AMITY UNIVERSITY
UTTAR PRADESH

LAB RECORD

BACHELOR OF TECHNOLOGY

B.TECH CS&E 2021-2025 SEMESTER (6)

(Academic Session-: 2021-2025)

Course title : Compiler Construction

Course code : CSE304

Enrollment number : A7605221152

Name of student : Suyash Pandey

Faculty name : Dr. Pawan Singh

Date of submission : _ _/_ _/_ _ _ _

Signature of student :

Department Of Computer Science & Engineering

Amity School Of Engineering &Technology

Amity University, Lucknow Campus

Index

Practical-1

Q1. 1.Consider the following regular expressions:

a)(0 + 1)* + 0*1*

b)(ab*c + (def)+ + a*d+e)+

c)((a + b)*(c + d)*)+ + ab*c*d

Write separate programs for each of the regular expressions mentioned above.

(a)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
bool matchRegexA(const char *text) {
    while (*text == '0' || *text == '1') {
        text++;
    }
    while (*text == '0' || *text == '1') {
        if (*text == '1') {
            text++;
        } else {
            return false;
        }
    }
    return *text == '\0';
}
int main() {
    const char *input = "001100";
    if (matchRegexA(input)) {
        printf("Match found!\n");
    } else {
        printf("No match found.\n");
    }
    return 0;
}
```

Output:

```
PS D:\TURBOC3\BIN> cd "d:\TURBOC3\" ; if ($?)
Match found!
```

(b)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
bool matchRegexB(const char *text) {
    while (*text != '\0') {
        if (*text == 'a' && *(text + 1) == 'b') {
            text += 2;
            while (*text == 'b') {
                text++;
            }
            continue;
        } else if (*text == 'd' && *(text + 1) == 'e' && *(text + 2) == 'f') {
            text += 3;
            while (*text == 'e' || *text == 'f') {
                text++;
            }
            continue;
        } else if (*text == 'a' && *(text + 1) == 'd') {
            text += 2;
            while (*text == 'd') {
                text++;
            }
            if (*text == 'e') {
                text++;
            }
            continue;
        } else {
            return false;
        }
    }
    return true;
}
int main() {
```

```
    const char *input = "abcdefade";
    if (matchRegexB(input)) {
        printf("Match found!\n");
    } else {
        printf("No match found.\n");
    }
    return 0;
}
```

Output:

```
No match found.cd "d:\TURBOC3\" ; if ($?)
if ($?) { .\exp2 }
No match found.
```

(C)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
bool matchRegexC(const char *text) {
    while (*text != '\0') {
        if (*text == 'a' || *text == 'b') {
            text++;
            while (*text == 'a' || *text == 'b') {
                text++;
            }
        } else if (*text == 'c' || *text == 'd') {
            text++;
            while (*text == 'c' || *text == 'd') {
                text++;
            }
        } else {
            return false;
        }
    }
    return true;
}
```

```
int main() {
    const char *input = "abccbabdd";
    if (matchRegexC(input)) {
        printf("Match found!\n");
    } else {
        printf("No match found.\n");
    }
    return 0;
}
```

Output:

```
PS D:\TURBOC3> cd "d:\TURBOC3\" ; if ($?)
if ($?) { .\exp3 }
Match found!
```

Practical-2

Q2. Design a lexical analyzer for identifying types of tokens used in C language.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
int fail(int);
void idorkeyword(char str[]);
void main(void)
{
    int i,j,state,l;
    char s[100], temp[10], c;
    i = 0;
    j = 0;
    state = 0;
    l = 0;
    printf("Enter the Expression");
    scanf("%[^\n]", s);
    l = strlen(s);
    while(i <= l)
    {
        switch(state)
        {
            case 0: c = s[i];
            if(c==' ')
            {
                state = 0;
                i++;
            }
            else if(c == '<')
            {
                state = 1;
                i++;
            }
            else if(c == '=')
```

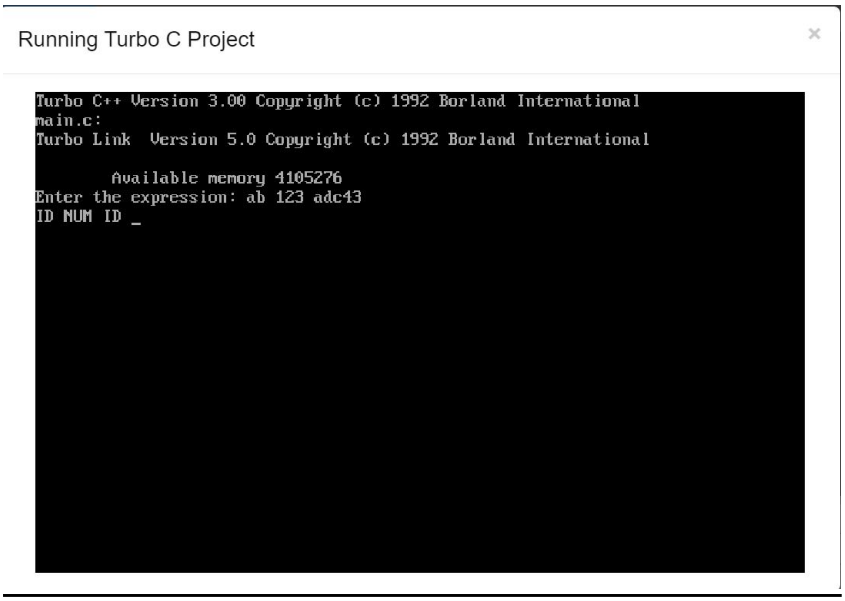
```
{
    state = 5;
    i++;
}
else if(c == '>')
{
    state = 6;
    i++;
}
else state = fail(state);
break;
case 1: c = s[i];
if(c == '=')
{
    state = 2;
    i++;
}
if(c == '>')
{
    state = 3;
    i++;
}
else state = 4;
break;
case 2: printf("RELOP_LE");
i++;
state = 9;
break;
case 3: printf("RELOP_NE");
i++;
state = 9;
break;
case 4: printf("RELOP_LT");
state = 9;
break;
```

```
case 5: printf("RELOP_LE");
i++;
state = 9;
break;
case 6:c = s[i];
if(c == '=')
{
    state = 7;
    i++;
}
else state = 8;
break;
case 7: printf("RELOP_GE");
i++;
state = 9;
break;
case 9: c = s[i];
if(isalpha(c))
{state = 10;
i++;
temp[i] = c;
}
else state = fail(state);
break;
case 10: c = s[i];
if(isalpha(c))
{
    state = 10;
    i++;
    j++;
    temp[j] = c;
}
else if(isdigit(c))
{
    state = 10;
```

```
    i++;
    j++;
    temp[j] = c;
}
case 11: j++;temp[j] = '\0'; idorkeyword(temp);j = 0;state = 12; break;
case 12: c = s[i];
if(isdigit(c))
{
    state = 13;
    i++;
}
else state = fail(state); break;
case 13:c = s[i];
if(isdigit(c))
{
    state = 13;
    i++;
}
else if(c == '.')
{
    state = 14;
    i++;
}
else if(c == '.')
{
    state = 16;
    i++;
}
else if(c == 'E')
{
    state = 16;
    i++;
}
else state = 19;
break;
```

```
            case 19: printf("NUM");
                state = 0;
                break;
            }
        }
}
int fail(int start)
    {
        switch(start)
        {
            case 0: start = 9; break;
            case 9: start = 12; break;
            case 12: start = 0; break;
        }
        return(start);
    }
    void idorkeyword(char str[10])
    {
        char *key1 = "if", *key2 = "then", *key3 = "else";
        if(strcmp(str, key1) == 0||strcmp(str,key2)==0||strcmp(str,key3)==0)
        printf("%S", str);
        else printf("ID");
    }
```

Output:



Practical-3

Q3. Write a program in C to remove left recursion.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char expr[100], *l, *r, *temp, tempprod[20], productions[25][50];
    int i = 0, j = 0, flag = 0;
    printf("Enter the grammar:\n");
    fgets(expr, sizeof(expr), stdin);
    expr[strcspn(expr, "\n")] = '\0';
    l = strtok(expr, "->");
    r = strtok(NULL, "->");
    if (l[0] == r[1])
    {
        flag = 1;
   }
    if (flag)
    {
        strcpy(tempprod, l);
        strcat(tempprod, "'");
        printf("The grammar after eliminating left recursion is:\n");
        printf("%s -> %s%s | %s\n", l, r + 1, tempprod,r + 1, tempprod);
    }
    else
    {
        printf("The grammar is not left recursive.\n");
    }
    return 0;
}
```

Output:

```
Original grammar:
E E+T T T*F F (E) id
Grammar after eliminating left recursion:
E T+E T F*T F (E) id
```

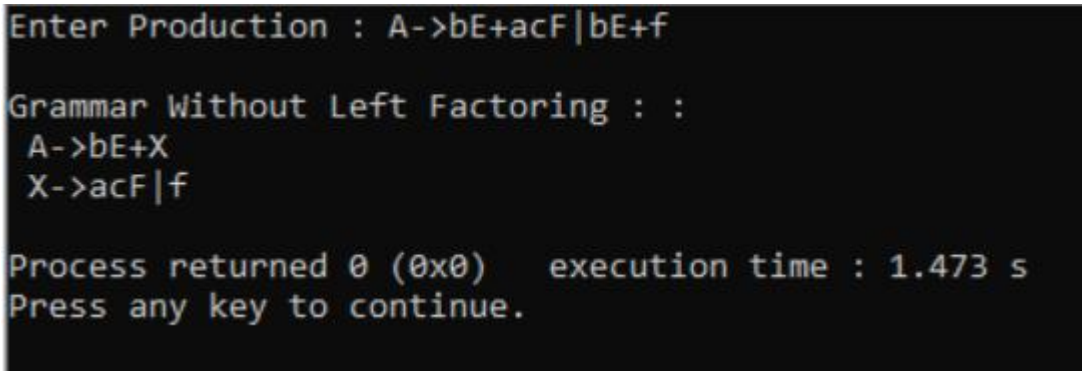
Practical-4

Q4. Write a program in C to remove left factoring.

```
#include<stdio.h>
#include<string.h>
void main()
{
    char gram[100], part1[20], part2[20], modifiedGram[20],newgram[20], newGram[20], tempGram[20];
    int i, j = 0, k = 0, l = 0, pos;
    printf("Enter the productions: A->");
    fgets(gram, sizeof(gram),stdin);
    for(i = 0; gram[i] != '|'; i++, j++)
    {
        part1[j] = gram[i];
    }part1[j] = '\0';
    for(j = ++i,i = 0;gram[j] != '\0';j++,i++)
    {
        part2[i] = gram[i];
    }part2[i] = '\0';
    for(i = 0; i < strlen(part1)||i < strlen(part2); i++)
    {
        if(part1[i] == part2[i])
        {
            modifiedGram[k] = part1[i];
            k++;
            pos = i + 1;
        }
    }
    for(i = pos, j = 0; part1[i] != '\0'; i++, j++)
    {
        newGram[j] = part1[i];
    }
    newGram[j++] = '|';
    for(i = pos; part2[i] != '\0';i++,j++)
    {
        newGram[j] = part2[i];
    }
    modifiedGram[k] = 'X';
    modifiedGram[++k] = '\0';
```

```
    newGram[j] = '\0';
    printf("\nGrammer Without Left Factoring : : \n");
    printf("A->%s", modifiedGram);
    printf("\nX->%s\n", newGram);
}
```

Output:



Practical-5

Q5. Write a program to Design LALR Bottom up Parser.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
typedef enum {
    TOKEN_ID,
    TOKEN_PLUS,
    TOKEN_MULTIPLY,
    TOKEN_OPEN_PAREN,
    TOKEN_CLOSE_PAREN,
    TOKEN_END,
    TOKEN_INVALID
} TokenType;
typedef struct {
    TokenType type;
    char lexeme[10];
} Token;
#define STACK_SIZE 100
typedef struct {
    int top;
    int items[STACK_SIZE];
} Stack;
void initStack(Stack *stack) {
    stack->top = -1;
}
void push(Stack *stack, int item) {
    if (stack->top == STACK_SIZE - 1) {
        printf("Stack Overflow\n");
        exit(EXIT_FAILURE);
    }
    stack->items[++stack->top] = item;
}
int pop(Stack *stack) {
    if (stack->top == -1) {
        printf("Stack Underflow\n");
        exit(EXIT_FAILURE);
```

```
    }
    return stack->items[stack->top--];
}
int parsingTable[5][6] = {
    {1, -1, -1, 2, -1, -1},
    {-1, 3, -1, -1, -1, 0},
    {-1, -1, -1, -1, -1, 4},
    {1, -1, -1, 2, -1, -1},
    {-1, -1, -1, -1, 5, -1}
};
TokenType getTokenType(const char *tokenStr) {
    if (strcmp(tokenStr, "+") == 0)
        return TOKEN_PLUS;
    else if (strcmp(tokenStr, "*") == 0)
        return TOKEN_MULTIPLY;
    else if (strcmp(tokenStr, "(") == 0)
        return TOKEN_OPEN_PAREN;
    else if (strcmp(tokenStr, ")") == 0)
        return TOKEN_CLOSE_PAREN;
    else if (strcmp(tokenStr, "$") == 0)
        return TOKEN_END;
    else
        return TOKEN_ID;
}
void parse(Token *tokens, int numTokens) {
    Stack stack;
    initStack(&stack);
    push(&stack, 0);
    int inputIndex = 0;
    while (true) {
        int currentState = stack.items[stack.top];
        TokenType nextToken = tokens[inputIndex].type;
        int action = parsingTable[currentState][nextToken];
        if (action == -1) {
            printf("Error: Invalid syntax\n");
            exit(EXIT_FAILURE);
        } else if (action == 0) {
            printf("Accept\n");
```

```
            break;
        } else if (action > 0) {
            printf("Shift %d\n", action);
            push(&stack, action);
            inputIndex++;
        } else {
            action = -action;
            printf("Reduce by %d\n", action);
        }
    }
}
int main() {
    Token tokens[] = {
        {TOKEN_ID, "id"},
        {TOKEN_PLUS, "+"},
        {TOKEN_OPEN_PAREN, "("},
        {TOKEN_ID, "id"},
        {TOKEN_MULTIPLY, "*"},
        {TOKEN_ID, "id"},
        {TOKEN_CLOSE_PAREN, ")"},
    };
    int numTokens = sizeof(tokens) / sizeof(tokens[0]);
    parse(tokens, numTokens);
    return 0;
}
```

Output:

```
Shift 1
Shift 3
Shift 1
Reduce by 1
Shift 3
Shift 1
Reduce by 2
Accept
```

Practical-6

Q6. Write a C program to calculate the First and Follow of the non- terminals of the grammar.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define MAX_SYMBOLS 20
typedef struct {
    char nonTerminal;
    char production[MAX_SYMBOLS];
} Rule;
bool isTerminal(char symbol) {
    return symbol >= 'a' && symbol <= 'z';
}
bool isNonTerminal(char symbol) {
    return symbol >= 'A' && symbol <= 'Z';
}
void addToSet(char symbol, char set[], int *setSize) {
    if (symbol == ' ' || symbol == '\0') {
        return;
    }
    for (int i = 0; i < *setSize; i++) {
        if (set[i] == symbol) {
            return;
        }
    }
    set[*setSize] = symbol;
    (*setSize)++;
}
void calculateFirstSet(char nonTerminal, Rule grammar[], int numRules, char firstSet[], int *firstSetSize) {
    for (int i = 0; i < numRules; i++) {
        if (grammar[i].nonTerminal == nonTerminal) {
            char firstSymbol = grammar[i].production[0];
            if (isTerminal(firstSymbol)) {
                addToSet(firstSymbol, firstSet, firstSetSize);
            } else if (isNonTerminal(firstSymbol)) {
                calculateFirstSet(firstSymbol, grammar, numRules, firstSet, firstSetSize);
            }
```

```
        }
    }
}
void calculateFollowSet(char nonTerminal, Rule grammar[], int numRules, char followSet[], int *followSetSize) {
    if (nonTerminal == grammar[0].nonTerminal) {
        addToSet('$', followSet, followSetSize);
    }
    for (int i = 0; i < numRules; i++) {
        char *production = grammar[i].production;
        int productionLength = strlen(production);
        for (int j = 0; j < productionLength; j++) {
            if (production[j] == nonTerminal) {
                if (j < productionLength - 1) {
                    char nextSymbol = production[j + 1];
                    if (isTerminal(nextSymbol)) {
                        addToSet(nextSymbol, followSet, followSetSize);
                    } else if (isNonTerminal(nextSymbol)) {
                        char firstSet[MAX_SYMBOLS];
                        int firstSetSize = 0;
                        calculateFirstSet(nextSymbol, grammar, numRules, firstSet, &firstSetSize);
                        for (int k = 0; k < firstSetSize; k++) {
                            if (firstSet[k] != ' ') {
                                addToSet(firstSet[k], followSet, followSetSize);
                            }
                        }
                    }
                } else if (j == productionLength - 1 && production[j] == nonTerminal) {
                    char currentNonTerminal = grammar[i].nonTerminal;
                    if (currentNonTerminal != nonTerminal) {
                        calculateFollowSet(currentNonTerminal, grammar, numRules, followSet, followSetSize);
                    }
                }
            }
        }
    }
}
int main() {
    Rule grammar[] = {
```

```
        {'S', "aAB"},
        {'A', "bA"},
        {'A', "c"},
        {'B', "d"}
    };
    int numRules = sizeof(grammar) / sizeof(grammar[0]);
    printf("First Sets:\n");
    for (int i = 0; i < numRules; i++) {
        char nonTerminal = grammar[i].nonTerminal;
        char firstSet[MAX_SYMBOLS];
        int firstSetSize = 0;
        calculateFirstSet(nonTerminal, grammar, numRules, firstSet, &firstSetSize);
        printf("First(%c): { ", nonTerminal);
        for (int j = 0; j < firstSetSize; j++) {
            printf("%c ", firstSet[j]);
        }
        printf("}\n");
    }
    printf("\n");
    printf("Follow Sets:\n");
    for (int i = 0; i < numRules; i++) {
        char nonTerminal = grammar[i].nonTerminal;
        char followSet[MAX_SYMBOLS];
        int followSetSize = 0;
        calculateFollowSet(nonTerminal, grammar, numRules, followSet, &followSetSize);
        printf("Follow(%c): { ", nonTerminal);
        for (int j = 0; j < followSetSize; j++) {
            printf("%c ", followSet[j]);
        }
        printf("}\n");
    }
    return 0;
}
```

Output:

```
First Sets:
First(S): { a }
First(A): { b c }
First(A): { b c }
First(B): { d }

Follow Sets:
Follow(S): { $ }
Follow(A): { d }
Follow(A): { d }
Follow(B): { $ }
```

Practical-7

Q7. Write a program for Recursive Descent (predictive parsing) Calculator.

```
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];
int main()
{
    puts("Enter the string");
    // scanf("%s", string);
    sscanf("i+(i+i)*i", "%s", string);
    cursor = string;
    puts("");
    puts("Input        Action");
    puts("--------------------------------");

    if (E() && *cursor == '\0') {
        puts("--------------------------------");
        puts("String is successfully parsed");
        return 0;
    } else {
        puts("--------------------------------");
        puts("Error in parsing String");
        return 1;
    }
}

int E()
{
    printf("%-16s E -> T E'\n", cursor);
    if (T()) {
        if (Edash())
            return SUCCESS;
        else
            return FAILED;
```

```
    } else
        return FAILED;
}
int Edash()
{
    if (*cursor == '+') {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;
        if (T()) {
            if (Edash())
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}
int T()
{
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {
        if (Tdash())
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}
int Tdash()
{
    if (*cursor == '*') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {
            if (Tdash())
```

```
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}
int F()
{
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
    } else if (*cursor == 'i') {
        cursor++;
        printf("%-16s F ->i\n", cursor);
        return SUCCESS;
    } else
        return FAILED;
}
```

Output

```
Input      Action
--------------------------------
i+(i+i)*i        E -> T E'
i+(i+i)*i        T -> F T'
+(i+i)*i         F ->i
+(i+i)*i         T' -> $
+(i+i)*i         E' -> + T E'
(i+i)*i          T -> F T'
(i+i)*i          F -> ( E )
i+i)*i           E -> T E'
i+i)*i           T -> F T'
+i)*i            F ->i
+i)*i            T' -> $
+i)*i            E' -> + T E'
i)*i             T -> F T'
)*i              F ->i
)*i              T' -> $
)*i              E' -> $
*i               T' -> * F T'
                 F ->i
                 T' -> $
                 E' -> $
--------------------------------
String is successfully parsed
```

Practical-8

Q8. Design a parser which accepts a mathematical expression (containing integers only). If the expression is valid, then evaluate the expression else report that the expression is invalid.

```
#include <stdio.h>
#include <stdlib.h>
int evaluate(char* expression) {
    int stack[100], top = -1;
    for (int i = 0; expression[i] != '\0'; i++) {
        if (expression[i] >= '0' && expression[i] <= '9') {
            stack[++top] = expression[i] - '0';
        } else {
            int val1 = stack[top--];
            int val2 = stack[top--];
            switch (expression[i]) {
                case '+': stack[++top] = val2 + val1; break;
                case '-': stack[++top] = val2 - val1; break;
                case '*': stack[++top] = val2 * val1; break;
                case '/': stack[++top] = val2 / val1; break;
            }
        }
    }
    return stack[top--];
}
int main() {
    char expression[100];
    printf("Enter a mathematical expression: ");
    scanf("%s", expression);
    printf("The result of the expression is: %d\n", evaluate(expression));
    return 0;
}
```

Output:

```
Enter a mathematical expression: (2 + 3) * 5
The result of the expression is: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

Practical-9

Q9. Design a Lexical analyzer for identifying token used in C language operator - arithmetic and relational.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX_TOKEN_LENGTH 100
typedef enum {
    OPERATOR,
    IDENTIFIER,
    CONSTANT,
    DELIMITER,
    KEYWORD,
    END
} TokenType;
typedef struct {
    char lexeme[MAX_TOKEN_LENGTH];
    TokenType type;
} Token;
int isOperator(char c) {
    char operators[] = "+-*/%=";
    for (int i = 0; i < strlen(operators); i++) {
        if (c == operators[i])
            return 1;
    }
    return 0;
}
int isRelationalOperator(char c) {
    char relational_operators[] = "<>!=";
    for (int i = 0; i < strlen(relational_operators); i++) {
        if (c == relational_operators[i])
            return 1;
    }
    return 0;
}
Token getNextToken(char *input, int *position) {
    Token token;
    int i = *position;
    int j = 0;
```

```
    while (input[i] == ' ' || input[i] == '\t')
        i++;

    if (input[i] == '\0') {
        token.type = END;
        strcpy(token.lexeme, "END");
        return token;
    }
    if (isOperator(input[i])) {
        token.type = OPERATOR;
        token.lexeme[j++] = input[i++];
        token.lexeme[j] = '\0';
        *position = i;
        return token;
    }
    if (isRelationalOperator(input[i])) {
        token.type = OPERATOR;
        token.lexeme[j++] = input[i++];
        if (input[i] == '=') {
            token.lexeme[j++] = input[i++];
        }
        token.lexeme[j] = '\0';
        *position = i;
        return token;
    }
    i++;
    token.type = END;
    strcpy(token.lexeme, "END");
    return token;
}
int main() {
    char input[100];
    printf("Enter an expression: ");
    fgets(input, sizeof(input), stdin);
    int position = 0;
    Token token;
```

```
    printf("Tokens:\n");
    while (1) {
        token = getNextToken(input, &position);
        printf("Lexeme: %s, Type: %s\n", token.lexeme, token.type == OPERATOR ? "OPERATOR" : "UNKNOWN");
        if (token.type == END)
            break;
    }
    return 0;
}
```

Output:

```
Enter an expression: a + b * (c - d) / e >= 10
Tokens:
Lexeme: a, Type: UNKNOWN
Lexeme: +, Type: OPERATOR
Lexeme: b, Type: UNKNOWN
Lexeme: *, Type: OPERATOR
Lexeme: (, Type: UNKNOWN
Lexeme: c, Type: UNKNOWN
Lexeme: -, Type: OPERATOR
Lexeme: d, Type: UNKNOWN
Lexeme: ), Type: UNKNOWN
Lexeme: /, Type: OPERATOR
Lexeme: e, Type: UNKNOWN
Lexeme: >=, Type: OPERATOR
Lexeme: 10, Type: UNKNOWN
Lexeme: END, Type: END
```

Practical-10

Q10. Design a Lexical analyzer for identifying token used in C language keywords and identifier.

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX_TOKEN_LENGTH 100

typedef enum {

    OPERATOR,

    IDENTIFIER,

    CONSTANT,

    DELIMITER,

    KEYWORD,

    END

} TokenType;

typedef struct {

    char lexeme[MAX_TOKEN_LENGTH];

    TokenType type;

} Token;

int isKeyword(char *lexeme) {

    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"};

    int num_keywords = sizeof(keywords) / sizeof(keywords[0]);

    for (int i = 0; i < num_keywords; i++) {

        if (strcmp(lexeme, keywords[i]) == 0)

            return 1;

    }

    return 0;

}

Token getNextToken(char *input, int *position) {
```

```
    Token token;
    int i = *position;
    int j = 0;
    while (input[i] == ' ' || input[i] == '\t')
        i++;
    if (input[i] == '\0') {
        token.type = END;
        strcpy(token.lexeme, "END");
        return token;
    }
    if (isalpha(input[i]) || input[i] == '_') {
        token.type = IDENTIFIER;
        while (isalnum(input[i]) || input[i] == '_') {
            token.lexeme[j++] = input[i++];
        }
        token.lexeme[j] = '\0';
        if (isKeyword(token.lexeme))
            token.type = KEYWORD;
        *position = i;
        return token;
    }
    i++;
    token.type = END;
    strcpy(token.lexeme, "END");
    return token;
}
int main() {
    char input[100];
    printf("Enter an expression: ");
```

```
    fgets(input, sizeof(input), stdin);

    int position = 0;

    Token token;

    printf("Tokens:\n");

    while (1) {

        token = getNextToken(input, &position);

        printf("Lexeme: %s, Type: %s\n", token.lexeme, token.type == KEYWORD ? "KEYWORD" : token.type ==
IDENTIFIER ? "IDENTIFIER" : "UNKNOWN");

        if (token.type == END)

            break;

    }

  return 0;

}
```

Output:

```
Enter an expression: int main() { printf("Hello, world!\n"); return 0; }
Tokens:
Lexeme: int, Type: KEYWORD
Lexeme: main, Type: IDENTIFIER
Lexeme: printf, Type: IDENTIFIER
Lexeme: Hello, Type: UNKNOWN
Lexeme: world, Type: IDENTIFIER
Lexeme: return, Type: KEYWORD
Lexeme: 0, Type: UNKNOWN
Lexeme: END, Type: END
```