Elimination of Left Recursion

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define MAX_PRODUCTIONS 10
#define MAX_LENGTH 10

int findLongestCommonPrefix(char productions[][MAX_LENGTH], int n, char prefix[]) {
    int i = 0;
    while (true) {
        char currentChar = productions[0][i];
        // Stop if we reach the end of the first string
        if (currentChar == '\0') break;
        // Check if all strings have the same character at position 'i'
        for (int j = 1; j < n; j++) {
            if (productions[j][i] != currentChar) {
                prefix[i] = '\0'; // End the prefix string
                return i; // Return the length of the prefix
            }
        }
        prefix[i] = currentChar; // Store the matching character
        i++;
    }
    prefix[i] = '\0'; // End the prefix string
    return i;
}

// Function to eliminate left recursion
void eliminateLeftRecursion(char nonTerminal, char productions[][MAX_LENGTH], int n) {
    char alpha[MAX_PRODUCTIONS][MAX_LENGTH], beta[MAX_PRODUCTIONS][MAX_LENGTH];
    int alphaCount = 0, betaCount = 0;

    // Splitting productions into  (recursive) and  (non-recursive)
    for (int i = 0; i < n; i++) {
        if (productions[i][0] == nonTerminal) {
            strcpy(alpha[alphaCount++], productions[i] + 1); // Store  (Recursive part)
        } else {
            strcpy(beta[betaCount++], productions[i]); // Store  (Non-recursive part)
        }
    }

    // If no left recursion exists, return the original productions
    if (alphaCount == 0) {
        printf("No Left Recursion Detected!\n");
        return;
```

```
    }

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

    printf("\nGrammar after eliminating left recursion:\n");

    // Print A →  A'
    for (int i = 0; i < betaCount; i++) {
        printf("%c -> %s %c'\n", nonTerminal, beta[i], newNonTerminal);
    }

    // Print A' →  A' |
    for (int i = 0; i < alphaCount; i++) {
        printf("%c' -> %s %c'\n", newNonTerminal, alpha[i], newNonTerminal);
    }

    printf("%c' -> \n", newNonTerminal);
}

// Driver function
int main() {
    char nonTerminal;
    int n;
    char productions[MAX_PRODUCTIONS][MAX_LENGTH];

    // User Input
    printf("Enter the non-terminal: ");
    scanf(" %c", &nonTerminal);
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    printf("Enter the productions (e.g., A  or ):\n");
    for (int i = 0; i < n; i++) {
        scanf("%s", productions[i]);
    }

    // Eliminate Left Recursion
    eliminateLeftRecursion(nonTerminal, productions, n);

    return 0;
}
```

Elimination of Left Factoring/Both Left Recursion and Left Factoring

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define MAX_PRODUCTIONS 10
#define MAX_LENGTH 10

int findLongestCommonPrefix(char productions[][MAX_LENGTH], int n, char prefix[]) {
    int i = 0;
    while (true) {
        char currentChar = productions[0][i];
        // Stop if we reach the end of the first string
        if (currentChar == '\0') break;
        // Check if all strings have the same character at position 'i'
        for (int j = 1; j < n; j++) {
            if (productions[j][i] != currentChar) {
                prefix[i] = '\0'; // End the prefix string
                return i; // Return the length of the prefix
            }
        }
        prefix[i] = currentChar; // Store the matching character
        i++;
    }
    prefix[i] = '\0'; // End the prefix string
    return i;
}

// Function to eliminate left recursion
void eliminateLeftRecursion(char nonTerminal, char productions[][MAX_LENGTH], int n) {
    char alpha[MAX_PRODUCTIONS][MAX_LENGTH], beta[MAX_PRODUCTIONS][MAX_LENGTH];
    int alphaCount = 0, betaCount = 0;

    // Splitting productions into  (recursive) and  (non-recursive)
    for (int i = 0; i < n; i++) {
        if (productions[i][0] == nonTerminal) {
            strcpy(alpha[alphaCount++], productions[i] + 1); // Store  (Recursive part)
        } else {
            strcpy(beta[betaCount++], productions[i]); // Store  (Non-recursive part)
        }
    }

    // If no left recursion exists, return the original productions
    if (alphaCount == 0) {
        printf("No Left Recursion Detected!\n");
```

```
        return;
    }

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

    printf("\nGrammar after eliminating left recursion:\n");

    // Print A → βA'
    for (int i = 0; i < betaCount; i++) {
        printf("%c -> %s %c'\n", nonTerminal, beta[i], newNonTerminal);
    }

    // Print A' → αA' | ε
    for (int i = 0; i < alphaCount; i++) {
        printf("%c' -> %s %c'\n", newNonTerminal, alpha[i], newNonTerminal);
    }

    printf("%c' -> ε\n", newNonTerminal);
}

// Function to perform left factoring
void leftFactor(char nonTerminal, char productions[][MAX_LENGTH], int n) {
    char prefix[MAX_LENGTH];
    int prefixLength = findLongestCommonPrefix(productions, n, prefix);

    if (prefixLength == 0) {
        printf("No Left Factoring Needed!\n");
        return;
    }

    char newNonTerminal = nonTerminal + 1; // Generate a new non-terminal

    // Print left-factored productions
    printf("\nGrammar after left factoring:\n");
    printf("%c -> %s%c'\n", nonTerminal, prefix, newNonTerminal);

    // Print new productions for the new non-terminal
    printf("%c' -> ", newNonTerminal);
    for (int i = 0; i < n; i++) {
        if (strncmp(productions[i], prefix, prefixLength) == 0) {
            printf("%s ", productions[i] + prefixLength);
        } else {
            printf("%s ", productions[i]);
        }
    }
```

```
    printf("\n");
}

// Driver function
int main() {
    char nonTerminal;
    int n;
    char productions[MAX_PRODUCTIONS][MAX_LENGTH];

    // User Input
    printf("Enter the non-terminal: ");
    scanf(" %c", &nonTerminal);
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    printf("Enter the productions (e.g., A  or  ):\n");
    for (int i = 0; i < n; i++) {
        scanf("%s", productions[i]);
    }

    // Step 1: Eliminate Left Recursion
    eliminateLeftRecursion(nonTerminal, productions, n);

    // Step 2: Perform Left Factoring
    leftFactor(nonTerminal, productions, n);

    return 0;
}
```