Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define MAX_TOKEN_LENGTH 100
#define MAX_INPUT_LENGTH 1000

typedef enum {
    TOKEN_KEYWORD,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_STRING_LITERAL,
    TOKEN_CHAR_LITERAL,
    TOKEN_OPERATOR,
    TOKEN_PUNCTUATION,
    TOKEN_UNKNOWN,
    TOKEN_END
} TokenType;

typedef struct {
    TokenType type;
    char value[MAX_TOKEN_LENGTH];
} Token;

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

Token getNextToken(const char **input);
void printToken(Token token);
int isKeyword(const char *str);
int isOperatorChar(char c);
void skipWhitespaceAndComments(const char **input);

int main() {
    char inputBuffer[MAX_INPUT_LENGTH];
    printf("Enter C code: ");
    if (fgets(inputBuffer, sizeof(inputBuffer), stdin) == NULL) {
        printf("Error reading input.\n");
        return 1;
    }
    size_t len = strlen(inputBuffer);
```

Date: 20.08.25

EXPERIMENT 1.1
LEXICAL ANALYZER USING C

AIM:

Design and implement a lexical analyzer for a given language using C and the lexical analyzer should ignore redundant spaces, tabs and newlines.

ALGORITHM:

Step 0: Start
Step 1: Display a message asking the user to enter C code.

Step 2: Read the entire line of C code from the user into a string variable. If reading fails, display an error message and stop execution.

Step 3: Remove any newline character from the end of the input string if present.

Step 4: Set a pointer or index to the beginning of the input string.

Step 5: Display a heading indicating that tokens will be shown.

Step 6: Repeat until the end of the input string is reached:
6.1 Ignore any whitespace characters by moving the pointer to the next non-space character.
6.2 If the end of the string is reached, stop the loop.
6.3 If the current character begins a string literal (starts with `"`), read characters until the matching `"` is found, and classify this as a string literal token.
6.4 If the current character begins a character literal (starts with `'`), read characters until the matching `'` is found, and classify this as a character literal token.
6.5 If the current character is a letter or underscore, continue reading while the characters are letters, digits, or underscores.
6.5.1 Compare the resulting word to a list of keywords.
6.5.2 If it matches, classify it as a keyword token; otherwise, classify it as an identifier token.
6.6 If the current character is a digit, continue reading while characters are digits, and classify the result as a number token.
6.7 If the current character matches any operator symbols, classify it as an operator token.
6.8 If the current character is punctuation, classify it as a punctuation token.
6.9 If the current character does not fit into any category, classify it as an unknown token.
6.10 Display the token type and the token value.

```
const char *currentPosition = inputBuffer;
```

```
  if (len > 0 && inputBuffer[len - 1] == '\n') {
     inputBuffer[len - 1] = '\0';
   } Token token;
   printf("\nTokens:\n");
   while ((token = getNextToken(&currentPosition)).type != TOKEN_END) {
      printToken(token);
   }
   return 0;
}
void skipWhitespaceAndComments(const char **input) {
   int inComment = 1;
   while (inComment) {
      inComment = 0;
      while (isspace(**input)) {
         (*input)++;
      }
      if (**input == '/' && *(*input + 1) == '/') {
         (*input) += 2;
         while (**input != '\n' && **input != '\0') {
            (*input)++;
         }
         inComment = 1;
         continue;
      }
      if (**input == '/' && *(*input + 1) == '*') {
         (*input) += 2;
         while (!(**input == '*' && *(*input + 1) == '/') && **input != '\0') {
            (*input)++;
         }
         if (**input == '*' && *(*input + 1) == '/') {
            (*input) += 2;
         }
         inComment = 1;
         continue;
      }
   }
}
Token getNextToken(const char **input) {
   Token token;
   token.type = TOKEN_UNKNOWN;
   token.value[0] = '\0';
   skipWhitespaceAndComments(input);
   if (**input == '\0') {
      token.type = TOKEN_END;
      return token;
   }
   if (**input == '"') {

      int length = 0;
```

Step 7: After processing all tokens, stop the program.

Step 8: Stop

```
        (*input)++;
        while (**input != '"' && **input != '\0' && length < MAX_TOKEN_LENGTH - 1) {
            token.value[length++] = **input;
            (*input)++;
        }
        if (**input == '"') {
            (*input)++;
        }
        token.value[length] = '\0';
        token.type = TOKEN_STRING_LITERAL;
        return token;
    }
    if (**input == '\'') {
        int length = 0;
        (*input)++;
        while (**input != '\'' && **input != '\0' && length < MAX_TOKEN_LENGTH - 1) {
            token.value[length++] = **input;
            (*input)++;
        }
        if (**input == '\'') {
            (*input)++;
        }
        token.value[length] = '\0';
        token.type = TOKEN_CHAR_LITERAL;
        return token;
    }
    if (isalpha(**input) || **input == '_') {
        int length = 0;
        while ((isalnum(**input) || **input == '_') && length < MAX_TOKEN_LENGTH -
1) {
            token.value[length++] = **input;
            (*input)++;
        }
        token.value[length] = '\0';
        token.type = isKeyword(token.value) ? TOKEN_KEYWORD :
TOKEN_IDENTIFIER;
        return token;
    }
    if (isdigit(**input)) {
        int length = 0;
        while (isdigit(**input) && length < MAX_TOKEN_LENGTH - 1) {
            token.value[length++] = **input;
            (*input)++;
        }
        token.value[length] = '\0';
        token.type = TOKEN_NUMBER;
        return token;
    }
    if (isOperatorChar(**input) || ispunct(**input)) {
```

```
        token.value[0] = **input;
        token.value[1] = '\0';
        token.type = isOperatorChar(**input) ? TOKEN_OPERATOR :
TOKEN_PUNCTUATION;
        (*input)++;
        return token;
    }
    token.value[0] = **input;
    token.value[1] = '\0';
    token.type = TOKEN_UNKNOWN;
    (*input)++;
    return token;
}

void printToken(Token token) {
    const char *typeStr;
    switch (token.type) {
        case TOKEN_KEYWORD: typeStr = "Keyword"; break;
        case TOKEN_IDENTIFIER: typeStr = "Identifier"; break;
        case TOKEN_NUMBER: typeStr = "Number"; break;
        case TOKEN_STRING_LITERAL: typeStr = "String Literal"; break;
        case TOKEN_CHAR_LITERAL: typeStr = "Char Literal"; break;
        case TOKEN_OPERATOR: typeStr = "Operator"; break;
        case TOKEN_PUNCTUATION: typeStr = "Punctuation"; break;
        case TOKEN_UNKNOWN: typeStr = "Unknown"; break;
        case TOKEN_END: typeStr = "End"; break;
        default: typeStr = "Invalid"; break;
    }
    printf("Token: %-15s Value: %s\n", typeStr, token.value);
}

int isKeyword(const char *str) {
    for (size_t i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isOperatorChar(char c) {
    return strchr("+-*/=<>!&|%^", c) != NULL;
}
```

OUTPUT:

```
Enter C code: int main() { int x = 42; char y = 'a'; if (x > 0) { printf("Hello, World!"); } }

Tokens:
Token: Keyword        Value: int
Token: Identifier     Value: main
Token: Punctuation    Value: (
Token: Punctuation    Value: )
Token: Punctuation    Value: {
Token: Keyword        Value: int
Token: Identifier     Value: x
Token: Operator       Value: =
Token: Number         Value: 42
Token: Punctuation    Value: ;
Token: Keyword        Value: char
Token: Identifier     Value: y
Token: Operator       Value: =
Token: Char Literal   Value: a
Token: Punctuation    Value: ;
Token: Keyword        Value: if
Token: Punctuation    Value: (
Token: Identifier     Value: x
Token: Operator       Value: >
Token: Number         Value: 0
Token: Punctuation    Value: )
Token: Punctuation    Value: {
Token: Identifier     Value: printf
Token: Punctuation    Value: (
Token: String Literal Value: Hello, World!
Token: Punctuation    Value: )
Token: Punctuation    Value: ;
Token: Punctuation    Value: }
Token: Punctuation    Value: }
```

RESULT:

Program to design and implement a lexical analyzer for a given language using C is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
char result[20][20], states[20][20];

void add_state(char a[10], int i) {
   strcpy(result[i], a);
}

void display(int n, char *origState) {
   int k = 0;
   printf("\nEpsilon closure of %s = {", origState);
   while (k < n) {
      printf(" %s", result[k++]);
   }
   printf(" }\n");
}

int is_present(char *state, int count) {
   for (int i = 0; i < count; i++) {
      if (strcmp(result[i], state) == 0) return 1;
   }
   return 0;
}

int main() {
   FILE *INPUT;
   INPUT = fopen("input.txt", "r");
   if (!INPUT) {
      printf("Error opening input.txt\n");
      return 1;
   }

   char currState[10], origState[10];
   int end, i = 0, n, k = 0;
   char state1[10], input[10], state2[10];

   printf("\nEnter the no of states: ");
   scanf("%d", &n);

   printf("\nEnter the states: ");
   for (k = 0; k < n; k++)
      scanf("%s", states[k]);

   for (k = 0; k < n; k++) {
      i = 0;
      strcpy(origState, states[k]);
```

Date: 20.08.25

<u>EXPERIMENT 1.2</u>
<u>E - CLOSURE OF AN NFA</u>

<u>AIM:</u>
Write a program to find ε - closure of all states of any given NFA with ε transition.

<u>ALGORITHM:</u>
Step 0: Start
Step 1: Open the file input.txt in read mode. If the file cannot be opened, display an error message and stop execution.
Step 2: Read the total number of states from the user and store it in n.
Step 3: Read and store all n state names in a list.
Step 4: For each state in the list:
4.1 Set this state as the original state.
4.2 Add the original state to the result list of reachable states.
4.3 Initialize a counter processed to 0.
4.4 Repeat until all states in the result list are processed:
    4.4.1 Take the state at position processed from the result list and call it the current state.
    4.4.2 Move the file pointer to the start of the input file.
    4.4.3 Read each transition from the file as (state1, inputSymbol, state2).
    4.4.4 If state1 matches the current state **and** inputSymbol is epsilon (e):
        - If state2 is not already in the result list, add it to the result list.
    4.4.5 Continue until all transitions in the file are checked.
    4.4.6 Increment processed by 1.
4.5 Display the epsilon closure of the original state by printing all states in the result list.
Step 5: Close the input file.
Step 6: Stop

```
    add_state(origState, i++);

        int processed = 0
while (processed < i) {
            strcpy(currState, result[processed]);
            rewind(INPUT);
            while ((end = fscanf(INPUT, "%s %s %s", state1, input, state2)) != EOF) {
                if (strcmp(currState, state1) == 0 && strcmp(input, "e") == 0) {
                    if (!is_present(state2, i)) {
                        add_state(state2, i++);
                    }
                }
            }
            processed++;
        }

        display(i, origState);
    }

    fclose(INPUT);
    return 0;
}
```

input.txt

```
q0 e q1
q1 e q2
q2 a q0
```

OUTPUT:

```
Enter the no of states: 3
Enter the states: q0 q1 q2
Epsilon closure of q0 = { q0 q1 q2 }
Epsilon closure of q1 = { q1 q2 }
Epsilon closure of q2 = { q2 }
```

RESULT:

Program to find ε - closure of all states of any given NFA with ε transition is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

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

int n, a, f;

struct node {
    int state;
    struct node *next;
};

int printTransition(struct node* transition[n][a]){
    printf("\nTransition table:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<a; j++){
            struct node * head = transition[i][j];
            printf("{");
            while(head != NULL){
                printf("q%d, ", head->state);
                head = head->next;
            }
            printf("}\t");
        }
        printf("\n");
    }
}

int findalpha(char inp, int a, char alphabet[a]){
    for(int i=0; i<a; i++){
        if(alphabet[i] == inp){
            return i;
        }
    }
    return -1;
}

int reset(struct node* transition[n][a]){
    for(int i=0; i<n; i++){
        for(int j=0; j<a; j++){
            transition[i][j] = NULL;
        }
    }
}
```

Date: 20.08.25

EXPERIMENT 1.3
CONVERT ε-NFA TO NFA

AIM:

Write a program to convert NFA with ε transition to NFA without ε transition.

ALGORITHM:

Step 0: Start

Step 1: Input number of states (n)

Step 2: Input number of input symbols (a)

Step 3: Initialize transition matrix as a 2D array of linked lists of size n × a

Step 4: Initialize epsilon closure matrix e_closure[n][n] and set[n]

Step 5: Reset transition matrix

5.1: For each state i = 0 to n-1

5.1.1: For each symbol j = 0 to a-1

5.1.1.1: Set transition[i][j] = NULL

Step 6: Input all input symbols in alphabet[] array

Step 7: Input total number of final states (f)

Step 8: Input final states array final[f]

Step 9: Read transitions from file "NFA.txt"

9.1: While not end of file

9.1.1: Read from state, input symbol, to state

9.1.2: Find index of input symbol in alphabet[] using findalpha()

9.1.3: Create a new node with destination state

9.1.4: Insert new node at head of transition[from][index] linked list

Step 10: Print transition table

```
void findClosure(int start, int state, int set[n], struct node * transition[n][a], int
e_closure[n][n]) {
    if(set[state] == 1)
        return;
    set[state] = 1;
    e_closure[start][state] = 1;
    struct node * temp = transition[state][a-1];
    while(temp != NULL){
        findClosure(start, temp->state, set, transition, e_closure);
        temp = temp->next;
    }
}

int print_e_closure(int e_closure[n][n], int i){
    printf("{");
    for(int j=0; j<n; j++){
        if(e_closure[i][j] == 1){
            printf("q%d, ", j);
        }
    }
    printf("} ");
}

int main(){
    printf("Enter no of states: ");
    scanf("%d", &n);
    printf("Enter no of input symbols: ");
    scanf("%d", &a);

    struct node  *transition[n][a];
    char alphabet[a];
    int e_closure[n][n], set[n];
    reset(transition);

    printf("Enter input symbols without space (if epsilon is present, it should be the last
symbol): ");
    scanf("%s", alphabet);

    printf("Enter total no. of final states: ");
    scanf("%d", &f);
    int final[f];

    printf("Enter final states:\n");
    for(int i=0; i<f; i++){
        scanf("%d", &final[i]);
    }

    char state1[3], state2[3], inp[2];
    int from, to, index, end;
```

Step 10.1: For each state i = 0 to n-1

10.1.1: For each symbol j = 0 to a-1

10.1.1.1: Traverse linked list transition[i][j]

10.1.1.2: Print all destination states

Step 11: Compute epsilon closure

11.1: For each state i = 0 to n-1

11.1.1: Set all set[j] = 0 and e_closure[i][j] = 0

11.1.2: Call findClosure(i, i, set, transition, e_closure)

Step 12: Function findClosure(start, state, set, transition, e_closure)

12.1: If set[state] = 1, return

12.2: Mark set[state] = 1

12.3: Set e_closure[start][state] = 1

12.4: Traverse epsilon transitions transition[state][a-1]

12.4.1: For each destination temp->state

12.4.1.1: Call findClosure(start, temp->state, set, transition, e_closure)

Step 13: Print epsilon closures

13.1: For each state i = 0 to n-1

13.1.1: Print e_closure[i][j] where value = 1

Step 14: Print NFA without epsilon transitions

14.1: For each state i = 0 to n-1

14.1.1: For each symbol j = 0 to a-2 (exclude epsilon)

14.1.1.1: Reset set[x] = 0 for all x

14.1.1.2: For each state k = 0 to n-1

14.1.1.2.1: If e_closure[i][k] = 1

14.1.1.2.1.1: Traverse transition[k][j]

14.1.1.2.1.2: Mark set[temp->state] = 1

14.1.1.3: Print e_closure[i] and corresponding reachable states from set[]

Step 15: Print final states after epsilon closure

15.1: For each final state f[i]

15.1.1: Print all states j where e_closure[f[i]][j] = 1

Step 16: Stop

```
  //Build Transition Matrix
    FILE *INPUT = fopen("NFA.txt", "r");
while((end = fscanf(INPUT, "%s %s %s", state1, inp, state2)) != EOF){
        from = state1[1] - '0';
        to = state2[1] - '0';
        index = findalpha(inp[0], a, alphabet);
        struct node * temp = (struct node *)malloc(sizeof(struct node));
        temp->state = to;
        temp->next = transition[from][index];
        transition[from][index] = temp;
    }

    printTransition(transition);

    //Find Epsilon Closure
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            set[j] = 0;
            e_closure[i][j] = 0;
        }
        findClosure(i, i, set, transition, e_closure);
    }

    // Print Epsilon Closure
    printf("\nStates: ");
    for(int i = 0; i < n; i++)
        print_e_closure(e_closure, i);
    printf("\n");

    // Print Epsilon Closure with input symbols
    printf("\nNFA  without epsilon transitions:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<a-1; j++){
            for(int x = 0; x < n; x++) set[x] = 0;

            for(int k=0; k<n; k++){
                if(e_closure[i][k] == 1){
                    struct node * temp = transition[k][j];
                    while(temp != NULL){
                        set[temp->state] = 1;
                        temp = temp->next;
                    }
                }
            }

            print_e_closure(e_closure, i);
            printf("\t%c\t", alphabet[j]);
            printf("{");
            for(int k=0; k<n; k++)
                if(set[k] == 1)
```

```
     printf("q%d, ", k);
            printf("}\n");
        }
    }


    //Final states
    printf("\nFinal states:\n");
    for(int i=0; i<f; i++){
        printf("{");
        for(int j=0; j<n; j++){
            if(e_closure[final[i]][j] == 1){
                printf("q%d ", j);
            }
        }
        printf("}\n");
    }
}
```

NFA.txt

```
q0 0 q0
q1 1 q1
q2 2 q2
q0 e q1
q1 e q2
```

OUTPUT:

```
Enter no of states: 3
Enter no of input symbols: 4
Enter input symbols without space (if epsilon is present, it should be the last symbol):
012e
Enter total no. of final states: 1
Enter final states: 2

Transition table:
{q0, }  {}      {}      {q1, }
{}      {q1, }  {}      {q2, }
{}      {}      {q2, }  {}

States: {q0, q1, q2, } {q1, q2, } {q2, }

NFA  without epsilon transitions:
{q0, q1, q2, }  0       {q0, }
{q0, q1, q2, }  1       {q1, }
{q0, q1, q2, }  2       {q2, }
{q1, q2, }      0       {}
```

{q1, q2, }	1	{q1, }
{q1, q2, }	2	{q2, }
{q2, }	0	{}
{q2, }	1	{}
{q2, }	2	{q2, }

Final states:

{q2 }

RESULT:

Program to convert NFA with ε transition to NFA without ε transition is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int st;
   struct node *link;
};
struct node1 {
   int nst[20];
};

void insert(int, char, int);
int findalpha(char);
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1, struct node1);
void printnewstate(struct node1);

static int set[20], nostate, noalpha, s, notransition, nofinal, start, finalstate[20], c, r,
buffer[20];
int complete = -1;
char alphabet[20];
static int eclosure[20][20] = {0};
struct node1 hash[20];
struct node *transition[20][20] = {NULL};

void main() {
   int i, j, k, m, t, n, l;
   struct node *temp;
   struct node1 newstate = {0}, tmpstate = {0};
   printf("NOTE: Use letter e as epsilon\n");
   printf("NOTE: e must be last character ,if it is present\n");
   printf("\nEnter the number of alphabets and alphabets: ");
   scanf("%d", &noalpha);
   getchar();
   for (i = 0; i < noalpha; i++) {
      alphabet[i] = getchar();
      getchar();
   }
   printf("Enter the number of states: ");
```

Date: 20.08.25

EXPERIMENT 1.4
CONVERT NFA TO DFA

AIM:

Write a program to convert NFA to DFA.

ALGORITHM:

Step 0: Start

Step 1: Input total number of alphabets (noalpha) and store them in alphabet[]

Step 2: Input number of states (nostate)

Step 3: Input start state (start)

Step 4: Input number of final states (nofinal) and store them in finalstate[]

Step 5: Input number of transitions (notransition)

Step 6: For each transition i = 1 to notransition

6.1: Read (r, input symbol c, s)

6.2: Find index j of c using findalpha()

6.3: Create a node for destination state s

6.4: Insert node into transition[r][j] linked list

Step 7: Initialize DFA state table hash[] to all zeros

Step 8: Set complete = -1

Step 9: Create new DFA state newstate containing start state

Step 10: Insert newstate into DFA state table hash[] using insertdfastate()

Step 11: Repeat until all DFA states are processed

11.1: Take next DFA state newstate = hash[i]

11.2: For each input symbol k = 0 to noalpha-1

11.2.1: Reset set[] = 0

11.2.2: For each NFA state j in newstate

11.2.2.1: Traverse transition[j][k] linked list

11.2.2.2: Add all reachable states into set[]

```
 scanf("%d", &nostate);
printf("Enter the start state: ");
scanf("%d", &start);
printf("Enter the number of final states: ");
scanf("%d", &nofinal);
printf("Enter the final state(s): ");
for (i = 0; i < nofinal; i++)
   scanf("%d", &finalstate[i]);
printf("Enter no of transition: ");
scanf("%d", &notransition);
printf("NOTE: Transition is in the form–> qno alphabet qno\n");
printf("NOTE: States number must be greater than zero\n");
printf("\nEnter the transition: \n");
for (i = 0; i < notransition; i++) {
   scanf("%d %lc%d", &r, &c, &s);
   insert(r, c, s);
}
for (i = 0; i < 20; i++) {
   for (j = 0; j < 20; j++)
      hash[i].nst[j] = 0;
}
complete = -1;
i = -1;
printf("\n....Equivalent DFA....\n");
printf(".....................\n");
printf("Trnsitions of DFA:\n");
newstate.nst[start] = start;
insertdfastate(newstate);
while (i != complete) {
   i++;
   newstate = hash[i];
   for (k = 0; k < noalpha; k++) {
      c = 0;
      for (j = 1; j <= nostate; j++)
         set[j] = 0;
      for (j = 1; j <= nostate; j++) {
         l = newstate.nst[j];
         if (l != 0) {
            temp = transition[l][k];
            while (temp != NULL) {
               if (set[temp->st] == 0) {
                  c++;
                  set[temp->st] = temp->st;
               }
```

11.2.3: If set[] is not empty

11.2.3.1: Create new DFA state tmpstate from set[]

11.2.3.2: Insert tmpstate into DFA state table (if not already present)

11.2.3.3: Print transition: newstate --alphabet[k]--> tmpstate

11.2.4: Else

11.2.4.1: Print transition: newstate --alphabet[k]--> NULL

Step 12: Print all DFA states stored in hash[]

Step 13: Print all alphabets

Step 14: Print DFA start state

Step 15: Find DFA final states

15.1: For each DFA state in hash[]

15.1.1: If it contains any NFA final state

15.1.1.1: Print that DFA state as final

Step 16: Stop

```
            temp = temp->link;
                }
            }
        }printf("\n");
        if (c != 0) {
            for (m = 1; m <= nostate; m++)
                tmpstate.nst[m] = set[m];
            insertdfastate(tmpstate);
            printnewstate(newstate);
            printf("%c\t", alphabet[k]);
            printnewstate(tmpstate);
            printf("\n");
        }
        else {
            printnewstate(newstate);
            printf("%c\t", alphabet[k]);
            printf("NULL\n");
        }
    }
  }
  printf("\nStates of DFA:\n");
  for (i = 0; i <= complete; i++)
      printnewstate(hash[i]);
  printf("\nAlphabets:\n");
  for (i = 0; i < noalpha; i++)
      printf("%c\t", alphabet[i]);
  printf("\nStart State:\n");
  printf("q%d", start);
  printf("\nFinal states:\n");
  findfinalstate();
}

int insertdfastate(struct node1 newstate) {
  int i;
  for (i = 0; i <= complete; i++) {
      if (compare(hash[i], newstate))
          return 0;
  }
  complete++;
  hash[complete] = newstate;
  return 1;
}

int compare(struct node1 a, struct node1 b) {
```

```
int i;
    for (i = 1; i <= nostate; i++) {
        if (a.nst[i] != b.nst[i])
            return 0;
    }   return 1;
}
void insert(int r, char c, int s) {
    int j;
    struct node *temp;
    j = findalpha(c);
    if (j == 999) {
        printf("error\n");
        exit(0);
    }
    temp = (struct node *)malloc(sizeof(struct node));
    temp->st = s;
    temp->link = transition[r][j];
    transition[r][j] = temp;
}
int findalpha(char c) {
    int i;
    for (i = 0; i < noalpha; i++)
        if (alphabet[i] == c)
            return i;
    return (999);
}
void findfinalstate() {
    int i, j, k, t;
    for (i = 0; i <= complete; i++) {
        for (j = 1; j <= nostate; j++) {
            for (k = 0; k < nofinal; k++) {
                if (hash[i].nst[j] == finalstate[k]) {
                    printnewstate(hash[i]);
                    printf("\t");
                    j = nostate;
                    break;
                }
            }
        }
    }
}
void printnewstate(struct node1 state {
    int j;
```

```
 printf("{");
    for (j = 1; j <= nostate; j++) {
        if (state.nst[j] != 0)
            printf("q%d,", state.nst[j]);
    }
    printf("}\t");
}
```

OUTPUT:

```
NOTE: Use letter e as epsilon
NOTE: e must be last character ,if it is present

Enter the number of alphabets and alphabets: 2
a b
Enter the number of states: 4
Enter the start state: 1
Enter the number of final states: 2
Enter the final state(s): 3 4
Enter no of transition: 8
NOTE: Transition is in the form–> qno alphabet qno
NOTE: States number must be greater than zero

Enter the transition:
1 a 1
1 b 1
1 a 2
2 b 2
2 a 3
3 a 4
3 b 4
4 b 3

....Equivalent DFA....
Trnsitions of DFA:

{q1,}   a       {q1,q2,}
{q1,}   b       {q1,}
{q1,q2,}        a       {q1,q2,q3,}
{q1,q2,}        b       {q1,q2,}
{q1,q2,q3,}     a       {q1,q2,q3,q4,}
{q1,q2,q3,}     b       {q1,q2,q4,}
{q1,q2,q3,q4,}  a       {q1,q2,q3,q4,}
```

{q1,q2,q3,q4,} b {q1,q2,q3,q4,}
{q1,q2,q4,} a {q1,q2,q3,}
{q1,q2,q4,} b {q1,q2,q3,}

States of DFA:
{q1,} {q1,q2,} {q1,q2,q3,} {q1,q2,q3,q4,} {q1,q2,q4,}
Alphabets:
a b
Start State:
q1
Final states:
{q1,q2,q3,} {q1,q2,q3,q4,} {q1,q2,q4,}

RESULT:

Program to convert NFA to DFA is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

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

int findalpha(char inp, int a, char alphabet[a]){
    for(int i=0; i<a; i++){
        if(alphabet[i] == inp){
            return i;
        }
    }
    return -1;
}

int reset(int n, int table[n][n]){
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            table[i][j] = 0;
        }
    }
}

int isFinal(int i, int f, int final[f]){
    for(int j=0; j<f; j++){
        if(final[j] == i){
            return 1;
        }
    }
    return 0;
}

int printTable(int n, int table[n][n]){
    printf("\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            printf("%d ", table[i][j]);
        }
        printf("\n");
    }
```

Date: 20.08.25

EXPERIMENT 2.5
MINIMIZATION OF DFA

AIM:

Write a program to minimize any given DFA.

ALGORITHM:

Step 0: Start

Step 1: Input total number of alphabets (noalpha)

1.1: For i = 0 to noalpha-1

1.1.1: Read alphabet[i]

Step 2: Input number of states (nostate)

Step 3: Input start state (start)

Step 4: Input number of final states (nofinal)

4.1: For i = 0 to nofinal-1

4.1.1: Read finalstate[i]

Step 5: Input number of transitions (notransition)

Step 6: For each transition i = 0 to notransition-1

6.1: Read (r, c, s) → from state, input symbol, to state

6.2: Call insert(r, c, s)

6.2.1: Find index j of c using findalpha(c)

6.2.1.1: If alphabet[i] = c → return i

6.2.1.2: If not found → return 999 (error)

6.2.2: If j == 999 → print error and exit

6.2.3: Create new node with st = s

6.2.4: Insert node at head of transition[r][j] linked list

```
int merge(int n, int empty[n][n], int i, int j){
    int flag = 0;
    for(int k=0; k<n; k++){
        if(empty[i][k] == 1 && empty[j][k]==1){
            flag = 1;
            break;
        }
    }

    if(flag){
        for(int k=0; k<n; k++){
            if(empty[j][k] == 1){
                empty[i][k] = 1;
                empty[j][k] = 0;
            }
        }
    }
}

int main(){
    int n, a, from, to, end, index;
    char state1[3], state2[3], inp[2];

    printf("Enter total no of states: ");
    scanf("%d", &n);

    printf("Enter total size of alphabet: ");
    scanf("%d", &a);

    char alphabet[a];
    int transition[n][a];
    int table[n][n];

    reset(n, table);

    printf("Enter alphabet without space: ");
    scanf("%s", alphabet);

    int f;
    printf("Enter no of final states: ");
    scanf("%d", &f);
    int final[f];
```

Step 7: Initialize hash[] DFA state table

7.1: For i = 0 to 19

7.1.1: For j = 0 to 19

7.1.1.1: hash[i].nst[j] = 0

Step 8: Set complete = -1, i = -1

Step 9: Create newstate with start state

9.1: newstate.nst[start] = start

Step 10: Insert newstate into DFA state table using insertdfastate()

10.1: For each existing DFA state in hash[]

10.1.1: Compare with newstate

10.1.1.1: If equal → return 0

10.2: Otherwise increment complete and store newstate in hash[complete]

Step 11: Construct DFA transitions

11.1: While i != complete

11.1.1: Increment i

11.1.2: Set newstate = hash[i]

11.1.3: For each alphabet symbol k = 0 to noalpha-1

11.1.3.1: Initialize set[] = 0 and c = 0

11.1.3.2: For each NFA state j = 1 to nostate

11.1.3.2.1: If newstate.nst[j] != 0

11.1.3.2.1.1: Traverse transition[newstate.nst[j]][k]

11.1.3.2.1.2: For each linked node temp

11.1.3.2.1.2.1: If set[temp->st] == 0

11.1.3.2.1.2.1.1: c++

11.1.3.2.1.2.1.2: set[temp->st] = temp->st

11.1.3.2.1.2.2: Move temp = temp->link

```
        for(int i=0; i<f; i++){  scanf("%d", &final[i]);
    }

    //Build Transition Matrix
    FILE *INPUT = fopen("DFA.txt", "r");
    while((end = fscanf(INPUT, "%s %s %s", state1, inp, state2)) != EOF){
        from = state1[1] - '0';
        to = state2[1] - '0';
        index = findalpha(inp[0], a, alphabet);
        transition[from][index] = to;
    }

    //Mark initial pairs
    for(int i=0; i<n; i++){
        for(int j=0; j<i; j++){
            if( (isFinal(i, f, final) && !isFinal(j, f, final)) || (isFinal(j, f, final) && !isFinal(i, f,
final)) ){
                table[i][j] = 1;
            }
        }
    }

    int out1, out2, flag = 1;

    while(flag){
        flag = 0;

        for(int i=0; i<n; i++){
            for(int j=0; j<i; j++){

                if(table[i][j] == 1)
                    continue;

                for(int k=0; k<a; k++){
                    out1 = transition[i][k];
                    out2 = transition[j][k];

                    if(table[out1][out2] == 1 || table[out2][out1] == 1){
                        table[i][j] = 1;
                        flag = 1;
                    }
```

11.1.3.3: If c != 0 (non-empty transition set)

11.1.3.3.1: Create tmpstate from set[]

11.1.3.3.2: Insert tmpstate into DFA state table

11.1.3.3.3: Print DFA transition: newstate --alphabet[k]--> tmpstate

11.1.3.4: Else

11.1.3.4.1: Print DFA transition: newstate --alphabet[k]--> NULL

Step 12: Print all DFA states

12.1: For i = 0 to complete

12.1.1: Print hash[i] using printnewstate()

Step 13: Print all alphabets

13.1: For i = 0 to noalpha-1

13.1.1: Print alphabet[i]

Step 14: Print DFA start state

14.1: Print q(start)

Step 15: Find DFA final states

15.1: For each DFA state hash[i]

15.1.1: For each NFA state j in hash[i]

15.1.1.1: For each NFA final state in finalstate[]

15.1.1.1.1: If hash[i].nst[j] matches finalstate[k]

15.1.1.1.2: Print DFA state as final

Step 16: Stop

```
            }

        }
    }
}
int empty[n][n], k=0;
reset(n, empty);
int flagarr[n];

for(int i = 0; i<n; i++)
    flagarr[i] = 0;

for(int i=0; i<n; i++){
    for(int j=0; j<i; j++){
        if(table[i][j] == 0){
            empty[k][i] = 1;
            empty[k][j] = 1;
            flagarr[i] = 1;
            flagarr[j] = 1;
            k++;
        }
    }
}

for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        if(i!=j){
            merge(n, empty, i, j);
        }
    }
}

for(int i= 0; i<n; i++){
    if(flagarr[i] == 0){
        empty[k][i] = 1;
        k += 1;
    }
}

printf("Minimized DFA Transition Table:\n");
for(int i=0; i<n; i++){
```

```
        int j;
        for(j=0; j<n && empty[i][j]==0; j++);

        if(j==n) continue;

        for(int b=0; b<a; b++){
            // Printing start set
            printf("{");
            for(int k=0; k<n; k++){
                if(empty[i][k] == 1)
                    printf("q%d, ", k);
            }
            printf("}\t");

            // Printing input symbol
            printf("%c \t", alphabet[b]);

            // Printing to set
            int to_state = transition[j][b];

            for(int y=0; y<n; y++){
                if(empty[y][to_state] == 1){
                    printf("{");
                    for(int k=0; k<n; k++){
                        if(empty[y][k] == 1)
                            printf("q%d, ", k);
                    }
                    printf("}\n");
                }
            }

        }
    }
}
```

DFA.txt

```
q0 0 q1
q0 1 q2
q1 0 q0
q1 1 q3
q2 0 q4
```

q2 1 q5
q3 0 q4
q3 1 q5
q4 0 q4
q4 1 q5
q5 0 q5
q5 1 q5

OUTPUT:

```
Enter total no of states: 6
Enter total size of alphabet: 2
Enter alphabet without space: 01
Enter no of final states: 1
5

Minimized DFA Transition Table:
{q0, q1, }      0       {q0, q1, }
{q0, q1, }      1       {q2, q3, q4, }
{q2, q3, q4, }  0       {q2, q3, q4, }
{q2, q3, q4, }  1       {q5, }
{q5, }  0       {q5, }
{q5, }  1       {q5, }
```

RESULT:

Program to minimize any given DFA is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

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

char forbidden[100];
int rejected = 0;
%}

%%
[^\n]+   {
                if (strstr(yytext, forbidden) != NULL)
                rejected = 1;
        }
\n      { /* Ignore newlines */ }
%%

int main() {
        printf("Enter forbidden word: ");
        fgets(forbidden, sizeof(forbidden), stdin);
        forbidden[strcspn(forbidden, "\n")] = '\0';

        printf("Enter message: ");
        yylex();

        if (rejected)
        printf("Rejected: contains \"%s\"\n", forbidden);
        else
        printf("Accepted: does not contain \"%s\"\n", forbidden);

        return 0;

}

int yywrap() {
        return 1;
}
```

Date: 20.08.25

EXPERIMENT 2.1
LEX PROGRAM FOR STRING ANALYSIS

AIM:
Write a lex program to recognize all strings which does not contain first four characters of your name as a substring

ALGORITHM:
Step 0: Start.

Step 1: Include the header files <stdio.h> for input/output functions and <string.h> for string handling.

Step 2: Declare a global character array forbidden[100] to store the forbidden word, and an integer variable rejected initialized to 0 to track whether the forbidden word is found.

Step 3: In the Lex rules section, create a rule [^\n]+ to match any sequence of characters until a newline.
3.1 Inside this rule, check if the matched text contains the forbidden word using strstr(yytext, forbidden).
3.2 If it does, set rejected = 1.

Step 4: Add a rule \n to match newline characters and ignore them.

Step 5: In the main() function, display a message asking the user to enter the forbidden word.

Step 6: Read the forbidden word using fgets() and remove the trailing newline using strcspn().

Step 7: Display a message asking the user to enter the message to check.

Step 8: Call yylex() to process the message according to the Lex rules.

Step 9: After scanning, check the value of rejected. If it is 1, print "Rejected: contains <forbidden>"; otherwise, print "Accepted: does not contain <forbidden>".

Step 10: Implement the yywrap() function to return 1, indicating the end of input.

Step 11: Stop.

OUTPUT:

Enter forbidden word: Rahu
Enter message: Rani is a girl
Accepted: does not contain "Rahu"

Enter forbidden word: Rahu
Enter message: Rahul is a boy
Rejected: contains "Rahu"

RESULT:

Lex program to recognize all strings which do not contain the first four characters of your name as a substring is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

LEX PROGRAM

```
%{
#include "p2.tab.h"
#include <string.h>
%}

%%
[a-zA-Z][a-zA-Z0-9]*          { yylval.str = strdup(yytext); return IDENTIFIER; }
\n                            { return '\n'; }  // let yacc handle newline
.*                            { yylval.str = strdup(yytext); return INVALID; }  // invalid token
%%

int yywrap() {
        return 1;
}
```

YACC PROGRAM

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

int yylex(void);
int yyerror(const char *s);
%}


%union {
        char* str;
}

%token <str> IDENTIFIER
%token <str> INVALID

%%

input:
 /* empty */
 | input line;
```

Date: 20.08.25

EXPERIMENT 2.2
YACC PROGRAM TO IDENTIFY AN INDETIFIER

AIM:
Write a YACC program to recognize a valid variable which starts with a letter followed by any number of letters or digits

ALGORITHM:
Step 0: Start.

Step 1: Include necessary header files.
1.1 In the Lex file, include <string.h> for string functions and "p2.tab.h" for Yacc token definitions.
1.2 In the Yacc file, include <stdio.h> for input/output and <stdlib.h> for memory allocation functions.

Step 2: Define Lex rules for matching input.
2.1 For [a-zA-Z][a-zA-Z0-9]* ,Store in yylval.str using strdup(yytext) and return IDENTIFIER.
2.2 For \n ,Return newline token '\n'.
2.3 For .* ,Store in yylval.str and return INVALID.

Step 3: Implement yywrap() in Lex to return 1, indicating the end of input.

Step 4: In Yacc, define a %union with a char* str to store matched strings. Declare tokens as %token <str> IDENTIFIER and %token <str> INVALID.

Step 5: Write grammar rules in Yacc.
5.1 IDENTIFIER '\n' → Print "Valid variable name" and free memory.
5.2 INVALID '\n' → Print "Invalid variable name" and free memory.
5.3 '\n' → Do nothing for empty lines.

Step 6: In main() of the Yacc file, print "Enter variable names :" and call yyparse() to start parsing.

Step 7: Implement yyerror(const char *s) to print "Invalid variable name: <error>" when syntax errors occur.

Step 8: Continue until end of input, then terminate.

Step 9: Stop.

```
line:
  IDENTIFIER '\n'     { printf("Valid variable name: %s\n", $1); free($1); }
  | INVALID '\n'      { printf("Invalid variable name: %s\n", $1); free($1); }
  | '\n'              { /* empty line, do nothing */ }
  ;

%%

int main() {
        printf("Enter variable names :\n");
        return yyparse();
}

int yyerror(const char *s) {
        printf("Invalid variable name: %s\n", s);
        return 0;
}
```

OUTPUT:

Enter variable names :
123abc
Invalid variable name: 123abc

abc123
Valid variable name: abc123

RESULT:

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

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

LEX PROGRAM

```
%{
#include "calc.tab.h"
#include <stdlib.h>
%}

%%
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
[\t ]+  ;
\n      { return '\n'; }
.       { return yytext[0]; }
%%

int yywrap() { return 1; }
```

YACC PROGRAM

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

%token NUMBER

%left '+' '-'
%left '*' '/'
%left UMINUS

%%
input:
        /* empty */
  | input line
  ;

line:
        '\n'
  | expr '\n'   { printf("Result = %d\n", $1); }
  ;
```

Date: 20.08.25

EXPERIMENT 2.3
IMPLEMENTATION OF CALCULATOR

AIM:
Write a program to implement a calculator using LEX and YACC.

ALGORITHM:
Step 0: Start.

Step 1: Include necessary header files.
1.1 In the Lex file, include "calc.tab.h" for token definitions and <stdlib.h> for type conversions.
1.2 In the Yacc file, include <stdio.h> for input/output and <stdlib.h> for error handling.

Step 2: Define Lex rules for matching input.
2.1 For [0-9]+, convert the string into an integer using atoi(yytext) and return NUMBER.
2.2 For [\t]+, ignore whitespace.
2.3 For \n, return newline token '\n'.
2.4 For ".", return the operator or parenthesis character directly.

Step 3: Implement yywrap() in Lex to return 1, indicating the end of input.

Step 4: In Yacc, declare tokens and precedence.
4.1 Use %token NUMBER to represent numeric tokens.
4.2 Define operator precedence: * and / have higher precedence than + and -, and %prec UMINUS handles unary minus.

Step 5: Write grammar rules in Yacc.
5.1 "input" allows multiple lines of expressions.
5.2 "line" can be a newline or an expression followed by a newline. If it's an expression, print "Result = <value>".
5.3 "expr" handles arithmetic:
- NUMBER → return its value.
- expr + expr → perform addition.
- expr - expr → perform subtraction.
- expr * expr → perform multiplication.
- expr / expr → check divisor, print error if zero, otherwise divide.
- -expr → apply unary negation.
- (expr) → evaluate expression inside parentheses.

Step 6: In main() of Yacc, print "Enter expressions:" and call yyparse() to start parsing.

```
expr:
  NUMBER              { $$ = $1; }
  | expr '+' expr     { $$ = $1 + $3; }
  | expr '-' expr     { $$ = $1 - $3; }
  | expr '*' expr     { $$ = $1 * $3; }
  | expr '/' expr     {
                      if ($3 == 0) {
                      printf("Error: Division by zero\n");
                      exit(1);
                      }
                      $$ = $1 / $3;
                      }
  | '-' expr %prec UMINUS { $$ = -$2; }
  | '(' expr ')'      { $$ = $2; }
  ;
%%

int main() {
        printf("Enter expressions (Ctrl+D to exit):\n");
        yyparse();
        return 0;
}

int yyerror(const char *s) {
        printf("Syntax Error: %s\n", s);
        return 0;
}
```

Step 7: Implement yyerror(const char *s) to print "Syntax Error: <message>" for invalid expressions.

Step 8: Continue parsing until end of input .

Step 9: Stop.

OUTPUT:

Enter expressions (Ctrl+D to exit):
1+(2-3)*3
Result = -2

1-1-2+11-10
Result = -1

1*2+(3+4)
Result = 9

RESULT:

The program to implement a calculator using LEX and YACC is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

LEX PROGRAM

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

%%
"for"           { return FOR; }
"++"            { return PLUSPLUS; }
"="             { return ASSIGN; }
"=="|"<"|">"|"<="|">=" { return RELOP; }
[0-9]+          { return NUM; }
[a-zA-Z_][a-zA-Z0-9_]*  { return ID; }
[ \t\n]+        { /* ignore whitespace */ }
.               { return yytext[0]; }
%%

int yywrap(void) {
        return 1;
}
```

YACC PROGRAM

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

int yylex(void);
void yyerror(const char *s);
%}

/* Tokens from Lex */
%token FOR ID NUM RELOP ASSIGN PLUSPLUS

%%
stmt:
        FOR '(' assign_stmt ';' cond ';' assign_stmt ')' '{' '}'
        {
        printf("Valid FOR loop syntax\n");
```

Date: 20.08.25

EXPERIMENT 2.5
SYNTAX CHECKER FOR “FOR STATEMENT”

AIM:
Write a YACC program to check the syntax of FOR statement in C.

ALGORITHM:
Step 0: Start

Step 1: Include necessary header files**.**
- **1.1** In the Lex file, include "for.tab.h" for token definitions.
- **1.2** In the Yacc file, include <stdio.h> for input/output and <stdlib.h> for memory functions.

Step 2: Define Lex rules for tokenizing input.
- **2.1** Match the keyword "for" and return token FOR.
- **2.2** Match "++" and return token PLUSPLUS.
- **2.3** Match "=" and return token ASSIGN.
- **2.4** Match relational operators (==, <, >, <=, >=) and return token RELOP.
- **2.5** Match numbers ([0-9]+) and return token NUM.
- **2.6** Match identifiers ([a-zA-Z_][a-zA-Z0-9_]*) and return token ID.
- **2.7** Ignore whitespace ([\t\n]+).
- **2.8** Return any other single character as is.

Step 3: Implement yywrap() in Lex to return 1, signaling the end of input.

Step 4: Define tokens in Yacc.
- **4.1** Declare %token FOR ID NUM RELOP ASSIGN PLUSPLUS.

Step 5: Write grammar rules in Yacc.
- **5.1** stmt → Recognize the structure of a valid for loop:
  FOR '(' assign_stmt ';' cond ';' assign_stmt ')' '{' '}'
  On success, print "Valid FOR loop syntax".
- **5.2** assign_stmt → Handle initialization or increment:
  - ID ASSIGN expr
  - ID PLUSPLUS
- **5.3** expr → Match either an identifier (ID) or a number (NUM).
- **5.4** cond → Recognize conditions in the loop:
  - ID RELOP ID
  - ID RELOP NUM

```
        }
;

assign_stmt:
        ID ASSIGN expr
        | ID PLUSPLUS
;

expr:
        ID
        | NUM
;

cond:
        ID RELOP ID
        | ID RELOP NUM
;
%%

void yyerror(const char *s) {
        printf("Syntax Error: %s\n", s);
}

int main() {
        printf("Enter a FOR loop:\n");
        yyparse();
        return 0;
}



line:
        '\n'
  | expr '\n'   { printf("Result = %d\n", $1); }
  ;
```

Step 6: Implement error handling.

6.1 Define yyerror(const char *s) to print "Syntax Error: <message>" when invalid syntax is detected.

Step 7: In main(), print "Enter a FOR loop:" and call yyparse() to begin parsing.

Step 8: Continue until the end of input.

Step 9: Stop.

OUTPUT:

```
Enter a FOR loop:
for (i=0;i<10;i++)
{
}
Valid FOR loop syntax

for (i=0;i<10;i++
Syntax Error: syntax error
```

RESULT:

The program to implement a for statement checker using YACC is completed.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

LEX PROGRAM

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "st.tab.h"
%}

%option noyywrap

%%
[a-zA-Z_][a-zA-Z_0-9]*          { yylval.str = strdup(yytext); return ID; }
[0-9]+(\.[0-9]+)?               { yylval.str = strdup(yytext); return VAL; }
\"[^\"]*\"                      { yylval.str = strdup(yytext); return VAL; }
\'[^\']*\'                      { yylval.str = strdup(yytext); return VAL; }
";"                             { yylval.str = strdup(yytext); return SC; }
"+"                             { yylval.str = strdup(yytext); return PL; }
"-"                             { yylval.str = strdup(yytext); return MI; }
"*"                             { yylval.str = strdup(yytext); return MUL; }
"/"                             { yylval.str = strdup(yytext); return DIV; }
"="                             { yylval.str = strdup(yytext); return EQ; }
"("                             { yylval.str = strdup(yytext); return OP; }
")"                             { yylval.str = strdup(yytext); return CL; }
"^"                             { yylval.str = strdup(yytext); return POW; }
[ \t]+                          ;
\n                              { return '\n'; }
.                               { yylval.str = strdup(yytext); return UNR; }
%%
```

YACC PROGRAM

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

void yyerror(const char *s);
int yylex(void);

typedef struct node {
   struct node *left, *right;
   char *val;
   int label;
} NODE;
```

Date: 20.08.25

EXPERIMENT 2.4
BNF TO YACC CONVERSION WITH ABSTRACT TREE GENERATION

AIM:
Convert the BNF rules into YACC form and write code to generate abstract syntax tree.

ALGORITHM:
Step 0: Start
Step 1: Initialize the NODE structure with fields for value (val), left child (left), right child (right), and label (label).
Step 2: Define the function makeNode(val, left, right) to:
- **2.1** Allocate memory for a new node.
- **2.2** Store the value in val.
- **2.3** Link left and right children.
- **2.4** Set the label to 0.
- **2.5** Return the created node.

Step 3: In the lexical analyzer (st.l), read the input expression character by character and:
- **3.1** Identify tokens such as identifiers (ID), values (VAL), operators (+, -, *, /, ^, =), parentheses, and semicolons.
- **3.2** Assign the token value to yylval.str.
- **3.3** Ignore whitespace and tabs.
- **3.4** Return the token type to the parser.

Step 4: In the parser (st.y), define grammar rules for expressions:
- **4.1** s → e '\n' | e : set the root of the syntax tree (synTree).
- **4.2** e → e + t | e - t | t : create nodes for addition and subtraction.
- **4.3** t → t * f | t / f | f : create nodes for multiplication and division.
- **4.4** f → g ^ f | g : create nodes for exponentiation.
- **4.5** g → (e) | ID | VAL | - g : handle parentheses, identifiers, values, and unary minus.
- **4.6** Use makeNode() in every production to build the syntax tree.

Step 5: After parsing, store the final syntax tree in synTree.
Step 6: If parsing is successful, traverse and display the tree:
- **6.1** In-order traversal (inOrder): Visit left subtree → root → right subtree.
- **6.2** Pre-order traversal (preOrder): Visit root → left subtree → right subtree.
- **6.3** Post-order traversal (postOrder): Visit left subtree → right subtree → root.

Step 7: Free allocated memory for the syntax tree using freeTree() by recursively freeing nodes and their values.
Step 8: Stop.

```
NODE* makeNode(char *val, NODE* left, NODE* right) {
   NODE *temp = (NODE*)malloc(sizeof(NODE));
   if (!temp) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
   }
   temp->val = strdup(val);
   temp->left = left;
   temp->right = right;
   temp->label = 0;
   return temp;
}

NODE* synTree = NULL;
%}

%union {
   char *str;
   struct node *node;
}

%token <str> PL MI MUL DIV OP CL EQ ID VAL SC UNR POW
%type <node> s e t f g

%left PL MI
%left MUL DIV
%right POW
%nonassoc UMINUS

%%

s   : e '\n'       { $$ = $1; synTree = $$; return 0; }
   | e            { $$ = $1; synTree = $$; return 0; }
   ;

e   : e PL t       { $$ = makeNode($2, $1, $3); }
   | e MI t       { $$ = makeNode($2, $1, $3); }
   | t            { $$ = $1; }
   ;

t   : t MUL f      { $$ = makeNode($2, $1, $3); }
   | t DIV f      { $$ = makeNode($2, $1, $3); }
   | f            { $$ = $1; }
   ;

f   : g POW f      { $$ = makeNode($2, $1, $3); }
   | g            { $$ = $1; }
   ;

g   : OP e CL      { $$ = $2; }
```

```
  | ID           { $$ = makeNode($1, NULL, NULL); }
    | VAL          { $$ = makeNode($1, NULL, NULL); }
    | MI g %prec UMINUS { $$ = makeNode($1, NULL, $2); };
%%

void inOrder(NODE* root) {
    if (root) {
        inOrder(root->left);
        printf("%s ", root->val);
        inOrder(root->right);
    }
}

void postOrder(NODE* root) {
    if (root) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%s ", root->val);
    }
}

void preOrder(NODE* root) {
    if (root) {
        printf("%s ", root->val);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void freeTree(NODE* root) {
    if (root) {
        freeTree(root->left);
        freeTree(root->right);
        free(root->val);
        free(root);
    }
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
    printf("Enter expression: ");
    if (yyparse() == 0 && synTree) {
        printf("In Order: ");
        inOrder(synTree);
        printf("\nPre Order: ");
        preOrder(synTree);
        printf("\nPost Order: ");
```

```
 postOrder(synTree);
        printf("\n");
        freeTree(synTree);
} else {
        printf("Parse failed\n");
    }
    return 0;
}
```

OUTPUT:

```
Enter expression: a+b*c -d
In Order: a + b * c - d
Pre Order: - + a * b c d
Post Order: a b c * + d -
```

RESULT:

Program to convert the BNF rules into YACC form and generate abstract syntax tree is completed.