CSA1445-CRYPTOGRAPHY AND NETWORK SECURITY FOR CYBER SECURITY

NAME: P.PANEENDRA

REG NO: 192321072

PROGRAM 1

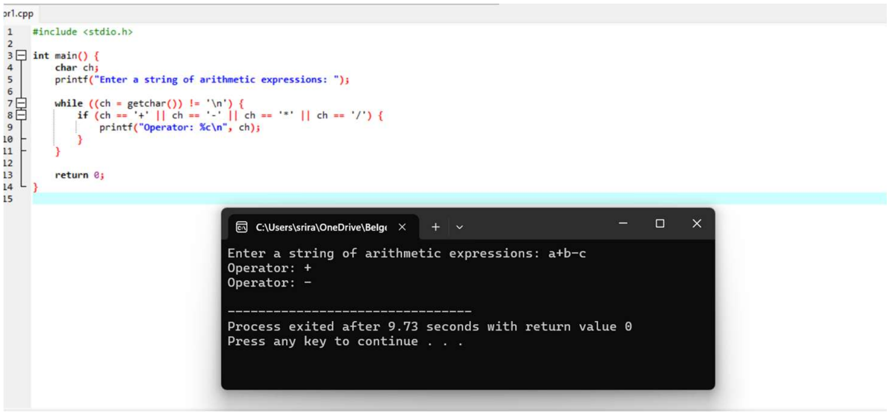
Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

Aim:
To identify and print the arithmetic operators +, -, *, and / from a given input string.

Code:

```
#include <stdio.h>

int main() {

    char ch;

    printf("Enter a string of arithmetic expressions: ");

    while ((ch = getchar()) != '\n') {

        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {

            printf("Operator: %c\n", ch);

        }

    }

    return 0;

}
```

Output:



```
#include <stdio.h>

int main() {
    char ch;
    printf("Enter a string of arithmetic expressions: ");

    while ((ch = getchar()) != '\n') {
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            printf("Operator: %c\n", ch);
        }
    }

    return 0;
}
```

```
Enter a string of arithmetic expressions: a+b-c
Operator: +
Operator: -

--------------------------------
Process exited after 9.73 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 2

Develop A Lexical Analyzer To Identify Whether A Given Line Is A Comment Or Not Using C

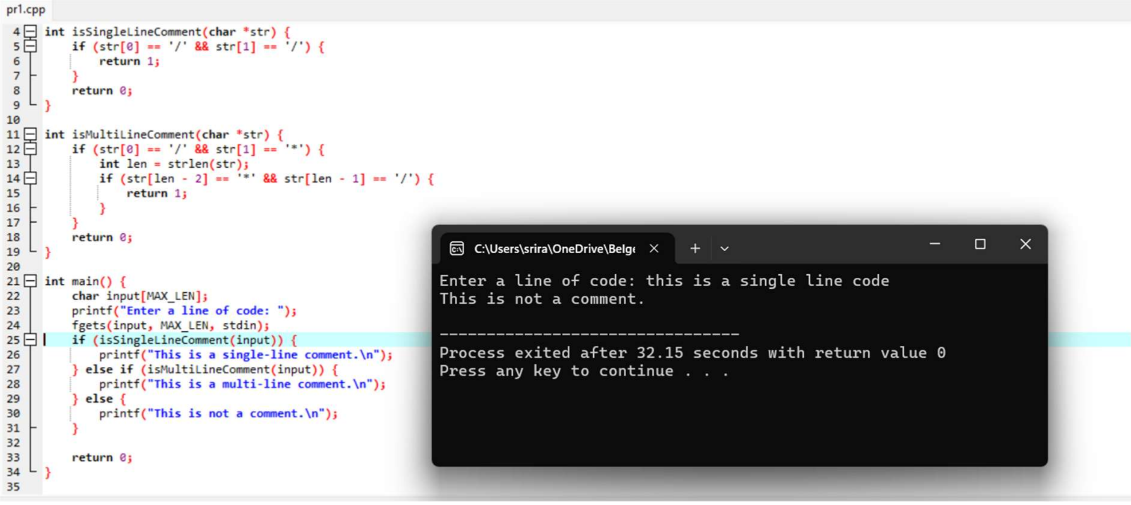
Aim:

To ignore spaces, tabs, newlines, and comments (// for single-line comments and /* */ for multi-line comments) while processing the input.

Code:

```
#include <stdio.h>
#include <string.h>
#define MAX_LEN 100
int isSingleLineComment(char *str) {
    if (str[0] == '/' && str[1] == '/') {
        return 1;
    }
    return 0;
}

int isMultiLineComment(char *str) {
    if (str[0] == '/' && str[1] == '*') {
        int len = strlen(str);
        if (str[len - 2] == '*' && str[len - 1] == '/') {
            return 1;
        }
    }
    return 0;
}

int main() {
    char input[MAX_LEN];
    printf("Enter a line of code: ");
    fgets(input, MAX_LEN, stdin);
    if (isSingleLineComment(input)) {
        printf("This is a single-line comment.\n");
    } else if (isMultiLineComment(input)) {
        printf("This is a multi-line comment.\n");
    } else {
        printf("This is not a comment.\n");
    }

    return 0;
}
```

Output:



```
int isSingleLineComment(char *str) {
    if (str[0] == '/' && str[1] == '/') {
        return 1;
    }
    return 0;
}

int isMultiLineComment(char *str) {
    if (str[0] == '/' && str[1] == '*') {
        int len = strlen(str);
        if (str[len - 2] == '*' && str[len - 1] == '/') {
            return 1;
        }
    }
    return 0;
}

int main() {
    char input[MAX_LEN];
    printf("Enter a line of code: ");
    fgets(input, MAX_LEN, stdin);
    if (isSingleLineComment(input)) {
        printf("This is a single-line comment.\n");
    } else if (isMultiLineComment(input)) {
        printf("This is a multi-line comment.\n");
    } else {
        printf("This is not a comment.\n");
    }

    return 0;
}
```

```
Enter a line of code: this is a single line code
This is not a comment.

--------------------------------
Process exited after 32.15 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 3

Design a lexical Analyzer for given language should ignore the redundant spaces, tabs and new lines and ignore comments using C

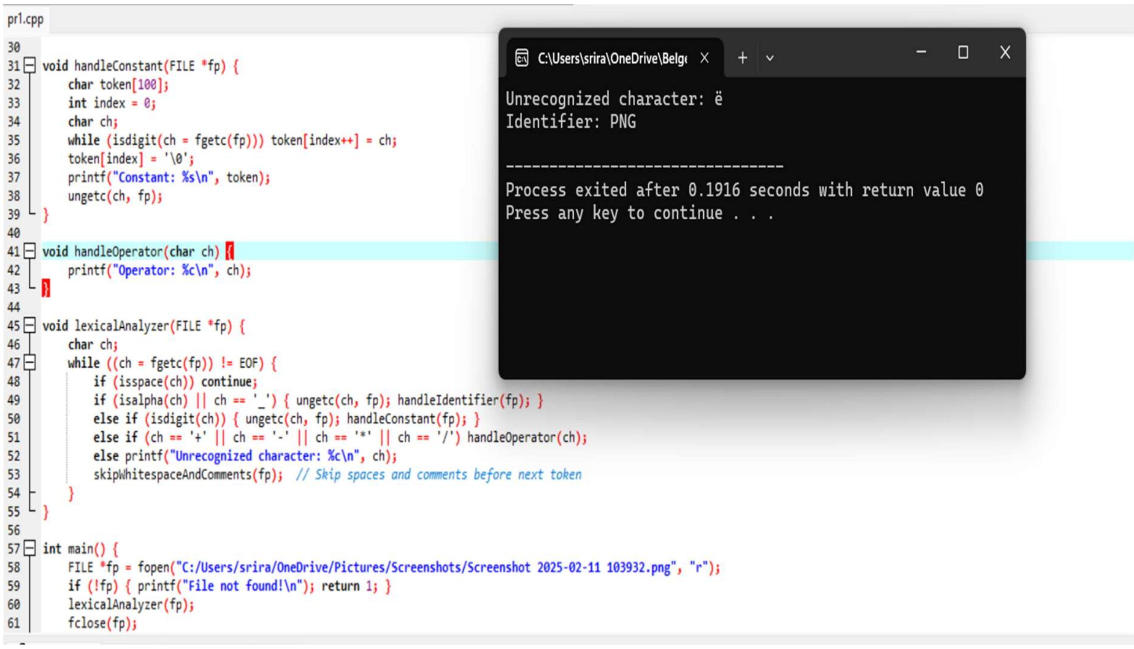
Aim:
To count the number of whitespace (spaces, tabs) and newline characters (\n) in a given input.

Code:

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

void skipWhitespaceAndComments(FILE *fp) {
    char ch;
    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) continue;  // Skip spaces, tabs, and newlines
        if (ch == '/' && fgetc(fp) == '/') {  // Skip single-line comment
            while ((ch = fgetc(fp)) != '\n' && ch != EOF);
        }
        else if (ch == '/' && fgetc(fp) == '*') {  // Skip multi-line comment
            while ((ch = fgetc(fp)) != '*' || fgetc(fp) != '/')
                if (ch == EOF) break;
        } else {
            ungetc(ch, fp);  // Valid character to process
            break;
        }
    }
}

void handleIdentifier(FILE *fp) {
    char token[100];
    int index = 0;
    char ch;
    while (isalpha(ch = fgetc(fp)) || ch == '_') token[index++] = ch;
    token[index] = '\0';
    printf("Identifier: %s\n", token);
    ungetc(ch, fp);
}
```

```
void handleConstant(FILE *fp) {
    char token[100];
    int index = 0;
    char ch;
    while (isdigit(ch = fgetc(fp))) token[index++] = ch;
    token[index] = '\0';
    printf("Constant: %s\n", token);
    ungetc(ch, fp);
}

void handleOperator(char ch) {
    printf("Operator: %c\n", ch);
}

void lexicalAnalyzer(FILE *fp) {
    char ch;
    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) continue;
        if (isalpha(ch) || ch == '_') { ungetc(ch, fp); handleIdentifier(fp); }
        else if (isdigit(ch)) { ungetc(ch, fp); handleConstant(fp); }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') handleOperator(ch);
        else printf("Unrecognized character: %c\n", ch);
        skipWhitespaceAndComments(fp);  // Skip spaces and comments before next token
    }
}

int main() {
    FILE *fp = fopen("source_code.txt", "r");
    if (!fp) { printf("File not found!\n"); return 1; }
    lexicalAnalyzer(fp);

    fclose(fp);

    return 0;

}
```

Output:



```
void handleConstant(FILE *fp) {
    char token[100];
    int index = 0;
    char ch;
    while (isdigit(ch = fgetc(fp))) token[index++] = ch;
    token[index] = '\0';
    printf("Constant: %s\n", token);
    ungetc(ch, fp);
}

void handleOperator(char ch) {
    printf("Operator: %c\n", ch);
}

void lexicalAnalyzer(FILE *fp) {
    char ch;
    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) continue;
        if (isalpha(ch) || ch == '_') { ungetc(ch, fp); handleIdentifier(fp); }
        else if (isdigit(ch)) { ungetc(ch, fp); handleConstant(fp); }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') handleOperator(ch);
        else printf("Unrecognized character: %c\n", ch);
        skipWhitespaceAndComments(fp);  // Skip spaces and comments before next token
    }
}

int main() {
    FILE *fp = fopen("C:/Users/srira/OneDrive/Pictures/Screenshots/Screenshot 2025-02-11 103932.png", "r");
    if (!fp) { printf("File not found!\n"); return 1; }
    lexicalAnalyzer(fp);
    fclose(fp);
```

```
Unrecognized character: ë
Identifier: PNG

--------------------------------
Process exited after 0.1916 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 4

Design a lexical Analyzer to validate operators to recognize the operators +,-,*,/ using regular arithmetic operators using C

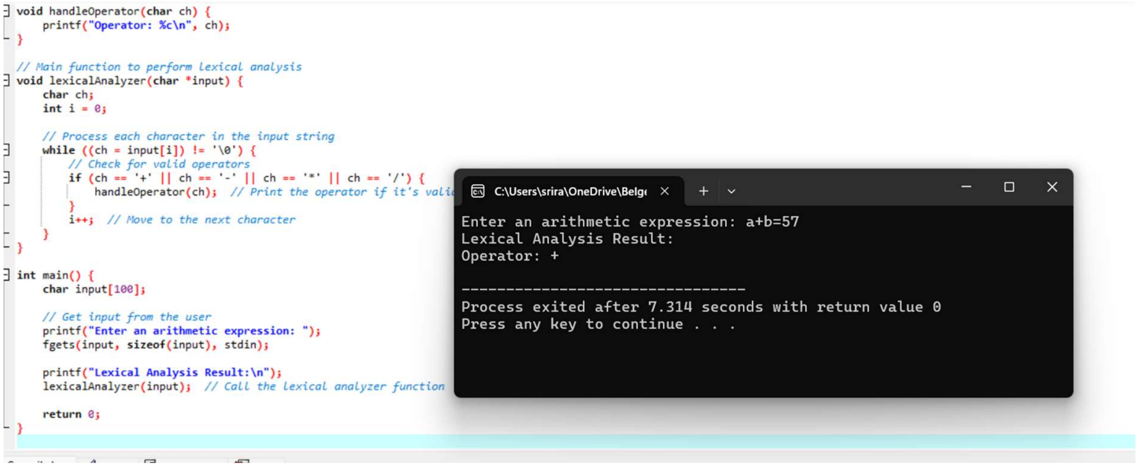
Aim:

The aim of this program is to design a **lexical analyzer** in C that recognizes and validates the basic arithmetic operators: +, -, *, /. The program will read an input string, process it character by character, and print out the recognized arithmetic operators.

Code:

```
 #include <stdio.h>
// Function to handle operators
void handleOperator(char ch) {
    printf("Operator: %c\n", ch);
}
// Main function to perform lexical analysis
void lexicalAnalyzer(char *input) {
    char ch;
    int i = 0;
    // Process each character in the input string
    while ((ch = input[i]) != '\0') {
        // Check for valid operators
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            handleOperator(ch);  // Print the operator if it's valid
        }
        i++;  // Move to the next character
    }
}
int main() {
    char input[100];
    // Get input from the user
    printf("Enter an arithmetic expression: ");
    fgets(input, sizeof(input), stdin);
    printf("Lexical Analysis Result:\n");
    lexicalAnalyzer(input);  // Call the lexical analyzer function
    return 0;
}
```

Output:



```
void handleOperator(char ch) {
    printf("Operator: %c\n", ch);
}

// Main function to perform lexical analysis
void lexicalAnalyzer(char *input) {
    char ch;
    int i = 0;

    // Process each character in the input string
    while ((ch = input[i]) != '\0') {
        // Check for valid operators
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            handleOperator(ch);  // Print the operator if it's vali
        }
        i++;  // Move to the next character
    }
}

int main() {
    char input[100];

    // Get input from the user
    printf("Enter an arithmetic expression: ");
    fgets(input, sizeof(input), stdin);

    printf("Lexical Analysis Result:\n");
    lexicalAnalyzer(input);  // Call the lexical analyzer function

    return 0;
}
```

```
Enter an arithmetic expression: a+b=57
Lexical Analysis Result:
Operator: +

--------------------------------
Process exited after 7.314 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 5

Design a lexical Analyzer to find the number of whitespaces and newline characters using C.

Aim:

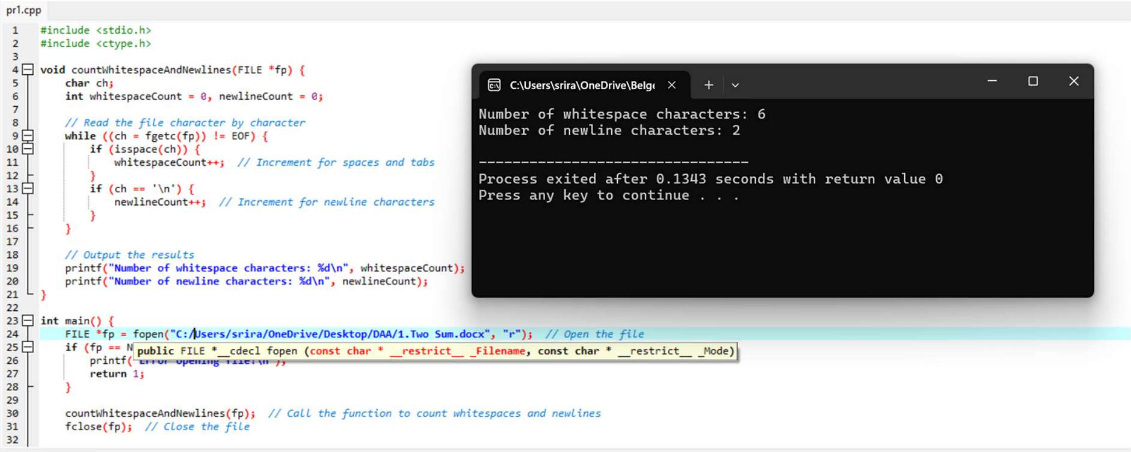
The aim of this program is to design a **lexical analyzer** in C that counts the number of whitespace characters (spaces and tabs) and newline characters (\n) in a given input string or file. This program will process the input character by character and keep track of the counts for whitespace and newline characters.

Code:

```
#include <stdio.h>

#include <ctype.h>


void countWhitespaceAndNewlines(FILE *fp) {

   char ch;

   int whitespaceCount = 0, newlineCount = 0;


   // Read the file character by character
```

```
    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) {
            whitespaceCount++;  // Increment for spaces and tabs
        }
        if (ch == '\n') {
            newlineCount++;  // Increment for newline characters
        }
    }

    // Output the results
    printf("Number of whitespace characters: %d\n", whitespaceCount);
    printf("Number of newline characters: %d\n", newlineCount);
}

int main() {
    FILE *fp = fopen("input.txt", "r");  // Open the file
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    countWhitespaceAndNewlines(fp);  // Call the function to count whitespaces and newlines
    fclose(fp);  // Close the file

    return 0;
}
```

Output:



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

void countWhitespaceAndNewlines(FILE *fp) {
    char ch;
    int whitespaceCount = 0, newlineCount = 0;

    // Read the file character by character
    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) {
            whitespaceCount++;  // Increment for spaces and tabs
        }
        if (ch == '\n') {
            newlineCount++;  // Increment for newline characters
        }
    }

    // Output the results
    printf("Number of whitespace characters: %d\n", whitespaceCount);
    printf("Number of newline characters: %d\n", newlineCount);
}

int main() {
    FILE *fp = fopen("C:/Users/srira/OneDrive/Desktop/DAA/1.Two Sum.docx", "r");  // Open the file
    if (fp == N public FILE *__cdecl fopen (const char * __restrict__ _Filename, const char * __restrict__ _Mode)
        printf(
        return 1;
    }

    countWhitespaceAndNewlines(fp);  // Call the function to count whitespaces and newlines
    fclose(fp);  // Close the file
```

```
Number of whitespace characters: 6
Number of newline characters: 2

--------------------------------
Process exited after 0.1343 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 6

Develop a lexical Analyzer to test whether a given identifier is valid or not using C.

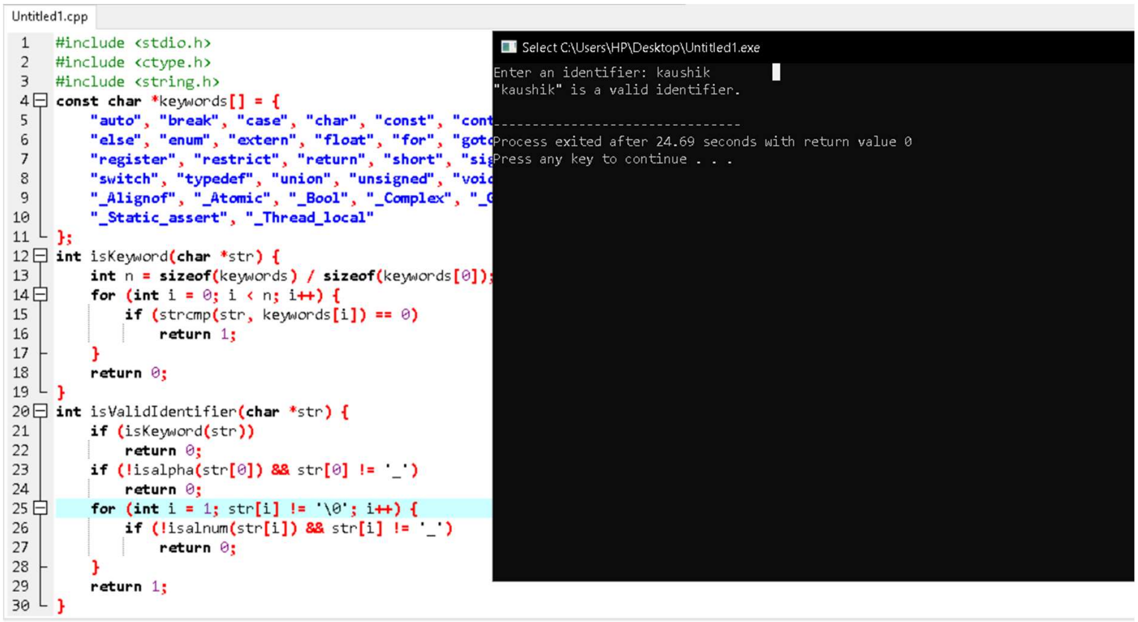
Aim:

To develop a lexical analyzer in C that checks whether a given identifier is valid according to the rules of the C programming language.

Code:

```c
#include <stdio.h>

#include <ctype.h>

#include <string.h>

// List of C keywords

const char *keywords[] = {

    "auto", "break", "case", "char", "const", "continue", "default", "do", "double",

    "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long",

    "register", "restrict", "return", "short", "signed", "sizeof", "static", "struct",

    "switch", "typedef", "union", "unsigned", "void", "volatile", "while", "_Alignas",

    "_Alignof", "_Atomic", "_Bool", "_Complex", "_Generic", "_Imaginary", "_Noreturn",

    "_Static_assert", "_Thread_local"

};

// Function to check if a given string is a keyword

int isKeyword(char *str) {

    int n = sizeof(keywords) / sizeof(keywords[0]);
    for (int i = 0; i < n; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1; // It is a keyword
    }
    return 0;
}
// Function to check if a given string is a valid identifier
int isValidIdentifier(char *str) {
    // Check if it's a keyword
    if (isKeyword(str))
        return 0;
    // Check if the first character is a letter or underscore
    if (!isalpha(str[0]) && str[0] != '_')
        return 0;
    // Check remaining characters
    for (int i = 1; str[i] != '\0'; i++) {
        if (!isalnum(str[i]) && str[i] != '_')
            return 0;
    }
    return 1;
}
int main() {
    char identifier[50];
    printf("Enter an identifier: ");
    scanf("%s", identifier);
    if (isValidIdentifier(identifier))
        printf("\"%s\" is a valid identifier.\n", identifier);
    else
        printf("\"%s\" is not a valid identifier.\n", identifier);
    return 0;     }
```

Output:



```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
const char *keywords[] = {
    "auto", "break", "case", "char", "const", "cont
    "else", "enum", "extern", "float", "for", "goto
    "register", "restrict", "return", "short", "sig
    "switch", "typedef", "union", "unsigned", "voic
    "_Alignof", "_Atomic", "_Bool", "_Complex", "_G
    "_Static_assert", "_Thread_local"
};
int isKeyword(char *str) {
    int n = sizeof(keywords) / sizeof(keywords[0]);
    for (int i = 0; i < n; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1;
    }
    return 0;
}
int isValidIdentifier(char *str) {
    if (isKeyword(str))
        return 0;
    if (!isalpha(str[0]) && str[0] != '_')
        return 0;
    for (int i = 1; str[i] != '\0'; i++) {
        if (!isalnum(str[i]) && str[i] != '_')
            return 0;
    }
    return 1;
}
```

```
Enter an identifier: kaushik
"kaushik" is a valid identifier.

--------------------------------
Process exited after 24.69 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 7

Aim:

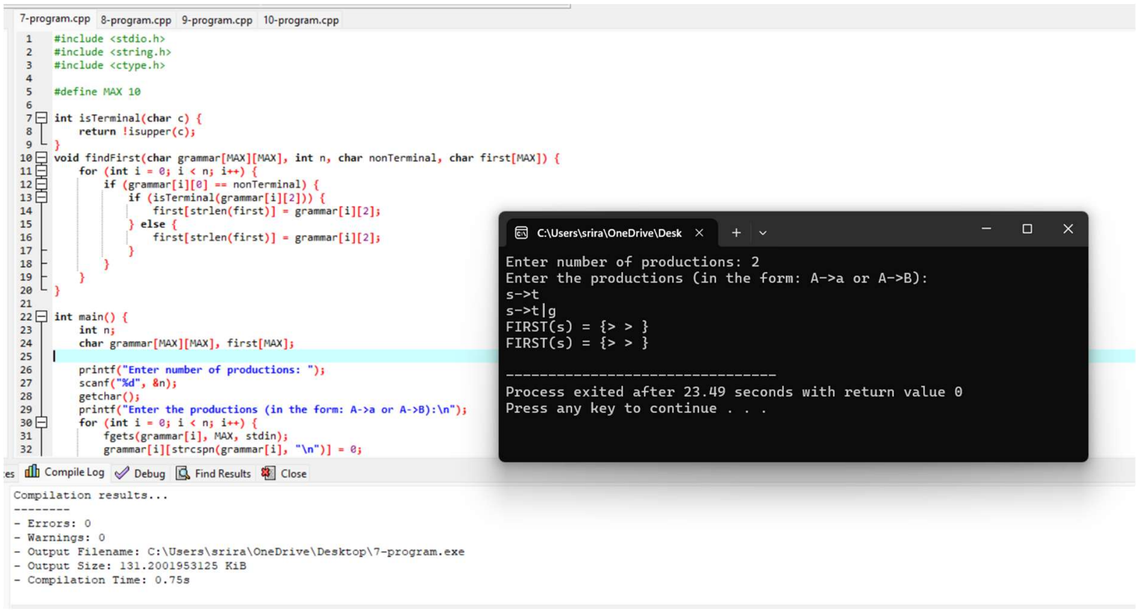
To implement a C program that computes the **FIRST()** sets for a given context-free grammar (CFG) as part of a predictive parser.

Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 10

int isTerminal(char c) {

  return !isupper(c);

}

void findFirst(char grammar[MAX][MAX], int n, char nonTerminal, char first[MAX]) {

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

    if (grammar[i][0] == nonTerminal) {

      if (isTerminal(grammar[i][2])) {
```

```
                first[strlen(first)] = grammar[i][2];
            } else {
                first[strlen(first)] = grammar[i][2];
            }
        }
    }
}
int main() {
    int n;
    char grammar[MAX][MAX], first[MAX];
    printf("Enter number of productions: ");
    scanf("%d", &n);
    getchar();
    printf("Enter the productions (in the form: A->a or A->B):\n");
    for (int i = 0; i < n; i++) {
        fgets(grammar[i], MAX, stdin);
        grammar[i][strcspn(grammar[i], "\n")] = 0;
    }
    for (int i = 0; i < n; i++) {
        char nonTerminal = grammar[i][0
        printf("FIRST(%c) = {", nonTerminal);
        memset(first, 0, sizeof(first));
        findFirst(grammar, n, nonTerminal, first);
                for (int j = 0; first[j] != '\0'; j++) {
            printf("%c ", first[j]);
        }
        printf("}\n");
    }
    return 0;
}
```

Output:



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

#define MAX 10

int isTerminal(char c) {
    return !isupper(c);
}
void findFirst(char grammar[MAX][MAX], int n, char nonTerminal, char first[MAX]) {
    for (int i = 0; i < n; i++) {
        if (grammar[i][0] == nonTerminal) {
            if (isTerminal(grammar[i][2])) {
                first[strlen(first)] = grammar[i][2];
            } else {
                first[strlen(first)] = grammar[i][2];
            }
        }
    }
}

int main() {
    int n;
    char grammar[MAX][MAX], first[MAX];

    printf("Enter number of productions: ");
    scanf("%d", &n);
    getchar();
    printf("Enter the productions (in the form: A->a or A->B):\n");
    for (int i = 0; i < n; i++) {
        fgets(grammar[i], MAX, stdin);
        grammar[i][strcspn(grammar[i], "\n")] = 0;
```

```
Enter number of productions: 2
Enter the productions (in the form: A->a or A->B):
s->t
s->t|g
FIRST(s) = {> > }
FIRST(s) = {> > }

--------------------------------
Process exited after 23.49 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\srira\OneDrive\Desktop\7-program.exe
- Output Size: 131.2001953125 KiB
- Compilation Time: 0.75s
```

PROGRAM 8

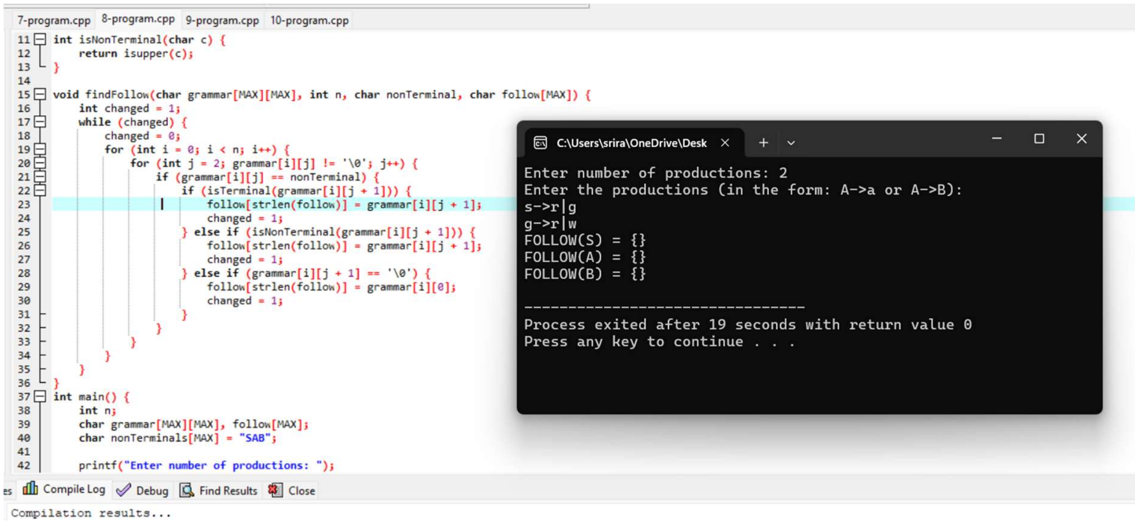
Aim:

To implement a C program that computes the **FOLLOW()** sets for a given context-free grammar (CFG) as part of a predictive parser. The **FOLLOW()** sets indicate which terminals can appear immediately to the right of a non-terminal in some sentential form.

Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 10

#define ALPHABET_SIZE 26

int isTerminal(char c) {

    return !isupper(c);

}

int isNonTerminal(char c) {

    return isupper(c);

}

void findFollow(char grammar[MAX][MAX], int n, char nonTerminal, char follow[MAX]) {
  int changed = 1;
    while (changed) {
        changed = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 2; grammar[i][j] != '\0'; j++) {
                if (grammar[i][j] == nonTerminal) {
                    if (isTerminal(grammar[i][j + 1])) {
                        follow[strlen(follow)] = grammar[i][j + 1];
                        changed = 1;
                    } else if (isNonTerminal(grammar[i][j + 1])) {
                        follow[strlen(follow)] = grammar[i][j + 1];
                        changed = 1;
                    } else if (grammar[i][j + 1] == '\0') {
                        follow[strlen(follow)] = grammar[i][0]; // Left-hand side non-terminal
                        changed = 1;
                    }
                }
            }
        }
    }
}
int main() {
    int n;
    char grammar[MAX][MAX], follow[MAX];
    char nonTerminals[MAX] = "SAB";
    printf("Enter number of productions: ");
    scanf("%d", &n);
    getchar();
    printf("Enter the productions (in the form: A->a or A->B):\n");
    for (int i = 0; i < n; i++) {
        fgets(grammar[i], MAX, stdin);
        grammar[i][strcspn(grammar[i], "\n")] = 0
    }
    for (int i = 0; i < MAX; i++) {
        follow[i] = '\0'; // Clear FOLLOW sets
    }
    follow[0] = '$';
    for (int i = 0; i < strlen(nonTerminals); i++) {
        char nonTerminal = nonTerminals[i];
        printf("FOLLOW(%c) = {", nonTerminal);
        memset(follow, 0, sizeof(follow)); // Clear the FOLLOW set
        findFollow(grammar, n, nonTerminal, follow);
        for (int j = 0; follow[j] != '\0'; j++) {
            printf("%c ", follow[j]);
        }
        printf("}\n");
    }
    return 0;
}
```

Output:



```
int isNonTerminal(char c) {
    return isupper(c);
}

void findFollow(char grammar[MAX][MAX], int n, char nonTerminal, char follow[MAX]) {
    int changed = 1;
    while (changed) {
        changed = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 2; grammar[i][j] != '\0'; j++) {
                if (grammar[i][j] == nonTerminal) {
                    if (isTerminal(grammar[i][j + 1])) {
                        follow[strlen(follow)] = grammar[i][j + 1];
                        changed = 1;
                    } else if (isNonTerminal(grammar[i][j + 1])) {
                        follow[strlen(follow)] = grammar[i][j + 1];
                        changed = 1;
                    } else if (grammar[i][j + 1] == '\0') {
                        follow[strlen(follow)] = grammar[i][0];
                        changed = 1;
                    }
                }
            }
        }
    }
}
int main() {
    int n;
    char grammar[MAX][MAX], follow[MAX];
    char nonTerminals[MAX] = "SAB";

    printf("Enter number of productions: ");
```

```
Enter number of productions: 2
Enter the productions (in the form: A->a or A->B):
s->r|g
g->r|w
FOLLOW(S) = {}
FOLLOW(A) = {}
FOLLOW(B) = {}

--------------------------------
Process exited after 19 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
```

PROGRAM 9

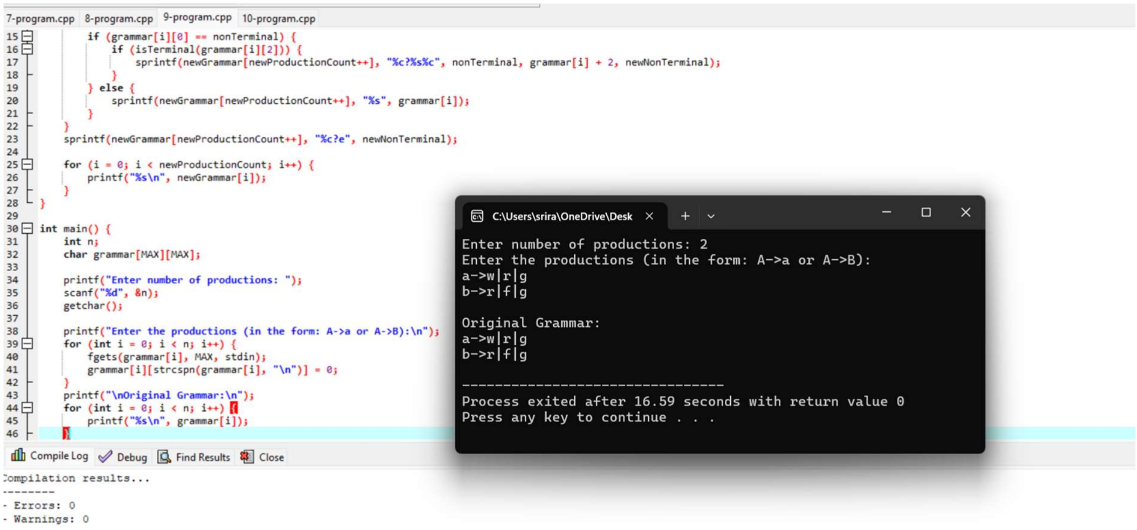
Aim:

To implement a C program that eliminates left recursion from a given context-free grammar (CFG). Left recursion occurs when a non-terminal on the left-hand side of a production rule appears at the beginning of its own right-hand side, leading to infinite recursion in recursive descent parsers.

Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 10

int isTerminal(char c) {

    return !isupper(c);

}

void eliminateLeftRecursion(char grammar[MAX][MAX], int *n, char nonTerminal) {

    char newNonTerminal = nonTerminal + '1'

    char newGrammar[MAX][MAX];

    int newProductionCount = 0;

    int i = 0, j = 0;

        for (i = 0; i < *n; i++) {

        if (grammar[i][0] == nonTerminal) {

            if (isTerminal(grammar[i][2])) {

                sprintf(newGrammar[newProductionCount++], "%c→%s%c", nonTerminal,
grammar[i] + 2, newNonTerminal);

            }

        } else {

            sprintf(newGrammar[newProductionCount++], "%s", grammar[i]);

        }

    }

    sprintf(newGrammar[newProductionCount++], "%c→ε", newNonTerminal);

    for (i = 0; i < newProductionCount; i++) {
        printf("%s\n", newGrammar[i]);
    }
}

int main() {
    int n;
    char grammar[MAX][MAX];

    printf("Enter number of productions: ");
    scanf("%d", &n);
    getchar();
    printf("Enter the productions (in the form: A->a or A->B):\n");
    for (int i = 0; i < n; i++) {
        fgets(grammar[i], MAX, stdin);
        grammar[i][strcspn(grammar[i], "\n")] = 0;
    }
    printf("\nOriginal Grammar:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", grammar[i]);
    }
    for (int i = 0; i < n; i++) {
        if (isupper(grammar[i][0])) {
            printf("\nAfter Eliminating Left Recursion for Non-Terminal %c:\n", grammar[i][0]);
            eliminateLeftRecursion(grammar, &n, grammar[i][0]);
        }
    }

    return 0;
}
```

Output:



```
            if (grammar[i][0] == nonTerminal) {
                if (isTerminal(grammar[i][2])) {
                    sprintf(newGrammar[newProductionCount++], "%c?%s%c", nonTerminal, grammar[i] + 2, newNonTerminal);
                }
            } else {
                sprintf(newGrammar[newProductionCount++], "%s", grammar[i]);
            }
        }
        sprintf(newGrammar[newProductionCount++], "%c?e", newNonTerminal);

        for (i = 0; i < newProductionCount; i++) {
            printf("%s\n", newGrammar[i]);
        }
    }

    int main() {
        int n;
        char grammar[MAX][MAX];

        printf("Enter number of productions: ");
        scanf("%d", &n);
        getchar();

        printf("Enter the productions (in the form: A->a or A->B):\n");
        for (int i = 0; i < n; i++) {
            fgets(grammar[i], MAX, stdin);
            grammar[i][strcspn(grammar[i], "\n")] = 0;
        }
        printf("\nOriginal Grammar:\n");
        for (int i = 0; i < n; i++) {
            printf("%s\n", grammar[i]);
        }
```

```
Enter number of productions: 2
Enter the productions (in the form: A->a or A->B):
a->w|r|g
b->r|f|g

Original Grammar:
a->w|r|g
b->r|f|g

--------------------------------
Process exited after 16.59 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
```

PROGRAM 10

Aim:

To implement a C program that eliminates **left factoring** from a given context-free grammar (CFG). Left factoring is a technique used to transform grammars that have common prefixes into a form where the choice between alternatives is made after the common prefix is processed.

Code:

```
#include <stdio.h>

#include <string.h>

#define MAX 10

#define MAX_PROD 100

int isTerminal(char c) {

    return !(c >= 'A' && c <= 'Z');

}

void eliminateLeftFactoring(char grammar[MAX_PROD][MAX], int *n) {

    char newGrammar[MAX_PROD][MAX];

    int newProductionCount = 0;

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

        for (int j = i + 1; j < *n; j++) {
            if (grammar[i][0] == grammar[j][0] && grammar[i][2] == grammar[j][2]) {
                char prefix[MAX] = {0};
                int k = 2;
                while (grammar[i][k] == grammar[j][k] && grammar[i][k] != '\0') {
                    prefix[k - 2] = grammar[i][k];
                    k++;
                }
                char newNonTerminal = grammar[i][0] + 1;
                sprintf(newGrammar[newProductionCount++], "%c→%s", newNonTerminal, prefix);
                sprintf(newGrammar[newProductionCount++], "%c→%s%c", grammar[i][0], prefix, newNonTerminal);
                sprintf(newGrammar[newProductionCount++], "%c→%s", newNonTerminal, grammar[i] + k);
                grammar[i][0] = '\0';
                grammar[j][0] = '\0';
            }
        }
    }
    printf("\nGrammar after Left Factoring:\n");
    for (int i = 0; i < newProductionCount; i++) {
        printf("%s\n", newGrammar[i]);
    }
}
int main() {
    int n;
    char grammar[MAX_PROD][MAX];
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    getchar();
    printf("Enter the productions in the form A->alpha:\n");
    for (int i = 0; i < n; i++) {
        fgets(grammar[i], MAX, stdin);
        grammar[i][strcspn(grammar[i], "\n")] = 0;
    }
    printf("\nOriginal Grammar:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", grammar[i]);
    }
    eliminateLeftFactoring(grammar, &n);

    return 0;
}
```

Output: