimization techniques such as common subexpression elimination and dead code elimination are implemented successfully and the output is verified.

Ex.No: 11 IMPLEMENTATION OF CALCULATOR USING LEX AND YACC
Date :

AIM

To implement a calculator using LEX and YACC.

ALGORITHM

1. Start the program.
2. Write a program in the vi editor and save it with .l extension.
3. In the lex program, write the translation rules for the various mathematical functions.
4. Write a program in the vi editor and save it with .y extension.
5. Compile the lex program with lex compiler to produce output file as lex.yy.c. eg $ lex filename.l
6. Compile the yacc program with yacc compiler to produce output file as y.tab.c. eg $ yacc –d arith_id.y
7. Compile these with the C compiler as gcc lex.yy.c y.tab.c
8. Enter an expression as input and it is evaluated and the answer is displayed as output.

PROGRAM

```
/*calculator.l*/

%{
#include"y.tab.h"
#include<math.h>
#include<stdio.h>
%}

%%

([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext); return NUMBER;}
log |
LOG {return LOG;} ln {return nLOG;} sin |
SIN {return SINE;} cos |
COS {return COS;} tan |
TAN {return TAN;} mem {return MEM;} [\t];
\$ {return 0;}
\n|. return yytext[0];
```

```
%%
/*calculator.y*/
%{
#include<stdio.h> #include<stdlib.h> #include<error.h> double memvar;
%}
%union

{
double dval;
}

%token<dval> NUMBER
%token<dval> MEM
%token LOG SINE nLOG COS TAN
%left '-' '+'
%left '*' '/'
%right '^'
%left LOG SINE nLOG COS TAN
%nonassoc UMINUS
%type <dval> expression

%%
start:statement'\n'
| start statement'\n'
;

statement: MEM '=' expression { memvar=$3;}
| expression {printf( "Answer=%g\n",$1);}
;
expression:expression '+' expression {$$=$1+$3;}
| expression '-' expression {$$=$1-$3;}
| expression '*' expression {$$=$1*$3;}
| expression '/' expression

{
if($3==0)
    yyerror("DIVIDE BY ZERO");
    else
$$=$1/$3;
}

| expression '^' expression {$$=pow($1,$3);}
;
expression: '_' expression %prec UMINUS {$$=-$2;}
| '(' expression ')' {$$=$2;}
```

```
| LOG expression {$$=log($2)/log(10);}
| nLOG expression {$$=log($2);}
| SINE expression {$$=sin($2*3.141592654/180);}
| COS expression {$$=cos($2*3.141592654/180);}
| TAN expression {$$=tan($2*3.141592654/180);}
| NUMBER {$$=$1;}
| MEM {$$=memvar;}
;
%%
main()
{
printf("Enter the expression:");
yyparse();
}

int yyerror(char *error)
{
fprintf(stderr,"%s\n",error);

}
```

OUTPUT

```
[user@localhost cs44]$ lex calculator.l [user@localhost cs44]$ yacc -d calculator.y
[user@localhost cs44]$ cc y.tab.c lex.yy.c -ll -ly -lm [user@localhost cs44]$ ./a.out
Enter the expression:2+2 Answer=4
2+5*42

Answer=212 cos45 Answer=0.707107 sin30
Answer=0.5 tan20
```

RESULT

The calculator is implemented using LEX and YACC and the output is verified.

Ex.No: 12 IMPLEMENTATION OF HEAP STORAGE ALLOCATION STRATEGY
Date :

AIM

To write a program to implement heap storage allocation strategy.

ALGORITHM

Start the program.

Define a function create( ) to create a list of allocated node. This function returns a pointer to head of list.

Define a function display(node) to display the list of allocated nodes.

Define a function search(node,key) to search for the element in list.

Define a function insert(node) to insert element into the list.

Define a function get_prev(node,value) to look for the previous element in the list.

Define a function delete() to remove an element from the list.

Stop the program.

PROGRAM

```
#include"stdio.h"
#include"conio.h"
#include"stdlib.h"
#define TRUE 1
#define FALSE 0

typedef struct Heap
{
int data;
struct Heap *next;
}node;

node *create();

void main()
{
/*local declarations*/ int choice,val;
char ans;
```

```
node *head;
void display(node *); node *search(node *,int); node *insert(node *); void dele(node
**); head=NULL;
do
{
        clrscr();
        printf("\n Program 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 ("\n Enter Your Choice(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:clrscr();
        printf("Invalid Choice,Try again");
        getch();
        }

}while(choice!=5);

}

/*The create function creates a list of allocated node
*Input:None
*Output:Retyurns a pointer to head of list
*Parameter Passing Methopd:Node
**/
node *create()

{
node *temp,*new,* head; int val,flag;
char ans='y';
node *get_node();
temp=NULL;
f lag=TRUE;
/*flag to indicate whether a new node is created for the first time or not*/ do
{
```

```
printf("\n Enter the Element"); scanf("%d",&val);
/*allocate new node*/ new =get_node(); if(new==NULL)
printf("\n Memory is not allocated");
new-> data=val;
if (flag==TRUE)/* Executed only for the first time*/
      {
      head=new;
      temp=head; /*head is the first node in the heap*/ flag=FALSE;
      }
else
      {
      /*temp keeps track of the most recently created node*/ temp->next=new;
      temp=new;
      }

printf(\nDo you want to enter more elements?(y/n)"); ans=getch();
}while(ans= = 'y'); printf("\nThe list is created"); getch();
clrscr(); return head;
}

node *get_node()
{
node *temp; temp=(node*)malloc(sizeof(node));
//using the mem. Allocation function temp->next=NULL;
return temp;
}
/*
*The display function
*Input:Address of the first node of the list
*Output:Displays the list of allocated nodes
*Parameter Passing Method : call by value
*Called by main
**/

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

```
temp=temp->next;
}
print("NULL"); getch();
clrscr();
}

/*
*The search function
*Input: Address of the starting node and the element which is *to be searched
*Output:Searches for the element in list
*If found returns pointer to that node Otherwise NULL
*Parameter passing Method:call by value
*Called by:main
*Calls:None
**/

node *search(node *head,int key)
{
node*temp; int found; temp=head;
if (temp= =Null)
{
printf("The linked list is empty \n"); getch();
clrscr();
return NULL;
}

found=FALSE;
While(temp!= NULL && found= =FALSE)
{
if(temp->data != key) temp = temp->next; else
found = True;
}

if(found == TRUE)
{
printf("\n The Elements is present in the list"\n); getch();
return temp;
}
else
printf("\n The Element is not present in the list \n"); getch();
return NULL;
}
/*

*The insert function
*Input: Address of starting node of the list
```

```
*Output:inserts element into the list
*Parameter Passing Methods: call by value
*Called by : main
*Calls : search()
**/

node *insert(node *head)
{
int choice;
node *insert_head(node*); void insert_after(node*); void insert_last(node*);
printf("\n"1.Insert a node as a head node"); printf("\n"1.Insert a node as a last node");
printf("\n"1.Insert a node as at the intermediate position in the list "); printf("\n"1.Enter
your choice for insertion of node "); scanf("%d",&choice);
switch(choice)
{
case 1:head = insert_head(head); break;
case2:insert_last(head); break; case2:insert_after (head);
break;
}
return head;
}

/*Insertion of node at first position*/ node *insert_head(node*head)
{
node *New,*temp;
New = get_node();
printf ("\n Enter 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;
}

/*Insertion of node at last position*/ void insert_last(node *head)
{
node *New,*temp;
New = get_node();
printf ("\n Enter 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;
}
}

/*Insertion of node at intermediate position*/ void insert_after(node *head)
{
int key;
node *New,*temp;
New = get_node();
printf("Enter the element after which you want to insert "); scanf("%d",&key);
temp=head; do
{

if(temp->data==key)
{
printf ("Enter element which you want to insert "); scanf("%d",&New->data);
New->next=temp->next; temp->next=New; return;
}

else
temp=temp->next;
}while(temp!=NULL);
}

/*
*The get prev function
*Input: Address of starting node and the elemnt to be *searched
*Output:looks for the element in the list
*If found returns pointer to the previous node otherwise NULL
*Parameter Passing Methods: call by value
*Called by : dele()
*Calls : none
**/

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) /*if Flag is true*/ return prev;
else
return NULL;
}

/*
*The get prev function
*Input: Address of starting node and the elemnt to be *searched
*Output:looks for the element in the list
*If found returns pointer to the previous node otherwise NULL
*Parameter Passing Methods: call by value
*Called by : dele()
*Calls : none
**/

void dele(node **head)
{
int key;
node *New,*temp; temp=*head;
if (temp== NULL)
{
printf (“\n The list is empty \n ”); getch();
clrscr(); return;
}

clrscr();

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); // using the mem. Dellocation function
}

printf("\n"The Element is deleted \n"); getch();
clrscr();
}
}
```

OUTPUT

Program to perform various operations on heap using Dynamic memory management.

Create

Display

Insert an element in a list

Delete an element from list

Quit

Enter your choice(1-5) 1 Enter the element: 10
Do you want to enter more elements? (y/n) y Enter the element:20
Do you want to enter more elements?(y/n)y Enter the element:30
Do you want to enter more elements?(y/n)n The List is created

Program to perform various operations on Heap using Dynamic memory management.
Create

Display

Insert an element in a list

Delete an element from list

Quit

Enter your choice(1-5) 4

Enter the element you want to delete: 20 The element is present in the list
The element is deleted

Program to perform various operations on Heap using Dynamic memory management.

Create

Display

Insert an element in a list

Delete an element from list

Quit

Enter your choice(1-5) 2 10-> 30-> NULL

RESULT

The heap storage allocation strategy is implemented successfully, and the output is verified.