VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE307P

Course Title: Compiler Design Lab

Lab Slot: L49 + L50

Guided By: Dr. Kannadasan R

Lab Assessment 2.

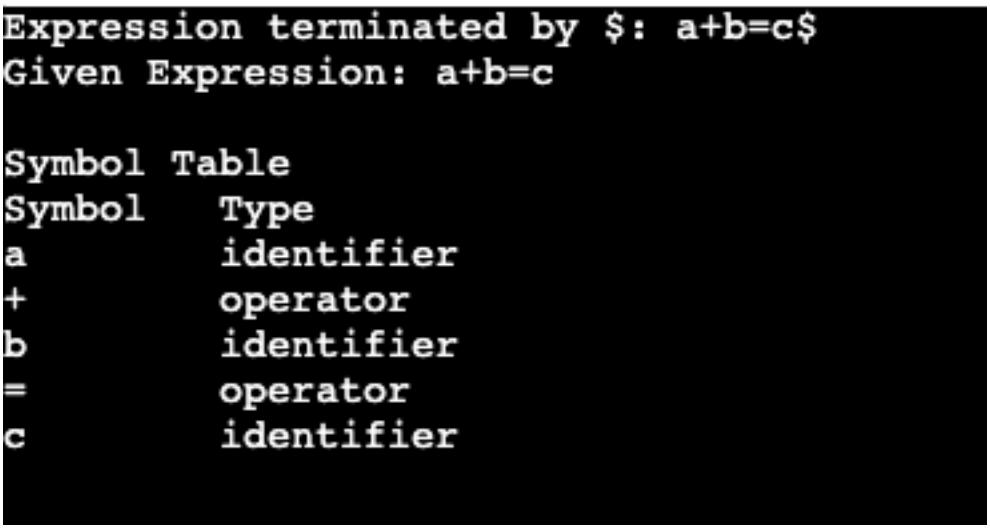
1. Write a C programme to implement symbol table.

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

#define MAX_SYMBOLS 15
#define MAX_INPUT_LENGTH 100

typedef struct {
   char symbol[MAX_INPUT_LENGTH];
   char type[12];
} Symbol;

int main() {
   int inputIndex = 0, exprIndex = 0, symbolIndex = 0, exprLength;
   char inputExpression[MAX_INPUT_LENGTH];
   Symbol symbolTable[MAX_SYMBOLS];

   printf("Expression terminated by $: ");
   while ((inputExpression[inputIndex] = getchar()) != '$' && inputIndex <
MAX_INPUT_LENGTH - 1) {
      inputIndex++;
   }
   inputExpression[inputIndex] = '\0';
```

```
    exprLength = inputIndex;

    printf("Given Expression: %s\n", inputExpression);
    printf("\nSymbol Table\n");
    printf("Symbol \t Type\n");

    while (exprIndex < exprLength) {
        char currentChar = inputExpression[exprIndex];

        if (isalpha(currentChar)) {
            int startOfIdentifier = exprIndex;

            while (exprIndex < exprLength && isalpha(inputExpression[exprIndex])) {
                exprIndex++;
            }

            int identifierLength = exprIndex - startOfIdentifier;
            strncpy(symbolTable[symbolIndex].symbol, &inputExpression[startOfIdentifier],
identifierLength);
            symbolTable[symbolIndex].symbol[identifierLength] = '\0';
            strcpy(symbolTable[symbolIndex].type, "identifier");
            symbolIndex++;

        } else if (currentChar == '+' || currentChar == '-' || currentChar == '*' || currentChar == '=')
{
            symbolTable[symbolIndex].symbol[0] = currentChar;
            symbolTable[symbolIndex].symbol[1] = '\0';
```

```
        strcpy(symbolTable[symbolIndex].type, "operator");
        symbolIndex++;
        exprIndex++;

      } else {
        exprIndex++;
      }
    }

    for (inputIndex = 0; inputIndex < symbolIndex; inputIndex++) {
      printf("%s \t %s\n", symbolTable[inputIndex].symbol, symbolTable[inputIndex].type);
    }
}
```

Output:



2. Write a C programme to develop a lexical analyzer to recognize a few patterns in C.

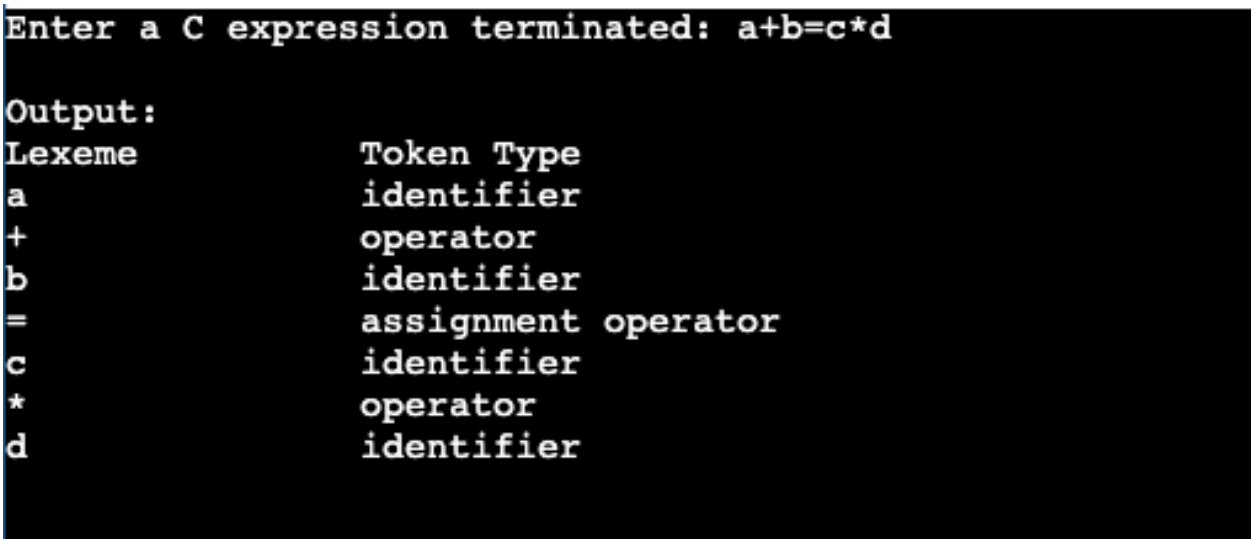
Code:

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

#define MAX_TOKENS 100
#define MAX_LEXEME_LENGTH 100

typedef struct {
   char lexeme[MAX_LEXEME_LENGTH];
   char tokenType[MAX_LEXEME_LENGTH];
} Token;

void addToken(Token tokenArray[], int *currentTokenCount, const char *lexeme, const char *tokenType) {
   strcpy(tokenArray[*currentTokenCount].lexeme, lexeme);
   strcpy(tokenArray[*currentTokenCount].tokenType, tokenType);
   (*currentTokenCount)++;
}

void analyzeExpression(char *expression, Token tokenArray[], int *currentTokenCount) {
   int charIndex = 0;
   while (expression[charIndex] != '\0') {
```

```
    if (isalpha(expression[charIndex])) {
        int startOfLexeme = charIndex;
        while (isalnum(expression[charIndex])) charIndex++;
        char lexeme[MAX_LEXEME_LENGTH];
        strncpy(lexeme, &expression[startOfLexeme], charIndex - startOfLexeme);
        lexeme[charIndex - startOfLexeme] = '\0';
        addToken(tokenArray, currentTokenCount, lexeme, "identifier");
    } else if (isdigit(expression[charIndex])) {
        int startOfLexeme = charIndex;
        while (isdigit(expression[charIndex])) charIndex++;
        char lexeme[MAX_LEXEME_LENGTH];
        strncpy(lexeme, &expression[startOfLexeme], charIndex - startOfLexeme);
        lexeme[charIndex - startOfLexeme] = '\0';
        addToken(tokenArray, currentTokenCount, lexeme, "integer literal");
    } else if (expression[charIndex] == '+' || expression[charIndex] == '-' ||
expression[charIndex] == '*' ||
            expression[charIndex] == '/' || expression[charIndex] == '=') {
        char lexeme[2] = {expression[charIndex], '\0'};
        const char *tokenType = (expression[charIndex] == '=') ? "assignment operator" :
"operator";
        addToken(tokenArray, currentTokenCount, lexeme, tokenType);
        charIndex++;
    } else if (expression[charIndex] == ';') {
        addToken(tokenArray, currentTokenCount, ";", "end of statement");
        charIndex++;
    } else {
        charIndex++;
```

```
        }
    }
}

int main() {
    char inputExpression[MAX_LEXEME_LENGTH];
    Token tokenArray[MAX_TOKENS];
    int currentTokenCount = 0;

    printf("Enter a C expression terminated: ");
    fgets(inputExpression, MAX_LEXEME_LENGTH, stdin);
    inputExpression[strcspn(inputExpression, "$")] = '\0';  // Remove the terminating $

    analyzeExpression(inputExpression, tokenArray, &currentTokenCount);

    printf("\nOutput:\n");
    printf("Lexeme\t\tToken Type\n");
    for (int tokenIndex = 0; tokenIndex < currentTokenCount; tokenIndex++) {
        printf("%s\t\t%s\n", tokenArray[tokenIndex].lexeme, tokenArray[tokenIndex].tokenType);
    }

    return 0;
}
```

Output:



Q3. Write a C programme to implement the Lexical Analyzer using Lex tool.

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
int isInCommentBlock = 0;
%}

identifier    [a-zA-Z][a-zA-Z0-9]*

%%

"#.*" {
```

```
    printf("\n%s is a Preprocessor Directive", yytext);
}

int|float|main|if|else|printf|scanf|for|char|getch|while {
    printf("\n%s is a Keyword", yytext);
}

"/*" {
    isInCommentBlock = 1;
}

"*/" {
    isInCommentBlock = 0;
}

{identifier}\( {
    if(!isInCommentBlock)
        printf("\nFunction: %s", yytext);
}

"{" {
    if(!isInCommentBlock)
        printf("\nBlock Begins");
}

"}" {
```

```
    if(!isInCommentBlock)
        printf("\nBlock Ends");
}

{identifier}(\[[0-9]*\])? {
    if(!isInCommentBlock)
        printf("\n%s is an Identifier", yytext);
}

\".*\" {
    if(!isInCommentBlock)
        printf("\n%s is a String", yytext);
}

[0-9]+ {
    if(!isInCommentBlock)
        printf("\n%s is a Number", yytext);
}

\)(\;)? {
    if(!isInCommentBlock) {
        printf("\t");
        ECHO;
        printf("\n");
    }
}
```

```
\( {
   ECHO;
}

"=" {
   if(!isInCommentBlock)
      printf("\n%s is an Assignment Operator", yytext);
}

"<="|">="|"<"|"=="|">" {
   if(!isInCommentBlock)
      printf("\n%s is a Relational Operator", yytext);
}

.|\n {
}

%%

int main(int argumentCount, char **argumentValues) {
   if(argumentCount > 1) {
      FILE *inputFile = fopen(argumentValues[1], "r");
      if(!inputFile) {
         printf("\nCould not open the file: %s", argumentValues[1]);
         exit(1);
```

```
        }
        yyin = inputFile;
    }
    yylex();
    printf("\n\n");
    return 0;
}

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

Output:

```
include is an Identifier
< is a Relational Operator
stdio is an Identifier
h is an Identifier
> is a Relational Operator
int is a Keyword
Function:       main(   )

Block Begins
int is a Keyword
x is an Identifier
= is an Assignment Operator
10 is a Number
float is a Keyword
y is an Identifier
= is an Assignment Operator
20 is a Number
5 is a Number
char is a Keyword
z is an Identifier
= is an Assignment Operator
A is an Identifier
Print is an Identifier
values is an Identifier
Function:       printf(
"x: %d, y: %f, z: %c\n" is a String
x is an Identifier
y is an Identifier
z is an Identifier      );
```

```
if is a Keyword(
x is an Identifier
>= is a Relational Operator
10 is a Number  )

Block Begins
x is an Identifier
= is an Assignment Operator
x is an Identifier
1 is a Number
Block Ends
return is an Identifier
0 is a Number
Block Ends
```

Q4. Write a C program for stack to use dynamic storage allocation.

Code:

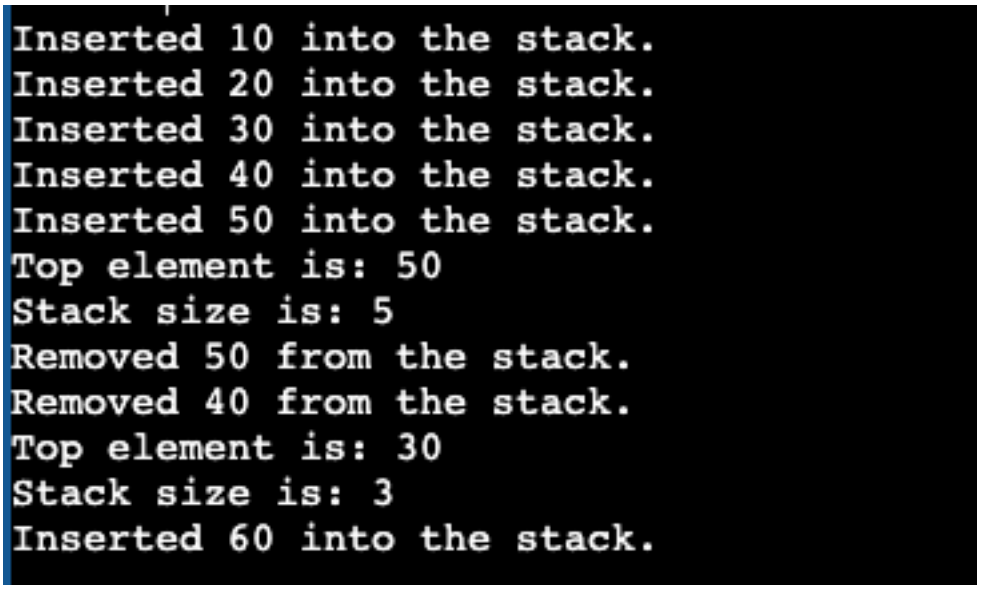
```
#include <stdio.h>

#include <stdlib.h>


typedef struct {

   int* elements;

   int capacity;

   int topIndex;

} Stack;
```

```
void initStack(Stack* stack, int size) {
    stack->capacity = size;
    stack->elements = (int*)malloc(stack->capacity * sizeof(int));
    stack->topIndex = -1;
}

void destroyStack(Stack* stack) {
    free(stack->elements);
}

void push(Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->elements[++stack->topIndex] = value;
    printf("Inserted %d into the stack.\n", value);
}

int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }
    printf("Removed %d from the stack.\n", stack->elements[stack->topIndex]);
```

```
    return stack->elements[stack->topIndex--];
}

int isEmpty(Stack* stack) {
    return stack->topIndex == -1;
}

int isFull(Stack* stack) {
    return stack->topIndex == stack->capacity - 1;
}

int getSize(Stack* stack) {
    return stack->topIndex + 1;
}

int peek(Stack* stack) {
    if (!isEmpty(stack)) {
        return stack->elements[stack->topIndex];
    } else {
        printf("Stack is empty\n");
        return -1;
    }
}

int main() {
    Stack stk;
```

```
    initStack(&stk, 5);

    push(&stk, 10);
    push(&stk, 20);
    push(&stk, 30);
    push(&stk, 40);
    push(&stk, 50);

    printf("Top element is: %d\n", peek(&stk));
    printf("Stack size is: %d\n", getSize(&stk));

    pop(&stk);
    pop(&stk);

    printf("Top element is: %d\n", peek(&stk));
    printf("Stack size is: %d\n", getSize(&stk));

    push(&stk, 60);

    destroyStack(&stk);
    return 0;
}
```

Output:


Inserted 10 into the stack.
Inserted 20 into the stack.
Inserted 30 into the stack.
Inserted 40 into the stack.
Inserted 50 into the stack.
Top element is: 50
Stack size is: 5
Removed 50 from the stack.
Removed 40 from the stack.
Top element is: 30
Stack size is: 3
Inserted 60 into the stack.