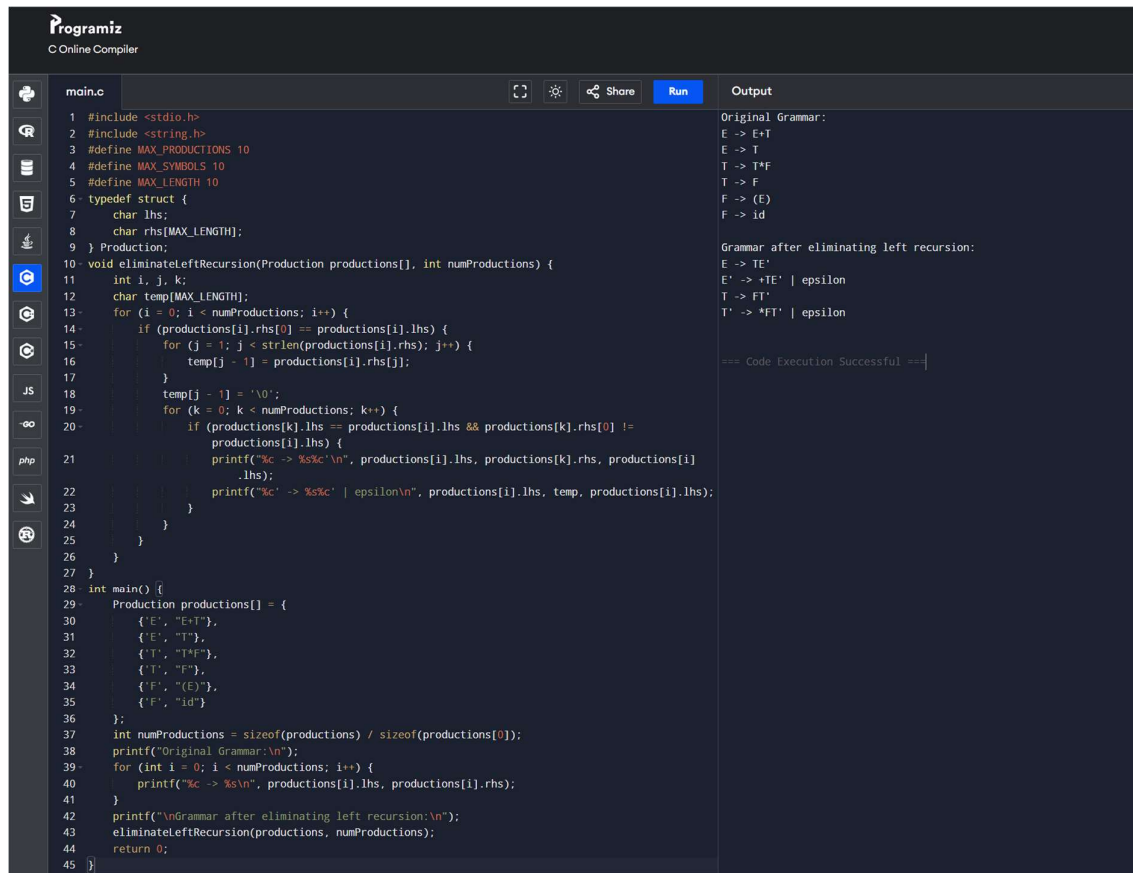


# Compiler Design (10/02/20250)

## 6. Implement a C program to eliminate left recursion.



The screenshot shows the Programiz C Online Compiler interface. The editor contains a C program that implements an algorithm to eliminate left recursion from a grammar. The program defines a structure for productions and a function `eliminateLeftRecursion` that iterates through the productions, identifying left-recursive rules and transforming them into equivalent non-left-recursive forms using the standard algorithm (introducing a new non-terminal and epsilon transitions). The `main` function initializes a grammar with the following productions:

- `E -> E+T`
- `E -> T`
- `T -> T*F`
- `T -> F`
- `F -> (E)`
- `F -> id`

The program then prints the original grammar and the grammar after eliminating left recursion. The output shows the transformed grammar:

- `E -> TE'`
- `E' -> +TE' | epsilon`
- `T -> FT'`
- `T' -> *FT' | epsilon`

```
1 #include <stdio.h>
2 #include <string.h>
3 #define MAX_PRODUCTIONS 10
4 #define MAX_SYMBOLS 10
5 #define MAX_LENGTH 10
6 typedef struct {
7     char lhs;
8     char rhs[MAX_LENGTH];
9 } Production;
10 void eliminateLeftRecursion(Production productions[], int numProductions) {
11     int i, j, k;
12     char temp[MAX_LENGTH];
13     for (i = 0; i < numProductions; i++) {
14         if (productions[i].rhs[0] == productions[i].lhs) {
15             for (j = 1; j < strlen(productions[i].rhs); j++) {
16                 temp[j - 1] = productions[i].rhs[j];
17             }
18             temp[j - 1] = '\0';
19             for (k = 0; k < numProductions; k++) {
20                 if (productions[k].lhs == productions[i].lhs && productions[k].rhs[0] !=
21                     productions[i].lhs) {
22                     printf("%c -> %s%c\n", productions[i].lhs, productions[k].rhs, productions[i]
23                         .lhs);
24                     printf("%c' -> %s%c' | epsilon\n", productions[i].lhs, temp, productions[i].lhs);
25                 }
26             }
27         }
28     }
29     int main() {
30         Production productions[] = {
31             {'E', "E+T"},
32             {'E', "T"},
33             {'T', "T*F"},
34             {'T', "F"},
35             {'F', "(E)"},
36             {'F', "id"}
37         };
38         int numProductions = sizeof(productions) / sizeof(productions[0]);
39         printf("Original Grammar:\n");
40         for (int i = 0; i < numProductions; i++) {
41             printf("%c -> %s\n", productions[i].lhs, productions[i].rhs);
42         }
43         printf("\nGrammar after eliminating left recursion:\n");
44         eliminateLeftRecursion(productions, numProductions);
45         return 0;
46     }
```

## 7. Implement a C program to eliminate left factoring.

Programiz  
C Online Compiler

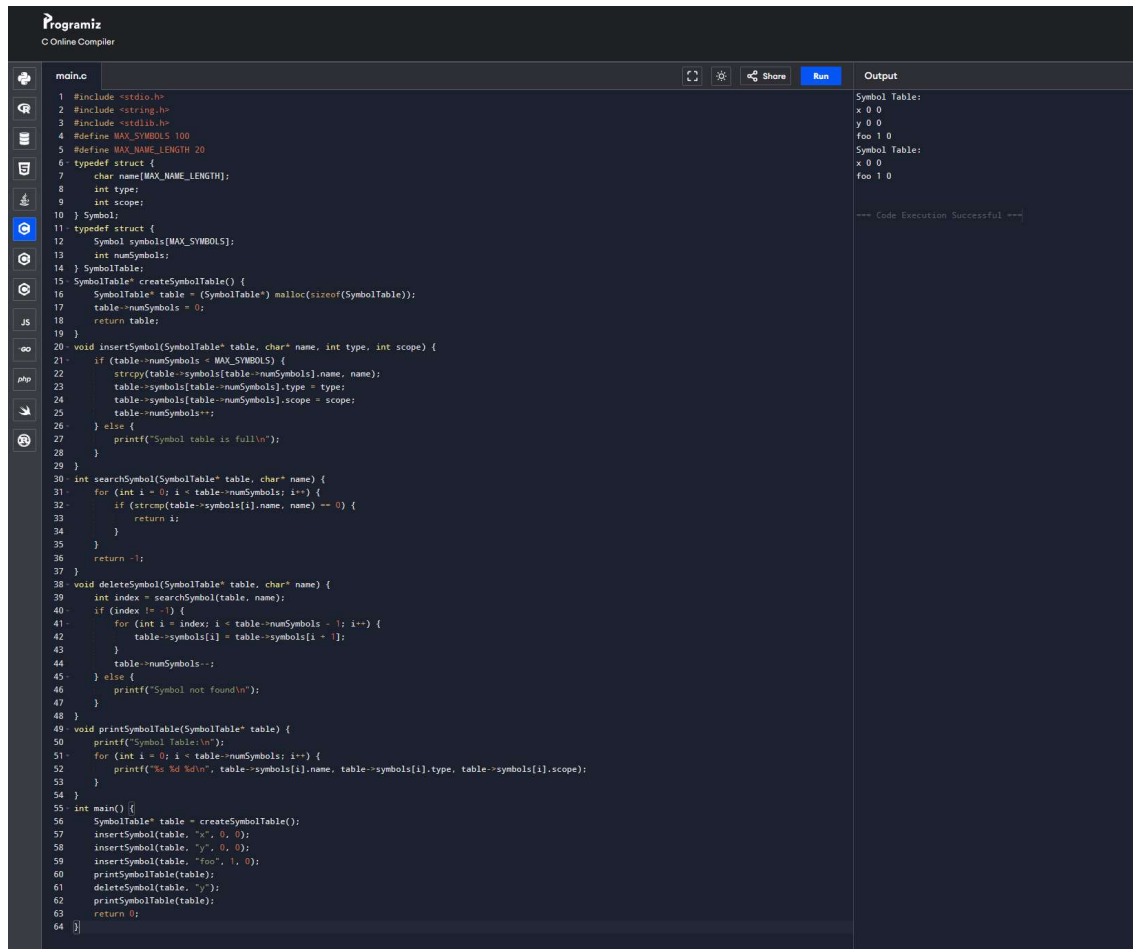
main.c

1 #include <stdio.h>  
2 #include <string.h>  
3 #define MAX\_PRODUCTIONS 10  
4 #define MAX\_SYMBOLS 10  
5 #define MAX\_LENGTH 10  
6 typedef struct {  
7 char lhs;  
8 char rhs[MAX\_LENGTH];  
9 } Production;  
10 void eliminateLeftFactoring(Production productions[], int numProductions) {  
11 int i, j, k;  
12 char temp[MAX\_LENGTH];  
13 char newProduction[MAX\_LENGTH];  
14 for (i = 0; i < numProductions; i++) {  
15 for (j = i + 1; j < numProductions; j++) {  
16 if (productions[j].lhs == productions[i].lhs) {  
17 int prefixLength = 0;  
18 while (prefixLength < strlen(productions[i].rhs) && prefixLength < strlen(productions[j].rhs) &&  
19 productions[i].rhs[prefixLength] == productions[j].rhs[prefixLength]) {  
20 prefixLength++;  
21 }  
22 if (prefixLength > 0) {  
23 newProduction[0] = productions[i].lhs;  
24 newProduction[1] = '\\';  
25 newProduction[2] = '\\0';  
26 printf("%c -> %s\\n", productions[i].lhs, productions[i].rhs, productions[i].lhs);  
27 for (k = 0; k < numProductions; k++) {  
28 if (productions[k].lhs == productions[i].lhs) {  
29 strcpy(temp, productions[k].rhs + prefixLength);  
30 printf("%c' -> %s | ", newProduction[0], temp);  
31 }  
32 }  
33 printf("epsilon\\n");  
34 }  
35 }  
36 }  
37 }  
38 }  
39 int main() {  
40 Production productions[] = {  
41 {'E', "TX"},  
42 {'E', "TY"}  
43 };  
44 int numProductions = sizeof(productions) / sizeof(productions[0]);  
45 printf("Original Grammar:\\n");  
46 for (int i = 0; i < numProductions; i++) {  
47 printf("%c -> %s\\n", productions[i].lhs, productions[i].rhs);  
48 }  
49 printf("\\nGrammar after eliminating left factoring:\\n");  
50 eliminateLeftFactoring(productions, numProductions);  
51 return 0;  
52 }

Output

Original Grammar:  
E -> TX  
E -> TY  
  
Grammar after eliminating left factoring:  
E -> TXE'  
E' -> X | E' -> Y | epsilon  
  
=== Code Execution Successful ===

## 8. Implement a C program to perform symbol table operations.



The screenshot displays the Programiz C Online Compiler interface. The main editor shows a C program for implementing a symbol table. The program includes headers for `stdio.h`, `string.h`, and `stdlib.h`. It defines `MAX_SYMBOLS` as 100 and `MAX_NAME_LENGTH` as 20. A `Symbol` struct is defined with `char name`, `int type`, and `int scope`. A `SymbolTable` struct is defined with an array of `Symbol` and a `numSymbols` counter. The `createSymbolTable` function allocates memory for the symbol table. The `insertSymbol` function adds a new symbol to the table, checking for full capacity. The `searchSymbol` function finds a symbol by name. The `deleteSymbol` function removes a symbol from the table. The `printSymbolTable` function displays the current state of the symbol table. The `main` function demonstrates the operations: creating the table, inserting symbols 'x', 'y', and 'foo', printing the table, deleting 'y', and printing the table again.

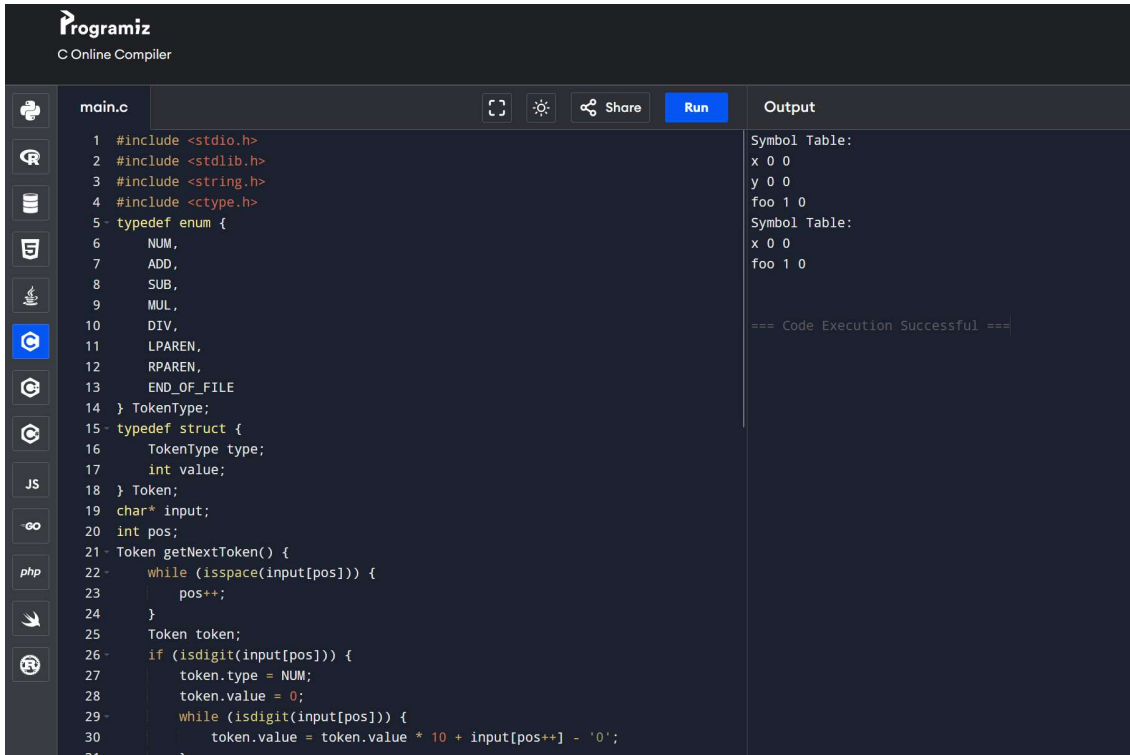
```
main.c
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #define MAX_SYMBOLS 100
5 #define MAX_NAME_LENGTH 20
6 typedef struct {
7     char name[MAX_NAME_LENGTH];
8     int type;
9     int scope;
10 } Symbol;
11 typedef struct {
12     Symbol symbols[MAX_SYMBOLS];
13     int numSymbols;
14 } SymbolTable;
15 SymbolTable* createSymbolTable() {
16     SymbolTable* table = (SymbolTable*) malloc(sizeof(SymbolTable));
17     table->numSymbols = 0;
18     return table;
19 }
20 void insertSymbol(SymbolTable* table, char* name, int type, int scope) {
21     if (table->numSymbols < MAX_SYMBOLS) {
22         strcpy(table->symbols[table->numSymbols].name, name);
23         table->symbols[table->numSymbols].type = type;
24         table->symbols[table->numSymbols].scope = scope;
25         table->numSymbols++;
26     } else {
27         printf("Symbol table is full\n");
28     }
29 }
30 int searchSymbol(SymbolTable* table, char* name) {
31     for (int i = 0; i < table->numSymbols; i++) {
32         if (strcmp(table->symbols[i].name, name) == 0) {
33             return i;
34         }
35     }
36     return -1;
37 }
38 void deleteSymbol(SymbolTable* table, char* name) {
39     int index = searchSymbol(table, name);
40     if (index != -1) {
41         for (int i = index; i < table->numSymbols - 1; i++) {
42             table->symbols[i] = table->symbols[i + 1];
43         }
44         table->numSymbols--;
45     } else {
46         printf("Symbol not found\n");
47     }
48 }
49 void printSymbolTable(SymbolTable* table) {
50     printf("Symbol Table:\n");
51     for (int i = 0; i < table->numSymbols; i++) {
52         printf("%s %d %d\n", table->symbols[i].name, table->symbols[i].type, table->symbols[i].scope);
53     }
54 }
55 int main() {
56     SymbolTable* table = createSymbolTable();
57     insertSymbol(table, "x", 0, 0);
58     insertSymbol(table, "y", 0, 0);
59     insertSymbol(table, "foo", 1, 0);
60     printSymbolTable(table);
61     deleteSymbol(table, "y");
62     printSymbolTable(table);
63     return 0;
64 }
```

Output

```
Symbol Table:
x 0 0
y 0 0
foo 1 0
Symbol Table:
x 0 0
foo 1 0

--- Code Execution Successful ---
```

## 9. Write a C program to construct recursive descent parsing.



The screenshot shows the Programiz C Online Compiler interface. The editor contains a C program named `main.c` that defines a token type enum and a token struct, and implements a `getNextToken()` function. The output panel shows the symbol table and a successful execution message.




```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <ctype.h>
5 typedef enum {
6     NUM,
7     ADD,
8     SUB,
9     MUL,
10    DIV,
11    LPAREN,
12    RPAREN,
13    END_OF_FILE
14 } TokenType;
15 typedef struct {
16     TokenType type;
17     int value;
18 } Token;
19 char* input;
20 int pos;
21 Token getNextToken() {
22     while (isspace(input[pos])) {
23         pos++;
24     }
25     Token token;
26     if (isdigit(input[pos])) {
27         token.type = NUM;
28         token.value = 0;
29         while (isdigit(input[pos])) {
30             token.value = token.value * 10 + input[pos++] - '0';
31         }
```

Symbol Table:  
x 0 0  
y 0 0  
foo 1 0  
Symbol Table:  
x 0 0  
foo 1 0

=== Code Execution Successful ===

### 13. Write a C program for implementing a Lexical Analyzer to Count the number of characters, words, and lines .

Programiz  
C Online Compiler



Run

Output

```
1 #include <stdio.h>
2 #include <ctype.h>
3 void countStatistics(FILE *file) {
4     int charCount = 0;
5     int wordCount = 0;
6     int lineCount = 0;
7     int inWord = 0;
8     char c;
9     while ((c = fgetc(file)) != EOF) {
10         charCount++;
11         if (c == '\n') {
12             lineCount++;
13         }
14         if (isspace(c)) {
15             inWord = 0;
16         } else if (!inWord) {
17             wordCount++;
18             inWord = 1;
19         }
20     }
21     printf("Character Count: %d\n", charCount);
22     printf("Word Count: %d\n", wordCount);
23     printf("Line Count: %d\n", lineCount);
24 }
25 int main() {
26     FILE *file;
27     char filename[100];
28     printf("Enter the filename: ");
29     scanf("%s", filename);
30     file = fopen(filename, "r");
31     if (file == NULL) {
32         printf("Error opening file\n");
33         return 1;
34     }
35     countStatistics(file);
36     fclose(file);
37     return 0;
38 }
```

Symbol Table:  
x 0 0  
y 0 0  
foo 1 0  
Symbol Table:  
x 0 0  
foo 1 0

=== Code Execution Successful ===

## 14. Write a C Program for code optimization to eliminate common subexpression.

Programiz  
C Online Compiler

```
main.c
1 #include <stdio.h>
2 int originalFunction(int a, int b, int c) {
3     int x = a + b;
4     int y = a + b + c;
5     int z = a + b * 2;
6     return x + y + z;
7 }
8 int optimizedFunction(int a, int b, int c) {
9     int x = a + b;
10    int y = x + c;
11    int z = x * 2;
12    return x + y + z;
13 }
14 int main() {
15     int a = 2, b = 3, c = 4;
16     printf("Original function result: %d\n", originalFunction(a, b, c));
17     printf("Optimized function result: %d\n", optimizedFunction(a, b, c));
18     return 0;
19 }
```

Output

Symbol Table:  
x 0 0  
y 0 0  
foo 1 0  
Symbol Table:  
x 0 0  
foo 1 0

=== Code Execution Successful ===