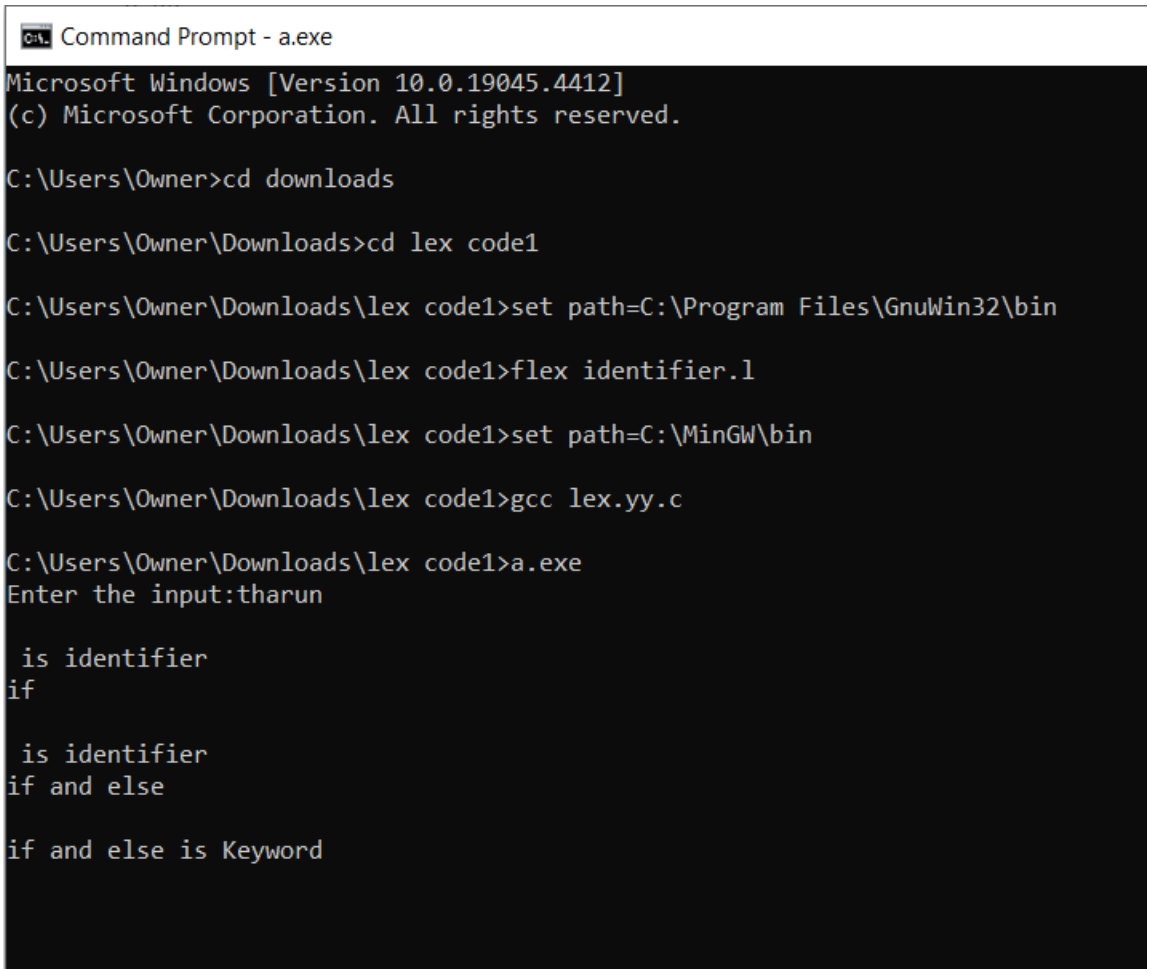
Day 5:

1.)check wheather identifier or not:

```
%{
%}
%%
[a-zA-Z][a-zA-Z0-9]+ {printf("¥n is identifier",yytext);}
.+ {printf("¥n%s is Keyword",yytext);}
%%
int yywrap(){}
int main()
{
printf("Enter the input:");
yylex();
}
```

output:



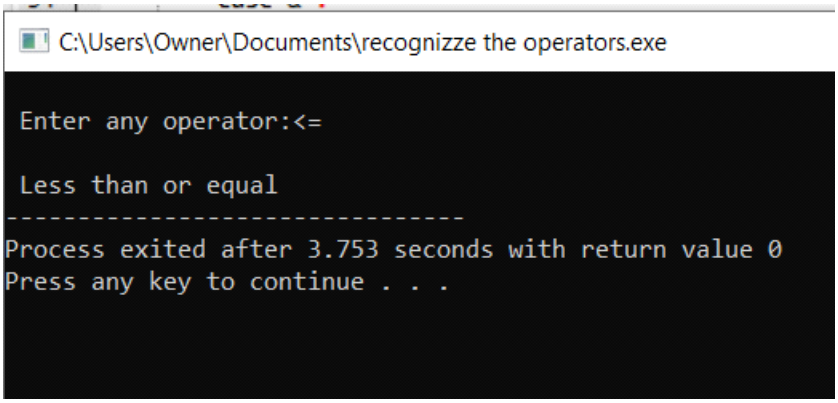
2.)In a class of Grade 3, Mathematics Teacher asked for the Acronym PEMDAS?. All of them are thinking for a while. A smart kid of the class Kishore of the class says it is Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction. Can you write a C Program to help the students to understand about the operator precedence parsing for an expression containing more than one operator, the order of evaluation depends on the order of operations.

```
#include<stdio.h>

#include<conio.h>

int main()

{

        char s[5];

        printf("¥n Enter any operator:");
```

```
gets(s);
switch(s[0])
{
        case'>':
                if(s[1]=='=')
                        printf("¥n Greater than or equal");
                else
                        printf("¥n Greater than");
                break;
        case'<':
                if(s[1]=='=')
                        printf("¥n Less than or equal");
                else
                        printf("¥nLess than");
                break;
        case'=':
                if(s[1]=='=')
                        printf("¥nEqual to");
                else
                        printf("¥nAssignment");
                break;
        case'!':
                if(s[1]=='=')
                        printf("¥nNot Equal");
                else
```

```
                printf("¥n Bit Not");
        break;
case'&':
        if(s[1]=='&')
                printf("¥nLogical AND");
        else
                printf("¥n Bitwise AND");
        break;
case'|':
        if(s[1]=='|')
                printf("¥nLogical OR");
        else
                printf("¥nBitwise OR");
        break;
case'+':
        printf("¥n Addition");
        break;
case'-':
        printf("¥nSubstraction");
        break;
case'*':
        printf("¥nMultiplication");
        break;
case'/':
        printf("¥nDivision");
```

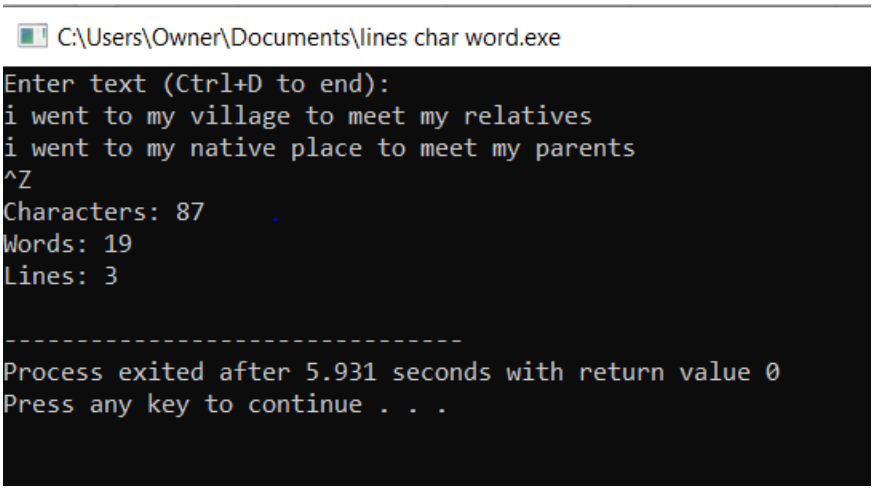
```
                break;
        case'%':
                printf("Modulus");
                break;
        default:
                printf("¥n Not a operator");
    }
}
```



3.)

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

int main() {
    char ch;
    int charCount = 0, wordCount = 0, lineCount = 0;
    int inWord = 0;

    printf("Enter text (Ctrl+D to end):¥n");

    while ((ch = getchar()) != EOF) {
```

```
        charCount++;

        if (ch == '¥n') {
            lineCount++;
        }

        if (isspace(ch)) {
            inWord = 0;
        } else if (!inWord) {
            inWord = 1;
            wordCount++;
        }
    }

    // To account for the last line if it doesn't end with a newline
    if (charCount > 0 && ch != '¥n') {
        lineCount++;
    }

    printf("Characters: %d¥n", charCount);
    printf("Words: %d¥n", wordCount);
    printf("Lines: %d¥n", lineCount);

    return 0;
}
```



4.)#include <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>

int tempVarCount = 0; // Counter for temporary variables

// Function to generate a new temporary variable

char* newTemp() {

**   static char temp[5];**

**   sprintf(temp, "t%d", tempVarCount++);**

**   return temp;**

}

// Function to print three-address code

void generateTAC(char* left, char op, char* right, char* result) {

**   printf("%s = %s %c %s¥n", result, left, op, right);**

```
}

// Recursive function to parse the expression and generate TAC
char* parseExpression(char* expr, int start, int end) {
    int i, lastOp = -1, opPosition = -1, parentheses = 0;

    // Find the last operator in the expression that is outside of any parentheses
    for (i = start; i <= end; i++) {
        if (expr[i] == '(') {
            parentheses++;
        } else if (expr[i] == ')') {
            parentheses--;
        } else if (parentheses == 0 && (expr[i] == '+' || expr[i] == '-')) {
            lastOp = i;
        } else if (parentheses == 0 && (expr[i] == '*' || expr[i] == '/') && lastOp == -1) {
            opPosition = i;
        }
    }

    if (lastOp == -1) {
        lastOp = opPosition;
    }
```

```
    if (lastOp == -1) {
        if (expr[start] == '(' && expr[end] == ')') {
            return parseExpression(expr, start + 1, end - 1);
        } else {
            char* operand = (char*)malloc(2);
            operand[0] = expr[start];
            operand[1] = '¥0';
            return operand;
        }
    }

    char* left = parseExpression(expr, start, lastOp - 1);
    char* right = parseExpression(expr, lastOp + 1, end);
    char op = expr[lastOp];
    char* result = newTemp();

    generateTAC(left, op, right, result);

    return result;
}

int main() {
    char expr[100];

    printf("Enter an arithmetic expression: ");
```

```
    scanf("%s", expr);

    int len = strlen(expr);
    parseExpression(expr, 0, len - 1);

    return 0;
}
```