

# 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);

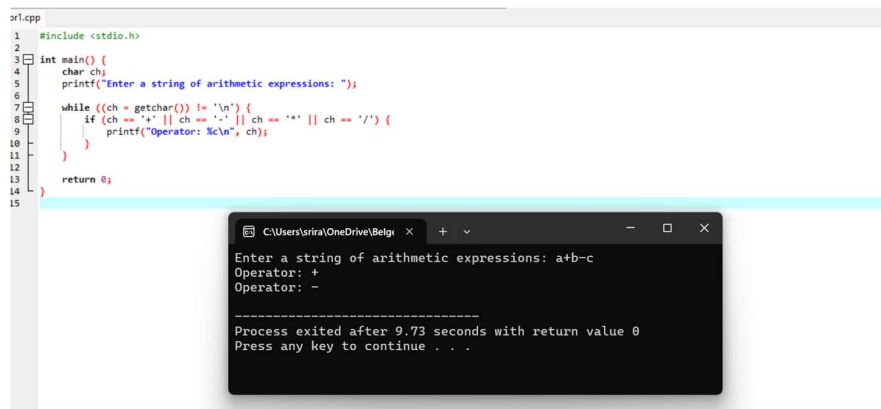
        }

    }

    return 0;

}
```

**Output:**

The image shows a screenshot of a C program in a text editor and its execution in a terminal window. The code in the editor is a simple program that prompts the user to enter a string of arithmetic expressions and then prints out each operator found. The terminal window shows the program being run, with the input 'a+b-c' and the output 'Operator: +', 'Operator: -'. The terminal also shows the process exiting after 9.73 seconds with a return value of 0.

```
orl.cpp
1  #include <stdio.h>
2
3  int main() {
4      char ch;
5      printf("Enter a string of arithmetic expressions: ");
6
7      while ((ch = getchar()) != '\n') {
8          if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
9              printf("Operator: %c\n", ch);
10         }
11     }
12
13     return 0;
14 }
15
```

```
C:\Users\srira\OneDrive\Belgu x + v - _ X
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:

The screenshot shows a C++ IDE with a file named 'pr1.cpp'. The code defines two functions: 'isSingleLineComment' which checks for a single-line comment starting with '//' and 'isMultiLineComment' which checks for a multi-line comment starting with '/\*'. The 'main' function prompts the user to enter a line of code, reads it using 'fgets', and then checks if it is a single-line or multi-line comment using the defined functions. The output window shows the program's execution: it prompts 'Enter a line of code: this is a single line code', prints 'This is not a comment.', and then displays a separator line followed by 'Process exited after 32.15 seconds with return value 0' and 'Press any key to continue . . .'. The line in the code where the single-line comment check is performed is highlighted in blue.

```

pr1.cpp
4 int isSingleLineComment(char *str) {
5     if (str[0] == '/' && str[1] == '/') {
6         return 1;
7     }
8     return 0;
9 }
10
11 int isMultiLineComment(char *str) {
12     if (str[0] == '/' && str[1] == '*') {
13         int len = strlen(str);
14         if (str[len - 2] == '*' && str[len - 1] == '/') {
15             return 1;
16         }
17     }
18     return 0;
19 }
20
21 int main() {
22     char input[MAX_LEN];
23     printf("Enter a line of code: ");
24     fgets(input, MAX_LEN, stdin);
25     if (isSingleLineComment(input)) {
26         printf("This is a single-line comment.\n");
27     } else if (isMultiLineComment(input)) {
28         printf("This is a multi-line comment.\n");
29     } else {
30         printf("This is not a comment.\n");
31     }
32     return 0;
33 }
34
35

```

Output Window:

```

C:\Users\sirira\OneDrive\Belgr... x + -
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:

```

pr1.cpp
30
31 void handleConstant(FILE *fp) {
32     char token[100];
33     int index = 0;
34     char ch;
35     while (isdigit(ch = fgetc(fp))) token[index++] = ch;
36     token[index] = '\0';
37     printf("Constant: %s\n", token);
38     ungetc(ch, fp);
39 }
40
41 void handleOperator(char ch) {
42     printf("Operator: %c\n", ch);
43 }
44
45 void lexicalAnalyzer(FILE *fp) {
46     char ch;
47     while ((ch = fgetc(fp)) != EOF) {
48         if (isspace(ch)) continue;
49         if (isalpha(ch) || ch == '_' || ch == '$') { ungetc(ch, fp); handleIdentifier(fp); }
50         else if (isdigit(ch)) { ungetc(ch, fp); handleConstant(fp); }
51         else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') handleOperator(ch);
52         else printf("Unrecognized character: %c\n", ch);
53         skipWhitespaceAndComments(fp); // Skip spaces and comments before next token
54     }
55 }
56
57 int main() {
58     FILE *fp = fopen("C:/Users/srira/OneDrive/Pictures/Screenshots/Screenshot 2025-02-11 103932.png", "r");
59     if (!fp) { printf("File not found!\n"); return 1; }
60     lexicalAnalyzer(fp);
61     fclose(fp);

```

```

C:\Users\sriira\OneDrive\Belgi X + - □ X
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;

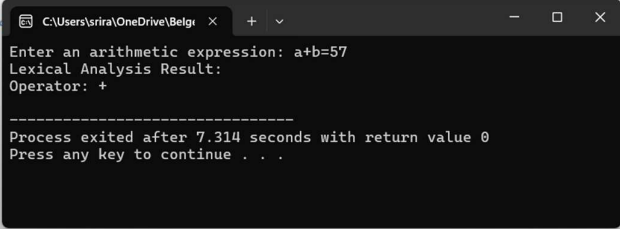
    // Process each character in the input string
    while ((ch = input[i]) != '\0') {
        // Check for valid operators
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            handleOperator(ch); // Print the operator if it's valid
        }
        i++; // Move to the next character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

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;
}
```



## 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);
}

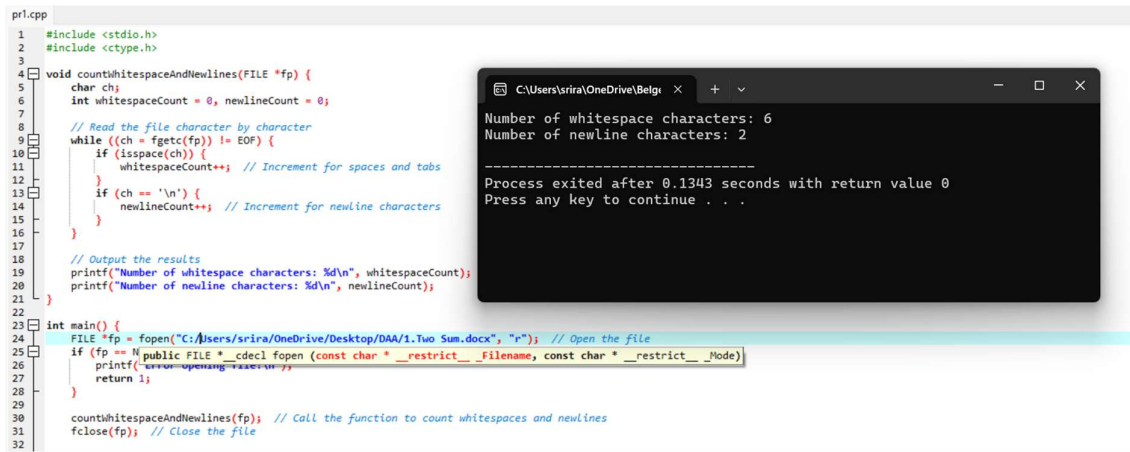
int main() {
    FILE *fp = fopen("input.txt", "r"); // Open the file
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

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

    return 0;
}

```

## Output:

The image shows a code editor on the left and a terminal window on the right. The code editor displays a C++ program named 'pr1.cpp' that counts whitespace and newline characters in a file. The terminal window shows the output of the program, which is: 'Number of whitespace characters: 6' and 'Number of newline characters: 2'. Below this, it shows 'Process exited after 0.1343 seconds with return value 0' and 'Press any key to continue . . .'.

```
pr1.cpp
1 #include <stdio.h>
2 #include <ctype.h>
3
4 void countWhitespaceAndNewlines(FILE *fp) {
5     char ch;
6     int whitespaceCount = 0, newlineCount = 0;
7
8     // Read the file character by character
9     while ((ch = fgetc(fp)) != EOF) {
10        if (isspace(ch)) {
11            whitespaceCount++; // Increment for spaces and tabs
12        }
13        if (ch == '\n') {
14            newlineCount++; // Increment for newline characters
15        }
16    }
17
18    // Output the results
19    printf("Number of whitespace characters: %d\n", whitespaceCount);
20    printf("Number of newline characters: %d\n", newlineCount);
21 }
22
23 int main() {
24     FILE *fp = fopen("C:/Users/srira/OneDrive/Desktop/DAA/1.Two Sum.docx", "r"); // Open the file
25     if (fp == NULL) {
26         printf("Error opening file!\n");
27         return 1;
28     }
29
30     countWhitespaceAndNewlines(fp); // Call the function to count whitespaces and newlines
31     fclose(fp); // Close the file
32 }
```

```
C:\Users\srira\OneDrive\Belgi
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:

```
#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:



The screenshot shows a C program in a text editor and its execution in a terminal. The program, named 'Untitled1.cpp', defines a list of C keywords and two functions: 'isKeyword' and 'isValidIdentifier'. The 'isValidIdentifier' function checks if a string is a valid C identifier by ensuring it doesn't start with a keyword and that its characters are either alphanumeric or underscores. The terminal output shows the program being executed, the user entering 'kaushik', and the program confirming it is a valid identifier before exiting.

```
1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4 const char *keywords[] = {
5     "auto", "break", "case", "char", "const", "continue",
6     "else", "enum", "extern", "float", "for", "goto",
7     "register", "restrict", "return", "short", "sizeof", "static",
8     "switch", "typedef", "union", "unsigned", "void",
9     "_Alignof", "_Atomic", "_Bool", "_Complex", "_Generic",
10    "_Imaginary", "_Intmax_t", "_Largest_width_t", "_Static_assert", "_Thread_local"
11 };
12 int isKeyword(char *str) {
13     int n = sizeof(keywords) / sizeof(keywords[0]);
14     for (int i = 0; i < n; i++) {
15         if (strcmp(str, keywords[i]) == 0)
16             return 1;
17     }
18     return 0;
19 }
20 int isValidIdentifier(char *str) {
21     if (isKeyword(str))
22         return 0;
23     if (!isalpha(str[0]) && str[0] != '_')
24         return 0;
25     for (int i = 1; str[i] != '\0'; i++) {
26         if (!isalnum(str[i]) && str[i] != '_')
27             return 0;
28     }
29     return 1;
30 }
```

Terminal Output:

```
Select C:\Users\HP\Desktop\Untitled1.exe
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();

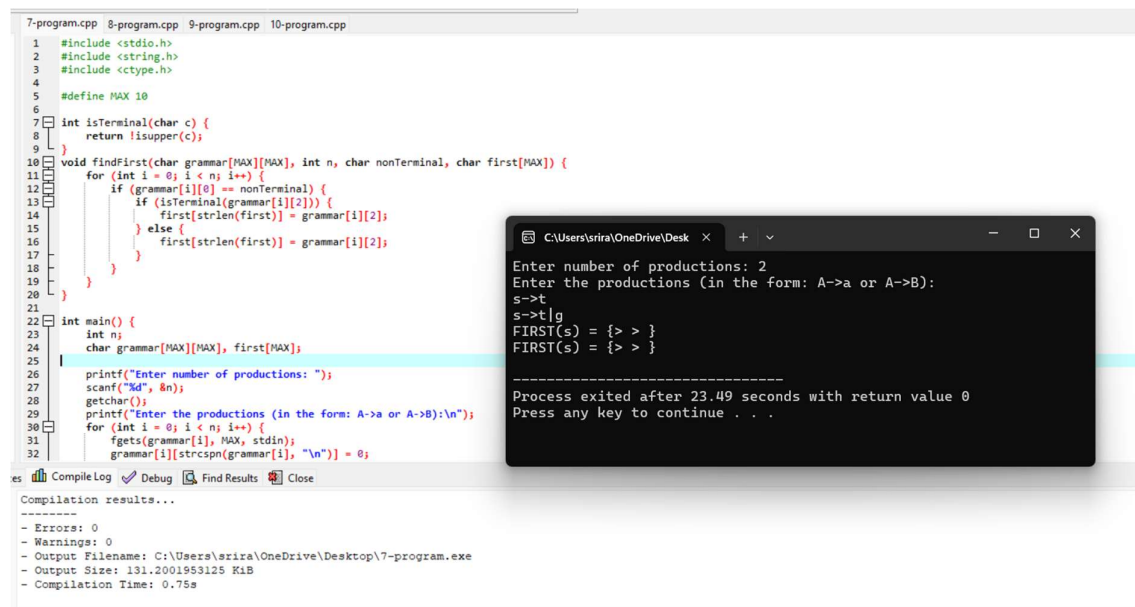
    printf("Enter the productions (in the form: A->a or A->B):\n");
    for (int i = 0; i < n; i++) {
        fgets(grammar[i], MAX, stdin);
        grammar[i][strlen(grammar[i])] = 0;
    }

    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:



The screenshot shows a C program editor with a file named 7-program.cpp. The code defines a function findFirst that calculates the first set for a given grammar. The main function prompts the user for the number of productions and the productions themselves. The output window shows the user inputting 2 productions: s->t and s->t|g. The first set for s is calculated as {>}. The program exits after 23.49 seconds with a return value of 0.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <ctype.h>
4
5 #define MAX 10
6
7 int isTerminal(char c) {
8     return !isupper(c);
9 }
10
11 void findFirst(char grammar[MAX][MAX], int n, char nonTerminal, char first[MAX]) {
12     for (int i = 0; i < n; i++) {
13         if (grammar[i][0] == nonTerminal) {
14             if (isTerminal(grammar[i][2])) {
15                 first[strlen(first)] = grammar[i][2];
16             } else {
17                 first[strlen(first)] = grammar[i][2];
18             }
19         }
20     }
21 }
22
23 int main() {
24     int n;
25     char grammar[MAX][MAX], first[MAX];
26
27     printf("Enter number of productions: ");
28     scanf("%d", &n);
29     getchar();
30     printf("Enter the productions (in the form: A->a or A->B):\n");
31     for (int i = 0; i < n; i++) {
32         fgets(grammar[i], MAX, stdin);
33         grammar[i][strlen(grammar[i], "\n")] = 0;
34     }
35
36     findFirst(grammar, n, nonTerminal, first);
37
38     printf("FIRST(s) = {> > }\n");
39     printf("FIRST(s) = {> > }\n");
40 }
```

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\arira\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:

The screenshot shows a C++ IDE with a file named 7-program.cpp. The code implements the FOLLOW set algorithm. The `findFollow` function iterates through the grammar rules and updates the FOLLOW sets for non-terminals. The `main` function prompts the user for the number of productions and the productions themselves, then calls `findFollow` and prints the results.

```

11 int isNonTerminal(char c) {
12     return isupper(c);
13 }
14
15 void findFollow(char grammar[MAX][MAX], int n, char nonTerminal, char follow[MAX]) {
16     int changed = 1;
17     while (changed) {
18         changed = 0;
19         for (int i = 0; i < n; i++) {
20             for (int j = 2; grammar[i][j] != '\0'; j++) {
21                 if (grammar[i][j] == nonTerminal) {
22                     if (isTerminal(grammar[i][j + 1])) {
23                         follow[strlen(follow)] = grammar[i][j + 1];
24                         changed = 1;
25                     } else if (isNonTerminal(grammar[i][j + 1])) {
26                         follow[strlen(follow)] = grammar[i][j + 1];
27                         changed = 1;
28                     } else if (grammar[i][j + 1] == '\0') {
29                         follow[strlen(follow)] = grammar[i][0];
30                         changed = 1;
31                     }
32                 }
33             }
34         }
35     }
36 }
37
38 int main() {
39     int n;
40     char grammar[MAX][MAX], follow[MAX];
41     char nonTerminals[MAX] = "SAB";
42     printf("Enter number of productions: ");

```

The output window shows the following text:

```

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 . . .

```



## 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]);
    }
}

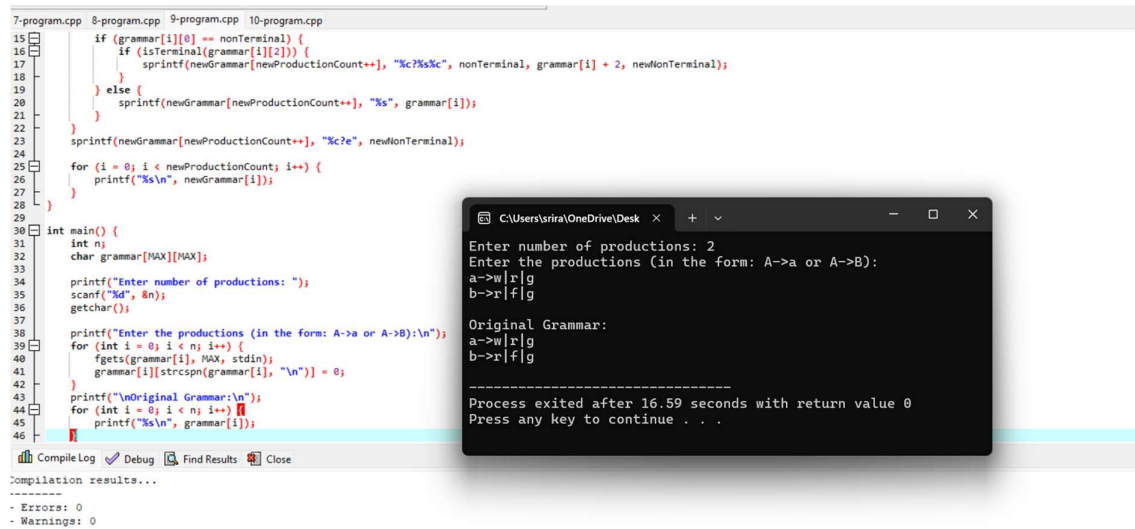
int main() {
    int n;
    char grammar[MAX][MAX];

    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:



The screenshot shows a C program in a code editor with tabs for 7-program.cpp, 8-program.cpp, 9-program.cpp, and 10-program.cpp. The code in 10-program.cpp implements a function to eliminate left factoring from a grammar. It defines constants MAX and MAX\_PROD, and functions isTerminal and eliminateLeftFactoring. The main function prompts the user for the number of productions and the productions themselves, then calls eliminateLeftFactoring and prints the original grammar. The terminal window shows the program's execution: it asks for the number of productions (2), the productions (a->w|r|g and b->r|f|g), and displays the original grammar. It also shows a message indicating the process exited after 16.59 seconds with a return value of 0.

```
15 if (grammar[i][0] == nonTerminal) {
16     if (isTerminal(grammar[i][2])) {
17         sprintf(newGrammar[newProductionCount++], "%c?%s%c", nonTerminal, grammar[i] + 2, newNonTerminal);
18     }
19 } else {
20     sprintf(newGrammar[newProductionCount++], "%s", grammar[i]);
21 }
22 }
23 sprintf(newGrammar[newProductionCount++], "%c?e", newNonTerminal);
24
25 for (i = 0; i < newProductionCount; i++) {
26     printf("%s\n", newGrammar[i]);
27 }
28
29
30 int main() {
31     int n;
32     char grammar[MAX][MAX];
33
34     printf("Enter number of productions: ");
35     scanf("%d", &n);
36     getchar();
37
38     printf("Enter the productions (in the form: A->a or A->B):\n");
39     for (int i = 0; i < n; i++) {
40         fgets(grammar[i], MAX, stdin);
41         grammar[i][strlen(grammar[i], "\n")] = 0;
42     }
43     printf("\nOriginal Grammar:\n");
44     for (int i = 0; i < n; i++) {
45         printf("%s\n", grammar[i]);
46     }
47
48     eliminateLeftFactoring(grammar, MAX_PROD, MAX, n);
49
50     printf("\n\n");
51 }
```

```
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 . . .
```

## 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:**