**Lab Report: Eliminating Left Recursion and Left Factoring in a Grammar**

**Objective**

The objective of this lab is to understand and implement programs to:

1. Eliminate left recursion in a grammar.
2. Perform left factoring on a grammar to make it suitable for top-down parsing.

**Problem Statements**

1. **Problem 1:**  
   Write a program to eliminate the left recursion from the given grammar using C programming.
2. **Problem 2:**  
   Write a program to perform left factoring on the given grammar to remove ambiguities in parsing.

**Program Code**

**Program 1: Removing Left Recursion**

#include <stdio.h>

#include <string.h>

#define MAX\_RULES 10

#define MAX\_LENGTH 100

#define MAX\_NON\_TERMINALS 10

typedef struct {

char non\_terminal;

char alpha[MAX\_RULES][MAX\_LENGTH];

char beta[MAX\_RULES][MAX\_LENGTH];

int alpha\_count;

int beta\_count;

} GrammarRule;

void remove\_left\_recursion(GrammarRule\* rule) {

if (rule->alpha\_count == 0) {

printf("%c -> ", rule->non\_terminal);

for (int i = 0; i < rule->beta\_count; i++) {

printf("%s", rule->beta[i]);

if (i < rule->beta\_count - 1) printf(" | ");

}

printf("\n");

return;

}

char new\_non\_terminal = rule->non\_terminal + '\'';

printf("%c -> ", rule->non\_terminal);

for (int i = 0; i < rule->beta\_count; i++) {

printf("%s%c", rule->beta[i], new\_non\_terminal);

if (i < rule->beta\_count - 1) printf(" | ");

}

printf("\n");

printf("%c -> ", new\_non\_terminal);

for (int i = 0; i < rule->alpha\_count; i++) {

printf("%s%c", rule->alpha[i], new\_non\_terminal);

if (i < rule->alpha\_count - 1) printf(" | ");

}

printf(" | epsilon\n");

}

int main() {

GrammarRule rules[MAX\_NON\_TERMINALS];

int num\_non\_terminals;

printf("Enter the number of non-terminals: ");

scanf("%d", &num\_non\_terminals);

for (int r = 0; r < num\_non\_terminals; r++) {

GrammarRule\* rule = &rules[r];

rule->alpha\_count = 0;

rule->beta\_count = 0;

printf("\nEnter the non-terminal %d (single character): ", r + 1);

scanf(" %c", &rule->non\_terminal);

int n;

printf("Enter the number of productions for %c: ", rule->non\_terminal);

scanf("%d", &n);

printf("Enter the productions for %c (one per line):\n", rule->non\_terminal);

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

char production[MAX\_LENGTH];

scanf("%s", production);

if (production[0] == rule->non\_terminal) {

strcpy(rule->alpha[rule->alpha\_count++], production + 1);

} else {

strcpy(rule->beta[rule->beta\_count++], production);

}

}

}

printf("\nGrammar without left recursion:\n");

for (int r = 0; r < num\_non\_terminals; r++) {

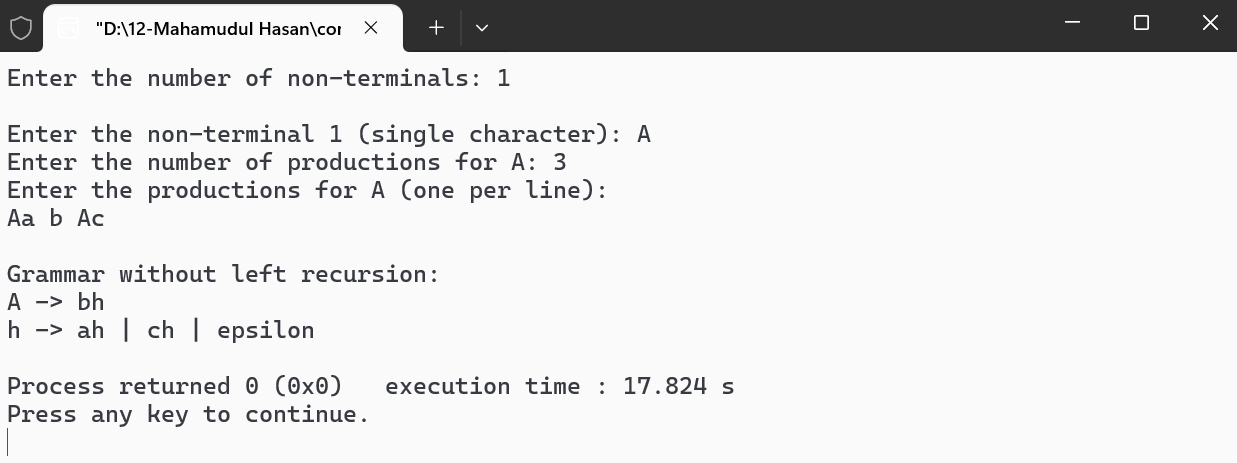
remove\_left\_recursion(&rules[r]);

}

return 0;

}

**Sample Input & Output:**



**Program 2: Performing Left Factoring**

#include <stdio.h>

#include <string.h>

#define MAX\_RULES 10

#define MAX\_LENGTH 100

#define MAX\_NON\_TERMINALS 10

typedef struct {

char non\_terminal;

char productions[MAX\_RULES][MAX\_LENGTH];

int production\_count;

} GrammarRule;

int longest\_common\_prefix(char \*str1, char \*str2) {

int i = 0;

while (str1[i] && str2[i] && str1[i] == str2[i]) {

i++;

}

return i;

}

void left\_factoring(GrammarRule\* rule) {

int prefix\_length;

int common\_prefix[MAX\_RULES] = {0};

for (int i = 0; i < rule->production\_count; i++) {

if (common\_prefix[i]) continue;

for (int j = i + 1; j < rule->production\_count; j++) {

prefix\_length = longest\_common\_prefix(rule->productions[i], rule->productions[j]);

if (prefix\_length > 0) {

char prefix[MAX\_LENGTH];

strncpy(prefix, rule->productions[i], prefix\_length);

prefix[prefix\_length] = '\0';

printf("%c -> %sH\n", rule->non\_terminal, prefix);

printf("H -> ");

int first = 1;

for (int k = i; k < rule->production\_count; k++) {

if (longest\_common\_prefix(rule->productions[i], rule->productions[k]) == prefix\_length) {

if (!first) printf(" | ");

printf("%s", rule->productions[k] + prefix\_length);

common\_prefix[k] = 1;

first = 0;

}

}

printf(" | epsilon\n");

break;

}

}

}

for (int i = 0; i < rule->production\_count; i++) {

if (!common\_prefix[i]) {

printf("%c -> %s\n", rule->non\_terminal, rule->productions[i]);

}

}

}

int main() {

GrammarRule rules[MAX\_NON\_TERMINALS];

int num\_non\_terminals;

printf("Enter the number of non-terminals: ");

scanf("%d", &num\_non\_terminals);

for (int r = 0; r < num\_non\_terminals; r++) {

GrammarRule\* rule = &rules[r];

rule->production\_count = 0;

printf("\nEnter the non-terminal %d (single character): ", r + 1);

scanf(" %c", &rule->non\_terminal);

int n;

printf("Enter the number of productions for %c: ", rule->non\_terminal);

scanf("%d", &n);

printf("Enter the productions for %c (one per line):\n", rule->non\_terminal);

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

scanf("%s", rule->productions[rule->production\_count++]);

}

}

printf("\nGrammar after left factoring:\n");

for (int r = 0; r < num\_non\_terminals; r++) {

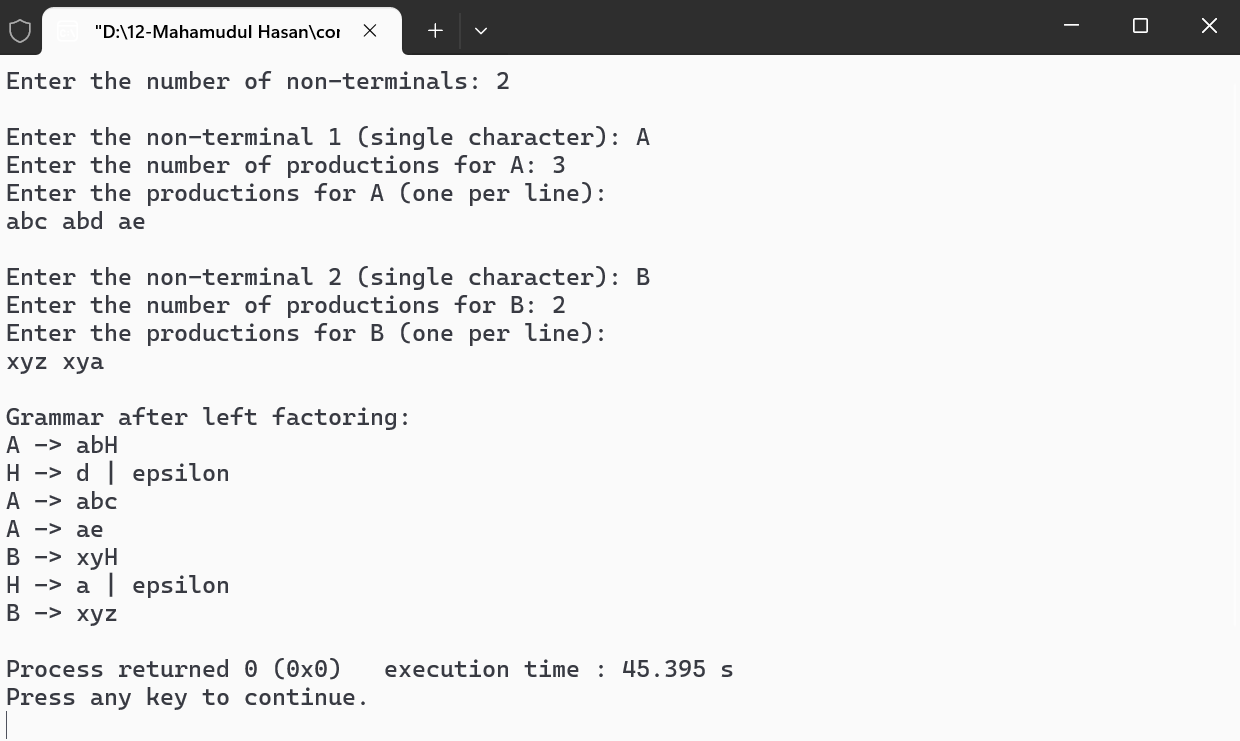
left\_factoring(&rules[r]);

}

return 0;

}

**Sample Input and Output**

**Conclusion**

The programs successfully eliminate left recursion and perform left factoring, enabling top-down parsing for grammars. The structured approach to modifying the grammar ensures compatibility with LL(1) parsers.