

13/02/2025 -LAB PROGRAMS

CSA1447 -COMPILER DESIGN FOR SYNTAX SMITH

1. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

Answer:

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

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

```
void lexicalAnalyzer(char code[]) {
```

```
    int i = 0;
```

```
    while (code[i] != '\0') {
```

```
        // Ignore spaces, tabs, and new lines
```

```
        if (isspace(code[i])) {
```

```
            i++;
```

```
            continue;
```

```
        }
```

```
        // Detect identifiers (starting with a letter or _)
```

```
        if (isalpha(code[i]) || code[i] == '_') {
```

```
            printf("Identifier: %c\n", code[i]);
```

```
            i++;
```

```
            continue;
```

```
        }
```

```
        // Detect constants (digits)
```

```
        if (isdigit(code[i])) {
```

```

        printf("Constant: %c\n", code[i]);

        i++;

        continue;
    }

    // Detect operators (+, -, *, /, =)
    if (code[i] == '+' || code[i] == '-' || code[i] == '*' || code[i] == '/' || code[i] == '=') {
        printf("Operator: %c\n", code[i]);

        i++;

        continue;
    }

    i++; // Move to next character
}

}

int main() {
    char code[100];

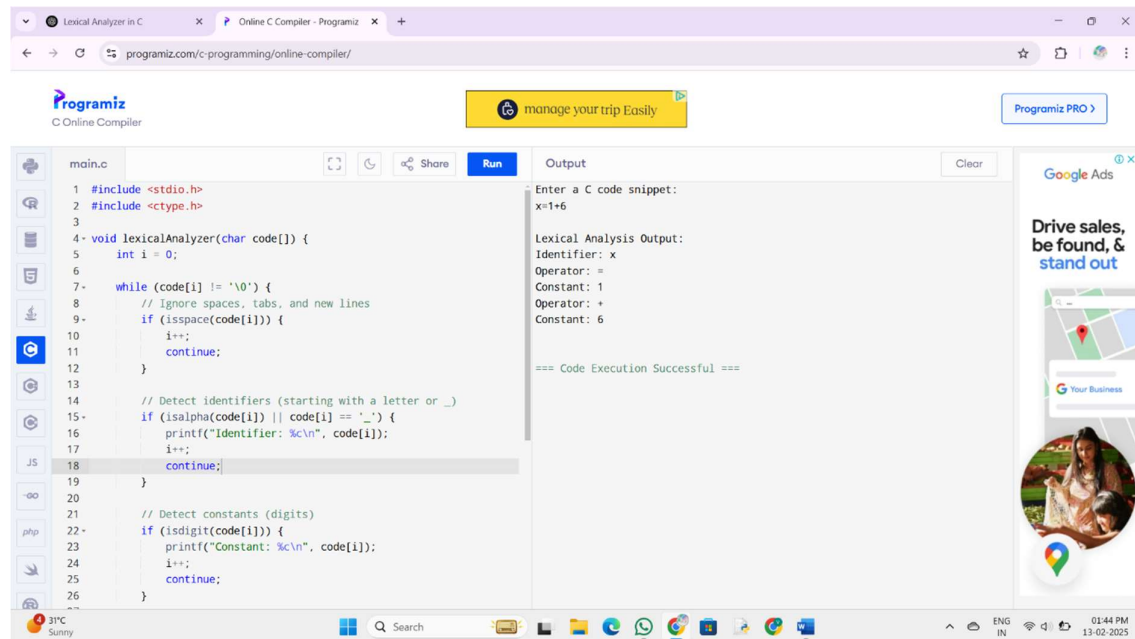
    printf("Enter a C code snippet:\n");
    fgets(code, sizeof(code), stdin);

    printf("\nLexical Analysis Output:\n");
    lexicalAnalyzer(code);

    return 0;
}

```

Op:



The screenshot shows the Programiz Online C Compiler interface. The code editor on the left contains a C program named 'main.c' that implements a lexical analyzer. The program includes `<stdio.h>` and `<ctype.h>`. It defines a function `lexicalAnalyzer(char code[])` that processes the input string 'x=1+6'. The function uses a `while` loop to iterate through the string, skipping spaces, tabs, and new lines. It then checks for identifiers (starting with a letter or underscore), operators, and constants (digits). The output window on the right shows the results of the lexical analysis: 'Identifier: x', 'Operator: =', 'Constant: 1', 'Operator: +', and 'Constant: 6'. Below the output, it states '=== Code Execution Successful ==='. The browser's address bar shows 'programiz.com/c-programming/online-compiler/'.

```
1 #include <stdio.h>
2 #include <ctype.h>
3
4 void lexicalAnalyzer(char code[]) {
5     int i = 0;
6
7     while (code[i] != '\0') {
8         // Ignore spaces, tabs, and new lines
9         if (isspace(code[i])) {
10             i++;
11             continue;
12         }
13
14         // Detect identifiers (starting with a letter or _)
15         if (isalpha(code[i]) || code[i] == '_') {
16             printf("Identifier: %c\n", code[i]);
17             i++;
18             continue;
19         }
20
21         // Detect constants (digits)
22         if (isdigit(code[i])) {
23             printf("Constant: %c\n", code[i]);
24             i++;
25             continue;
26         }
27     }
28 }
```

Output

Enter a C code snippet:  
x=1+6

Lexical Analysis Output:  
Identifier: x  
Operator: =  
Constant: 1  
Operator: +  
Constant: 6

=== Code Execution Successful ===

2. Extend the lexical Analyzer to Check comments, dened as follows in C:

- a) A comment begins with `//` and includes all characters until the end of that line.
  - b) A comment begins with `/*` and includes all characters through the next occurrence of the character sequence `*/`.
- Develop a lexical Analyzer to identify whether a given line is a comment or not.

Answer:

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

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

```
// Function to check if a line is a comment
```

```
void checkComment(char line[]) {
```

```
    int len = strlen(line);
```

```
    // Check for single-line comment (//)
```

```
    if (line[0] == '/' && line[1] == '/') {
```

```
        printf("This is a Single-line Comment.\n");
```

```
        return;
```

```

    }

    // Check for multi-line comment (/* ... */)
    if (line[0] == '/' && line[1] == '*') {
        if (len >= 4 && line[len - 2] == '*' && line[len - 1] == '/') {
            printf("This is a Multi-line Comment.\n");
        } else {
            printf("This is an Unclosed Multi-line Comment.\n");
        }
        return;
    }

    // If not a comment
    printf("This is NOT a comment.\n");
}

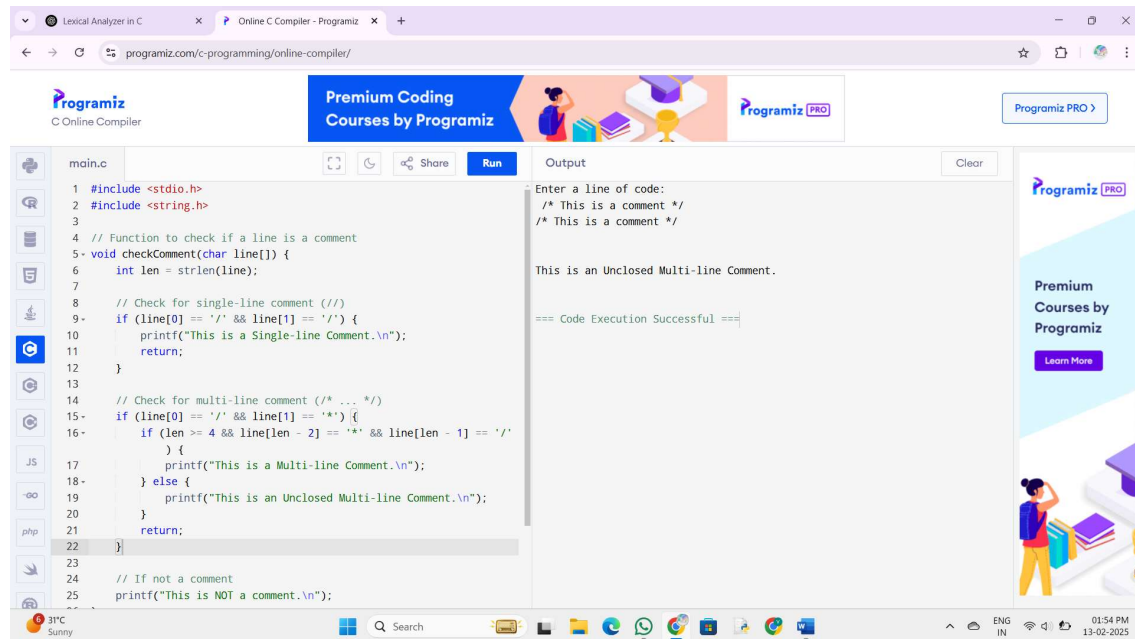
int main() {
    char line[200];

    printf("Enter a line of code:\n");
    fgets(line, sizeof(line), stdin);

    checkComment(line);
    return 0;
}

```

Op:



The screenshot shows a web browser window with the URL `programiz.com/c-programming/online-compiler/`. The page features the Programiz logo and a navigation bar with "Premium Coding Courses by Programiz" and a "Programiz PRO" button. The main content area is divided into two panels. The left panel, titled "main.c", contains C code for a lexical analyzer that checks for single-line and multi-line comments. The right panel, titled "Output", shows the execution results: "Enter a line of code: /\* This is a comment \*/", "/\* This is a comment \*/", "This is an Unclosed Multi-line Comment.", and "=== Code Execution Successful ===". A sidebar on the right promotes "Premium Courses by Programiz" with a "Learn More" button. The bottom of the browser window shows a Windows taskbar with various icons and a system tray displaying "31°C Sunny" and the time "01:54 PM 13-02-2025".

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Function to check if a line is a comment
5 void checkComment(char line[]) {
6     int len = strlen(line);
7
8     // Check for single-line comment (//)
9     if (line[0] == '/' && line[1] == '/') {
10         printf("This is a Single-line Comment.\n");
11         return;
12     }
13
14     // Check for multi-line comment (/* ... */)
15     if (line[0] == '/' && line[1] == '*') {
16         if (len >= 4 && line[len - 2] == '*' && line[len - 1] == '/') {
17             printf("This is a Multi-line Comment.\n");
18         } else {
19             printf("This is an Unclosed Multi-line Comment.\n");
20         }
21         return;
22     }
23
24     // If not a comment
25     printf("This is NOT a comment.\n");
26 }
```

3. Design a lexical Analyzer to validate operators to recognize the operators +,-,\*,/ using regular Arithmetic operators .

Answer

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

```
// Function to check if the input is a valid operator
```

```
void checkOperator(char op) {
```

```
    switch(op) {
```

```
        case '+':
```

```
            printf("Valid Operator: Addition (+)\n");
```

```
            break;
```

```
        case '-':
```

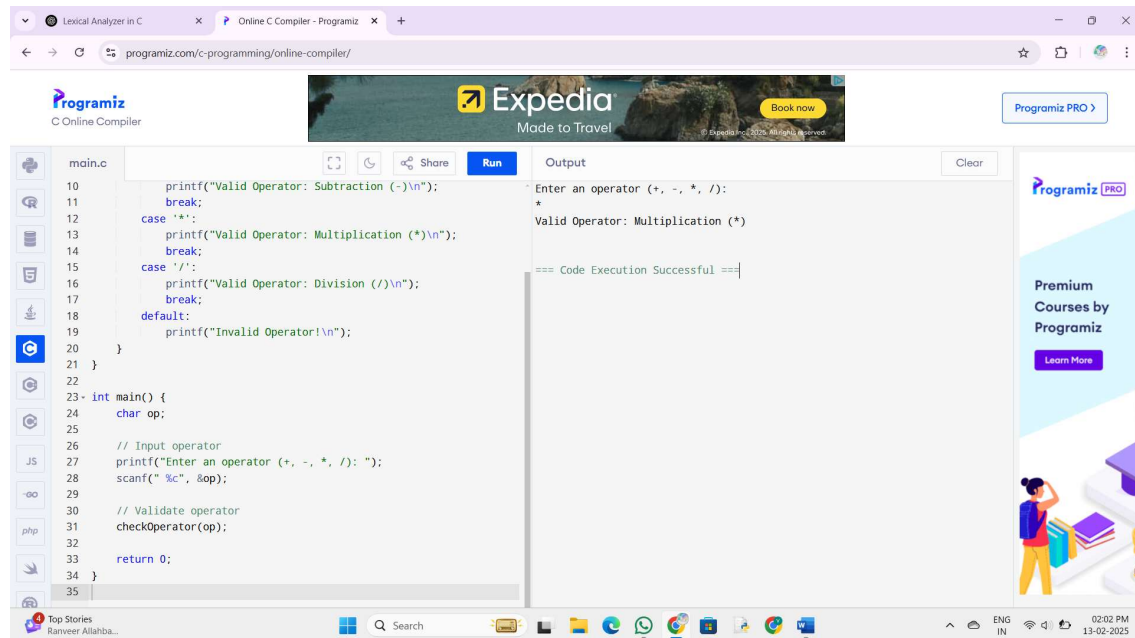
```
            printf("Valid Operator: Subtraction (-)\n");
```

```
            break;
```

```
        case '*':  
            printf("Valid Operator: Multiplication (*)\n");  
            break;  
        case '/':  
            printf("Valid Operator: Division (/)\n");  
            break;  
        default:  
            printf("Invalid Operator!\n");  
    }  
}
```

```
int main() {  
    char op;  
  
    // Input operator  
    printf("Enter an operator (+, -, *, /): ");  
    scanf(" %c", &op);  
  
    // Validate operator  
    checkOperator(op);  
  
    return 0;  
}
```

Op:



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

Answer:

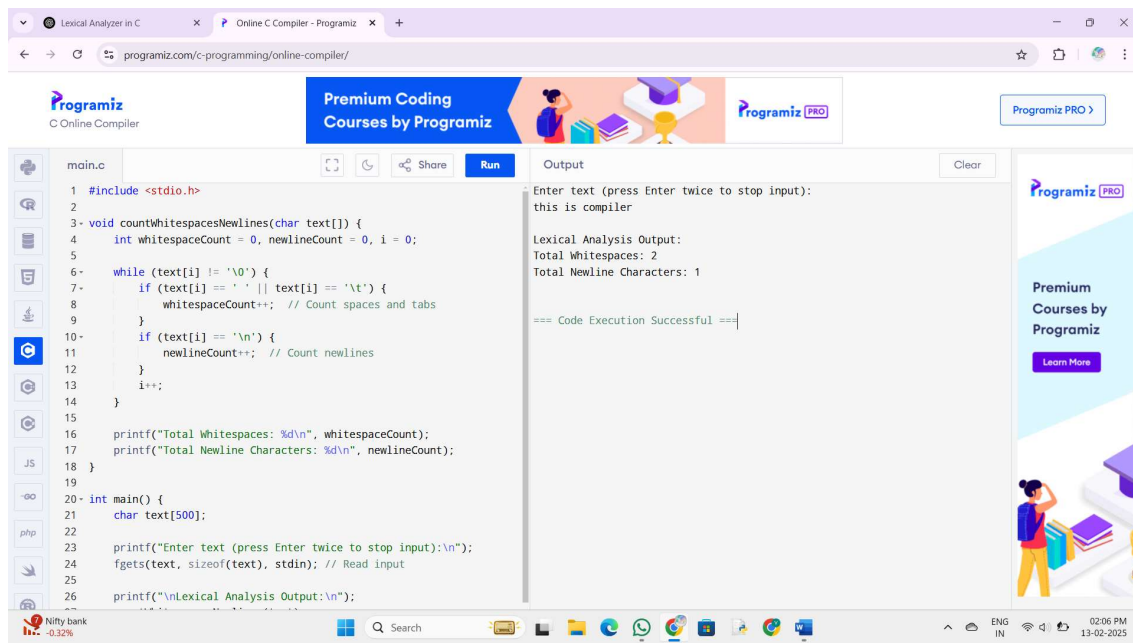
```
#include <stdio.h>
```

```
void countWhitespacesNewlines(char text[]) {  
  
    int whitespaceCount = 0, newlineCount = 0, i = 0;  
  
    while (text[i] != '\0') {  
  
        if (text[i] == ' ' || text[i] == '\t') {  
  
            whitespaceCount++; // Count spaces and tabs  
  
        }  
  
        if (text[i] == '\n') {  
  
            newlineCount++; // Count newlines  
  
        }  
  
    }  
  
}
```

```
        i++;  
    }  
  
    printf("Total Whitespaces: %d\n", whitespaceCount);  
    printf("Total Newline Characters: %d\n", newlineCount);  
}  
  
int main() {  
    char text[500];  
  
    printf("Enter text (press Enter twice to stop input):\n");  
    fgets(text, sizeof(text), stdin); // Read input  
  
    printf("\nLexical Analysis Output:\n");  
    countWhitespacesNewlines(text);  
  
    return 0;  
}
```

Op:





The screenshot shows a web browser with the URL `programiz.com/c-programming/online-compiler/`. The page features the Programiz logo and a navigation bar with "Premium Coding Courses by Programiz". The main content area displays a C program in a text editor, with line numbers 1 through 26. The code defines a function `countWhitespacesNewLines` that iterates through a string, counting spaces and newlines. The `main` function prompts the user to enter text and prints the results. The output window on the right shows the execution results: "Enter text (press Enter twice to stop input): this is compiler", "Lexical Analysis Output:", "Total Whitespaces: 2", "Total Newline Characters: 1", and "=== Code Execution Successful ===". A sidebar on the right promotes "Premium Courses by Programiz".

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

Answer:

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

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

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

```
// List of C keywords (for additional validation)
```

```
const char *keywords[] = {
```

```
    "int", "float", "char", "double", "if", "else", "while", "return", "for",
```

```
    "switch", "case", "break", "continue", "void", "static", "struct", "typedef"
```

```
};
```

```
// Function to check if an identifier is a keyword
```

```
int isKeyword(char *word) {
```

```
    int numKeywords = sizeof(keywords) / sizeof(keywords[0]);
```

```

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

        if (strcmp(word, keywords[i]) == 0) {

            return 1; // It is a keyword

        }

    }

    return 0;

}

// Function to check if the identifier is valid

int isValidIdentifier(char *id) {

    if (!isalpha(id[0]) && id[0] != '_') {

        return 0; // Must start with a letter or underscore

    }

    for (int i = 1; id[i] != '\0'; i++) {

        if (!isalnum(id[i]) && id[i] != '_') {

            return 0; // Can contain only letters, digits, and underscores

        }

    }

    if (isKeyword(id)) {

        return 0; // Identifiers cannot be C keywords

    }

```

```
        return 1;
    }

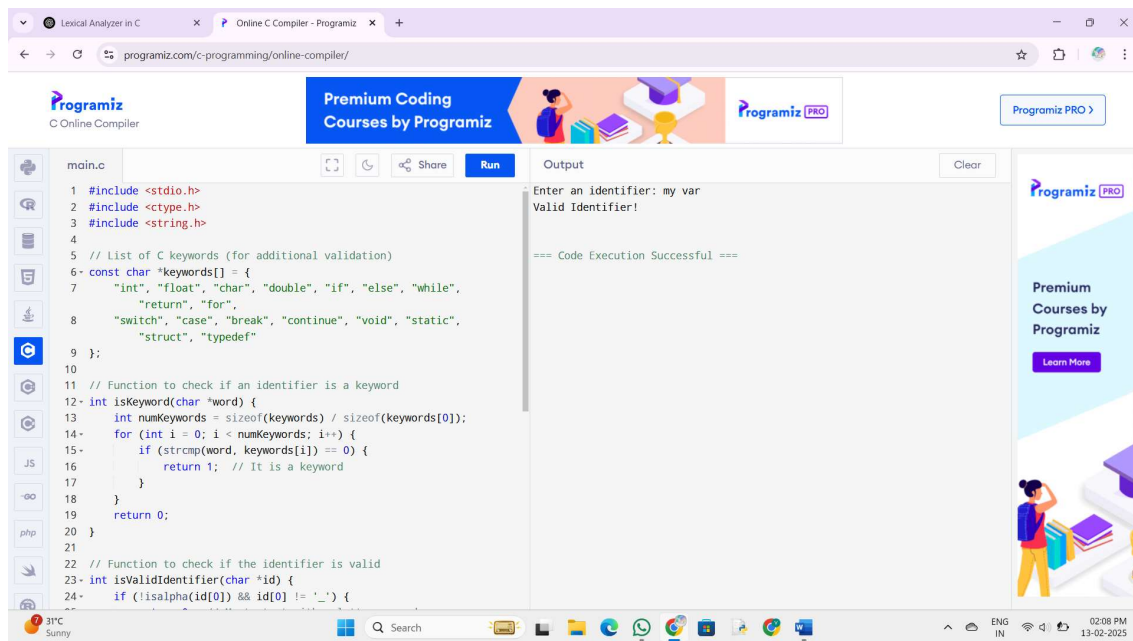
int main() {
    char identifier[50];

    printf("Enter an identifier: ");
    scanf("%s", identifier);

    if (isValidIdentifier(identifier)) {
        printf("Valid Identifier!\n");
    } else {
        printf("Invalid Identifier!\n");
    }

    return 0;
}
```

Op:



6. Implement a C program to eliminate left recursion.

Answer:

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

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

```
void eliminateLeftRecursion(char nonTerminal, char alpha[], char beta[]) {
    char newNonTerminal = nonTerminal + '1'; // Create new non-terminal (A → A')

    printf("\nAfter Eliminating Left Recursion:\n");
    printf("%c → %s%c\n", nonTerminal, beta, newNonTerminal);
    printf("%c' → %s%c' | ε\n", newNonTerminal, alpha, newNonTerminal);
}
```

```
int main() {
    char production[50], nonTerminal, alpha[20], beta[20];

    printf("Enter a left-recursive production (e.g., A→Aa|b): ");
```

```

scanf("%s", production);

nonTerminal = production[0]; // Extract non-terminal (e.g., A)

int i = 3, j = 0, isLeftRecursive = 0;

// Extract alpha and beta
while (production[i] != '\0') {
    if (production[i] == nonTerminal) {
        isLeftRecursive = 1;
        i++; // Skip the non-terminal
        while (production[i] != '|' && production[i] != '\0') {
            alpha[j++] = production[i++];
        }
        alpha[j] = '\0';
    } else {
        j = 0;
        while (production[i] != '|' && production[i] != '\0') {
            beta[j++] = production[i++];
        }
        beta[j] = '\0';
    }
    if (production[i] == '|') i++; // Move to the next part
}

if (isLeftRecursive) {
    eliminateLeftRecursion(nonTerminal, alpha, beta);
} else {

```

```

        printf("No Left Recursion Found!\n");
    }

    return 0;
}

```

Op:

The screenshot shows a web browser with the URL 'programiz.com/c-programming/online-compiler/'. The page features a 'Programiz' logo and a 'Premium Coding Courses by Programiz' banner. The main content area displays a C program in a code editor. The code includes headers for `<stdio.h>` and `<string.h>`, and defines a function `eliminateLeftRecursion` that takes a non-terminal, a common prefix, and two result arrays. The `main` function prompts the user to enter a left-recursive production, reads it, and calls the `eliminateLeftRecursion` function. The output window shows the input 'A->Aa|b' and the resulting productions 'A -> br' and 'r' -> ar' | ε'. A message at the bottom of the output window states '=== Code Execution Successful ==='.

7. Implement a C program to eliminate left factoring.

Answer:

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

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

```
// Function to remove left factoring
```

```
void eliminateLeftFactoring(char nonTerminal, char commonPrefix[], char beta1[], char
beta2[]) {
```

```
    char newNonTerminal = nonTerminal + '1'; // Create new non-terminal (A -> A')
```

```

printf("\nAfter Eliminating Left Factoring:\n");

printf("%c → %s%c\n", nonTerminal, commonPrefix, newNonTerminal);

printf("%c' → %s | %s\n", newNonTerminal, beta1, beta2);
}

int main() {

    char production[50], nonTerminal, commonPrefix[20], beta1[20], beta2[20];

    printf("Enter a left-factored production (e.g., A->abc|abd): ");

    scanf("%s", production);

    nonTerminal = production[0]; // Extract non-terminal (e.g., A)

    int i = 3, j = 0;

    // Extract the common prefix

    while (production[i] != '\0' && production[i] != '|') {

        commonPrefix[j++] = production[i++];

    }

    commonPrefix[j] = '\0';

    if (production[i] == '|') i++; // Skip '|'

    j = 0;

```

```
while (production[i] != '\0' && production[i] != '|') {
```

```
    beta1[j++] = production[i++];
```

```
}
```

```
beta1[j] = '\0';
```

```
if (production[i] == '|') i++; // Skip '|'
```

```
j = 0;
```

```
while (production[i] != '\0') {
```

```
    beta2[j++] = production[i++];
```

```
}
```

```
beta2[j] = '\0';
```

```
eliminateLeftFactoring(nonTerminal, commonPrefix, beta1, beta2);
```

```
return 0;
```

```
}
```

Op:



Lexical Analyzer in C

Online C Compiler - Programiz

programiz.com/c-programming/online-compiler/

Programiz

C Online Compiler

Premium Coding Courses by Programiz

Programiz

PRO

Programiz PRO

main.c

Share

Run

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Function to remove left factoring
5 void eliminateLeftFactoring(char nonTerminal, char commonPrefix[],
6 char beta1[], char beta2[]) {
7     char newNonTerminal = nonTerminal + '1'; // Create new non
8     -terminal (A → A')
9
10    printf("\nAfter Eliminating Left Factoring:\n");
11    printf("%c → %s%c'\n", nonTerminal, commonPrefix, newNonTerminal
12    );
13    printf("%c' → %s | %s'\n", newNonTerminal, beta1, beta2);
14 }
15
16 int main() {
17     char production[50], nonTerminal, commonPrefix[20], beta1[20],
18     beta2[20];
19
20     printf("Enter a left-factored production (e.g., A->abc|abd): ");
21     scanf("%s", production);
22
23     nonTerminal = production[0]; // Extract non-terminal (e.g., A)
24
25     int i = 3, j = 0;
```

Output

Clear

Enter a left-factored production (e.g., A->abc|abd):  
S->aB|cD

After Eliminating Left Factoring:  
S → aBϕ'  
ϕ' → cD |

=== Code Execution Successful ===

Programiz

PRO

Premium Courses by Programiz

Learn More

31°C Sunny

Search

ENG IN

02:17 PM 13-02-2025