**C PROGRAMS COMPILER DESIGN**

**1. Lexical Analyzer for Identifiers, Constants, and Operators**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX\_ID\_LEN 31

int isOperator(char ch) {

char operators[] = "+-\*/%=<>&|^!";

for (int i = 0; i < strlen(operators); i++) {

if (ch == operators[i]) {

return 1;

}

}

return 0;

}

int isKeyword(const char \*str) {

char \*keywords[] = {"int", "float", "char", "if", "else", "while", "for", "return", "void"};

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

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

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

return 1;

}

}

return 0;

}

void lexicalAnalyzer(const char \*code) {

int i = 0;

char token[MAX\_ID\_LEN];

int tokenIndex = 0;

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

if (isspace(code[i])) {

i++; // Ignore spaces, tabs, and new lines

continue;

}

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

tokenIndex = 0;

while (isalnum(code[i]) || code[i] == '\_') {

if (tokenIndex < MAX\_ID\_LEN - 1)

token[tokenIndex++] = code[i];

i++;

}

token[tokenIndex] = '\0';

if (isKeyword(token)) {

printf("Keyword: %s\n", token);

} else {

printf("Identifier: %s\n", token);

}

}

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

tokenIndex = 0;

while (isdigit(code[i])) {

token[tokenIndex++] = code[i];

i++;

}

token[tokenIndex] = '\0';

printf("Constant: %s\n", token);

}

else if (isOperator(code[i])) {

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

i++;

}

else if (code[i] == '/' && code[i + 1] == '/') {

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

i++;

}

}

else if (code[i] == '/' && code[i + 1] == '\*') {

i += 2;

while (code[i] != '\0' && !(code[i] == '\*' && code[i + 1] == '/')) {

i++;

}

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

i += 2;

}

}

else {

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

i++;

}

}

}

int main() {

char code[] = "int a = 5; \na = a + 10;\n

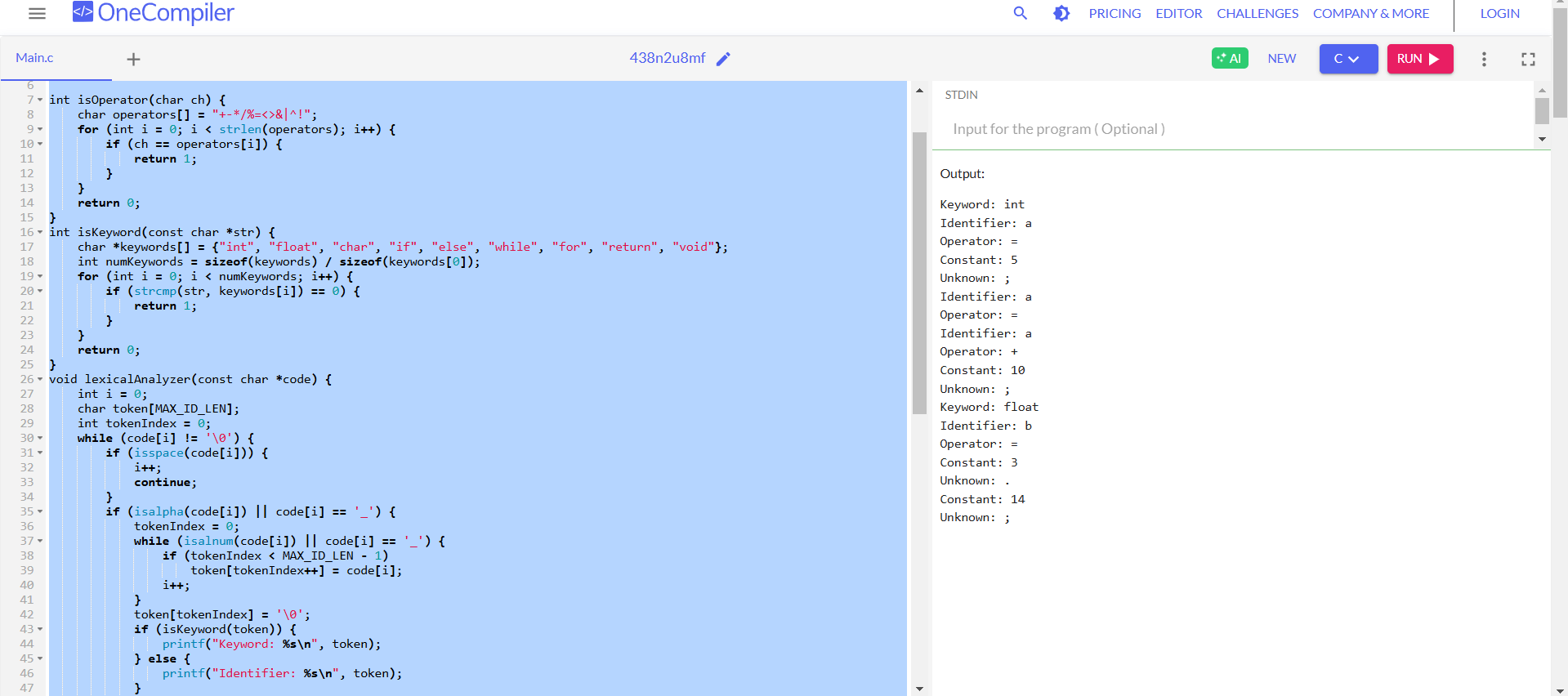
float b = 3.14;\n";

lexicalAnalyzer(code);

return 0;

}

**OUTPUT:**



**2. Lexical Analyzer for Comments**

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

B

ool is\_single\_line\_comment(const char \*line) {

return strncmp(line, "//", 2) == 0;

}

bool is\_multi\_line\_comment(const char \*line) {

return strncmp(line, "/\*", 2) == 0 && strstr(line, "\*/") != NULL;

}

const char\* check\_comment(const char \*line) {

if (is\_single\_line\_comment(line)) {

return "Single-line comment";

} else if (is\_multi\_line\_comment(line)) {

return "Multi-line comment";

} else {

return "Not a comment";

}

}

int main() {

const char \*test\_lines[] = {

"// This is a single-line comment",

"/\* This is a multi-line comment \*/",

"int a = 10; // Variable declaration",

"printf(\"Hello World\");",

"/\* Multi-line\n comment spanning\n multiple lines \*/"

};

int num\_tests = sizeof(test\_lines) / sizeof(test\_lines[0]);

for (int i = 0; i < num\_tests; i++) {

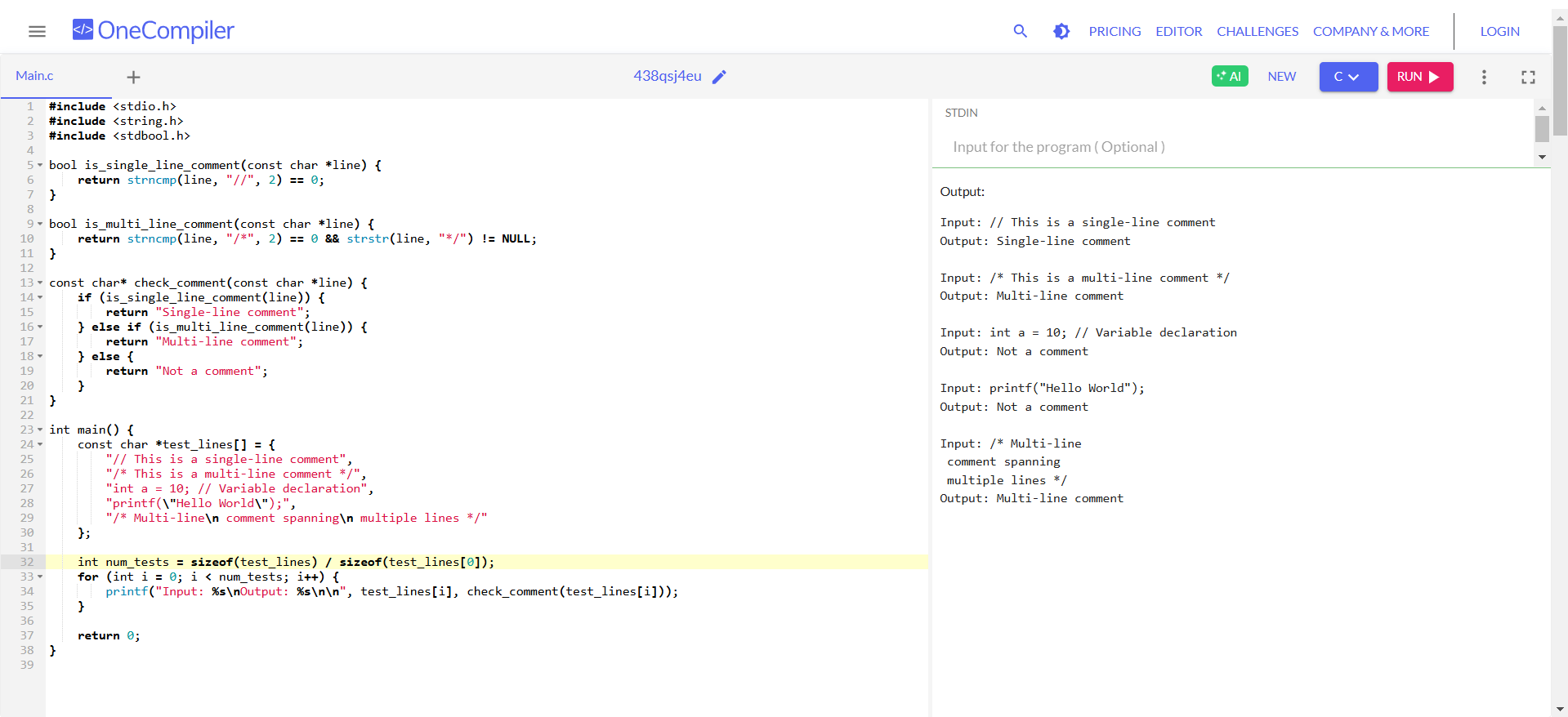
printf("Input: %s\nOutput: %s\n\n", test\_lines[i], check\_comment(test\_lines[i]));

}

return 0;

}

**OUTPUT:**



**3. Lexical Analyzer for Arithmetic Operators**

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

bool is\_operator(const char \*token) {

return (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||

strcmp(token, "\*") == 0 || strcmp(token, "/") == 0);

}

const char\* check\_operator(const char \*token) {

if (is\_operator(token)) {

return "Valid operator";

} else {

return "Not an operator";

}

}

int main() {

const char \*test\_tokens[] = {

"+", "-", "\*", "/", "%", "&", "x", "++"

};

int num\_tests = sizeof(test\_tokens) / sizeof(test\_tokens[0]);

for (int i = 0; i < num\_tests; i++) {

printf("Input: %s\nOutput: %s\n\n", test\_tokens[i], check\_operator(test\_tokens[i]));

}

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**4. Lexical Analyzer for Whitespace and Newlines**

#include <stdio.h>

void countSpaces(char str[]) {

int i = 0, spaces = 0, newlines = 0;

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

if (str[i] == ' ') spaces++;

if (str[i] == '\n') newlines++;

i++;

}

printf("Spaces: %d, Newlines: %d\n", spaces, newlines);

}

int main() {

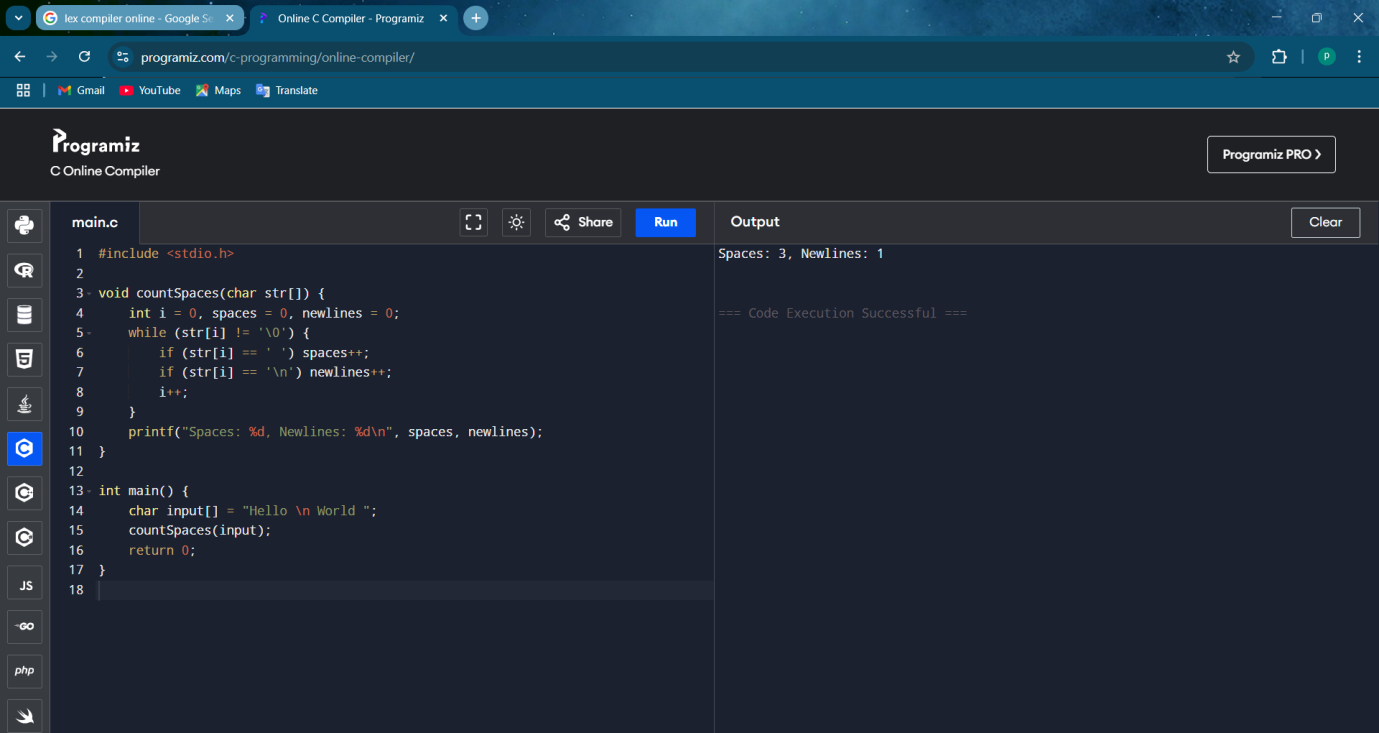
char input[] = "Hello \n World ";

countSpaces(input);

return 0;

}

**OUTPUT:**



**5. Lexical Analyzer for Valid Identifiers**

c

CopyEdit

#include <stdio.h>

#include <ctype.h>

void checkIdentifier(char str[]) {

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

printf("Invalid Identifier\n");

return;

}

int i = 1;

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

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

printf("Invalid Identifier\n");

return;

}

i++;

}

printf("Valid Identifier\n");

}

int main() {

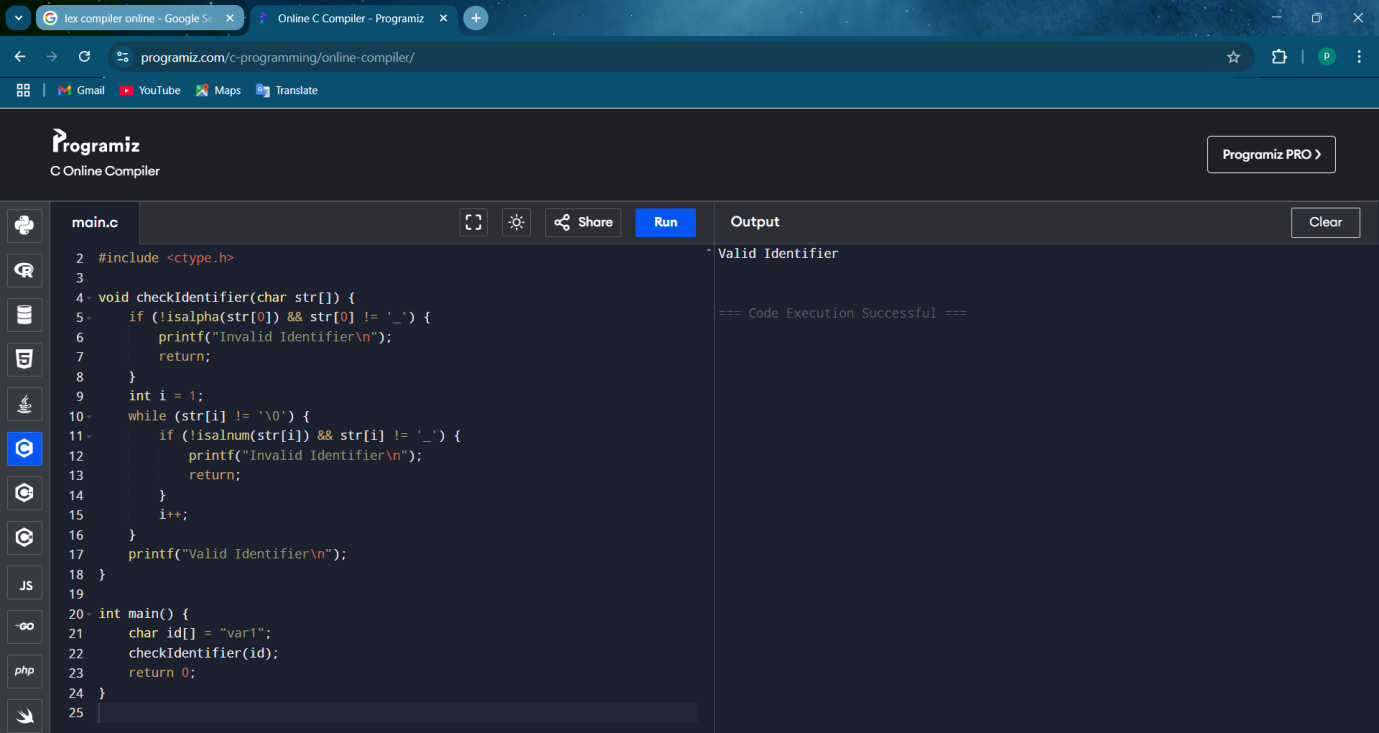
char id[] = "var1";

checkIdentifier(id);

return 0;

}

**OUTPUT:**



**6. Eliminate Left Recursion**

c

CopyEdit

#include <stdio.h>

#include <string.h>

void eliminateLeftRecursion(char nonTerminal, char alpha[], char beta[]) {

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

printf("%c -> %s%c'\n", nonTerminal, beta, nonTerminal);

printf("%c' -> %s%c' | ε\n", nonTerminal, alpha, nonTerminal);

}

int main() {

char nonTerminal = 'A';

char alpha[] = "x"; // Recursive part

char beta[] = "y"; // Non-recursive part

printf("Given Grammar:\n");

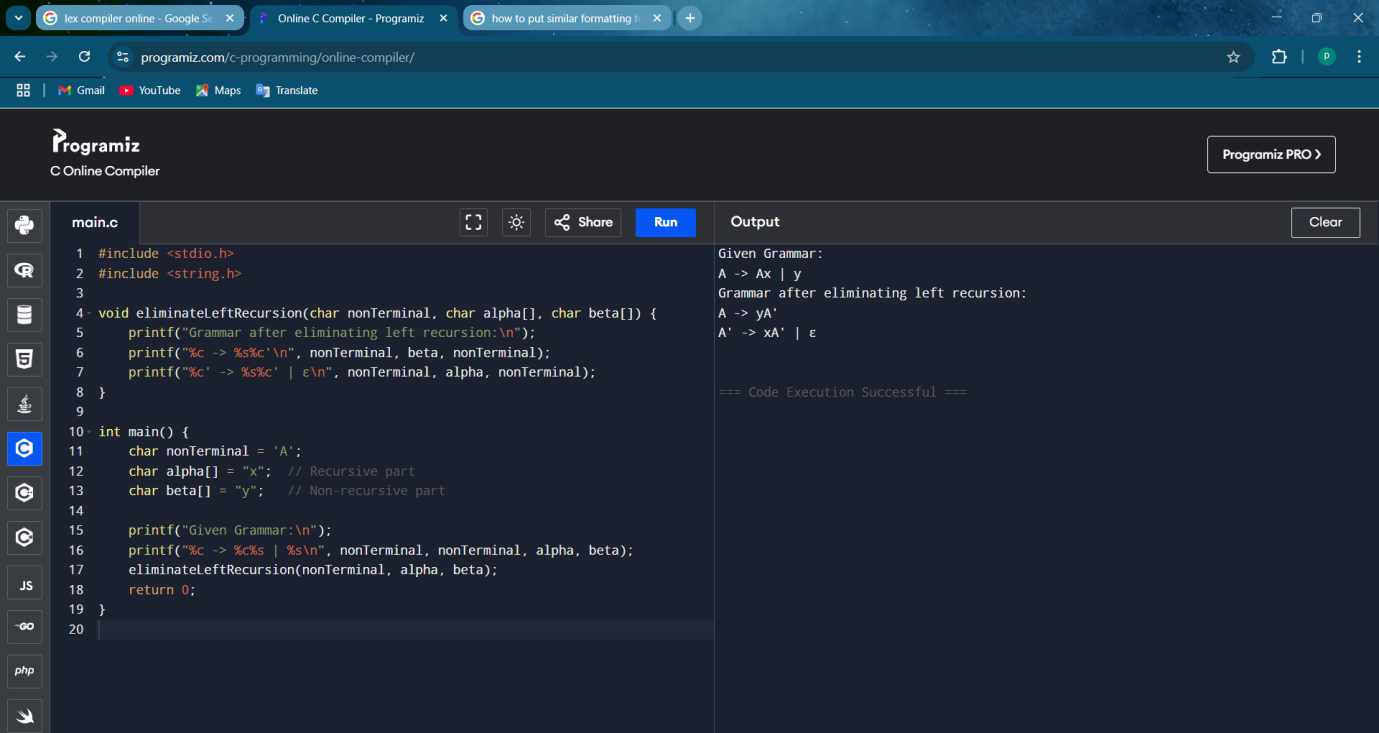
printf("%c -> %c%s | %s\n", nonTerminal, nonTerminal, alpha, beta);

eliminateLeftRecursion(nonTerminal, alpha, beta);

return 0;

}

**OUTPUT:**



**7. Eliminate Left Factoring**

c

CopyEdit

#include <stdio.h>

#include <string.h>

void eliminateLeftFactoring(char nonTerminal, char common[], char diff1[], char diff2[]) {

printf("Grammar after eliminating left factoring:\n");

printf("%c -> %s%c'\n", nonTerminal, common, nonTerminal);

printf("%c' -> %s | %s\n", nonTerminal, diff1, diff2);

}

int main() {

char nonTerminal = 'A';

char common[] = "x"; // Common part

char diff1[] = "y"; // First alternative

char diff2[] = "z"; // Second alternative

printf("Given Grammar:\n");

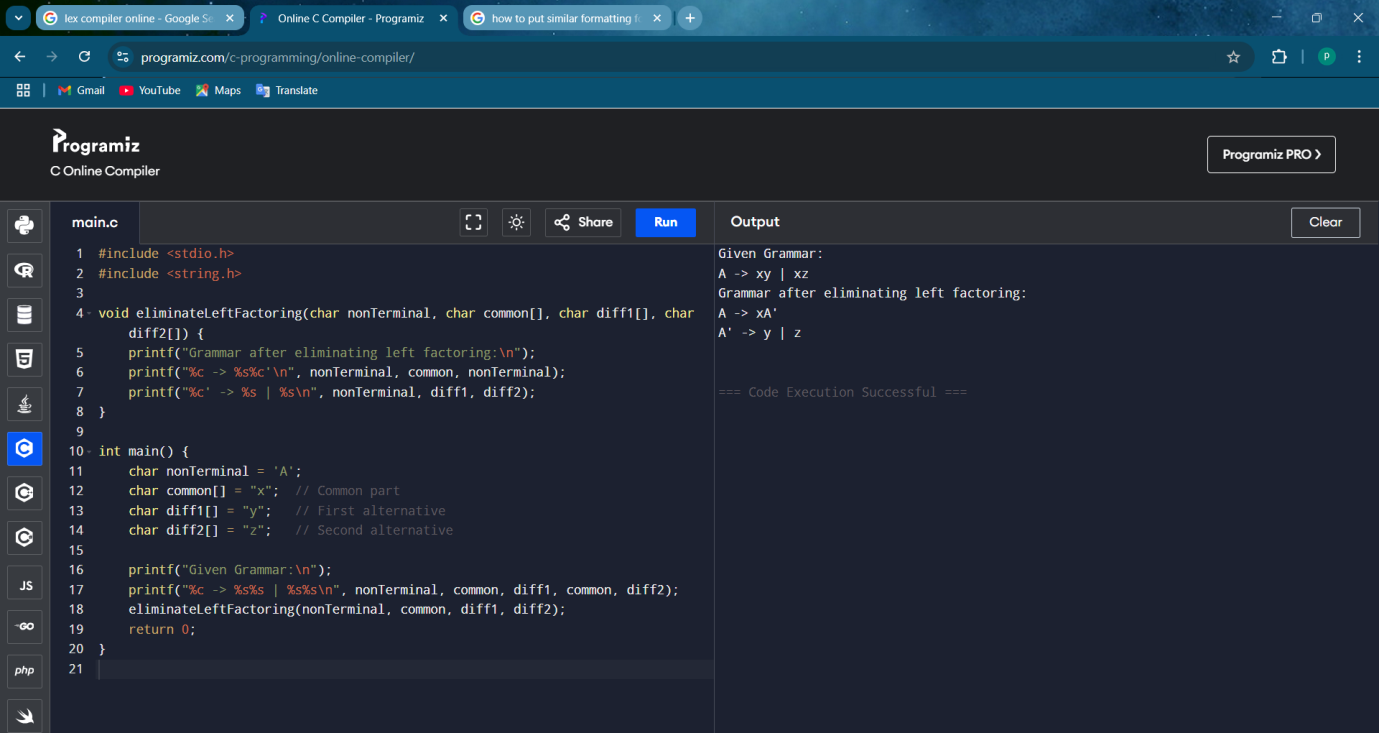
printf("%c -> %s%s | %s%s\n", nonTerminal, common, diff1, common, diff2);

eliminateLeftFactoring(nonTerminal, common, diff1, diff2);

return 0;

}

**OUTPUT:**



**8. Symbol Table Implementation**

#include <stdio.h> #include <string.h>

#define MAX 100 struct SymbolTable {

char identifier[50]; char type[20];

int address;

} table[MAX];

int count = 0;

void insert(char \*id, char \*type, int addr) { strcpy(table[count].identifier, id); strcpy(table[count].type, type); table[count].address = addr;

count++;

printf("Inserted: %s, Type: %s, Address: %d\n", id, type, addr);

}

void search(char \*id) {

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

if (strcmp(table[i].identifier, id) == 0) {

printf("Found: %s, Type: %s, Address: %d\n", table[i].identifier, table[i].type, table[i].address); return;

}

}

printf("Identifier not found.\n");

}

void display() { printf("\nSymbol Table:\n");

printf("Identifier\tType\tAddress\n"); for (int i = 0; i < count; i++) {

printf("%s\t\t%s\t%d\n", table[i].identifier, table[i].type, table[i].address);

}

}

int main() { int choice;

char id[50], type[20];

int address = 1000; // Initial memory address

while (1) {

printf("\n1. Insert\n2. Search\n3. Display\n4. Exit\nEnter choice: "); scanf("%d", &choice);

switch (choice) { case 1:

printf("Enter identifier and type: "); scanf("%s %s", id, type);

insert(id, type, address++); break;

case 2:

printf("Enter identifier to search: "); scanf("%s", id);

search(id); break;

case 3:

display(); break;

case 4:

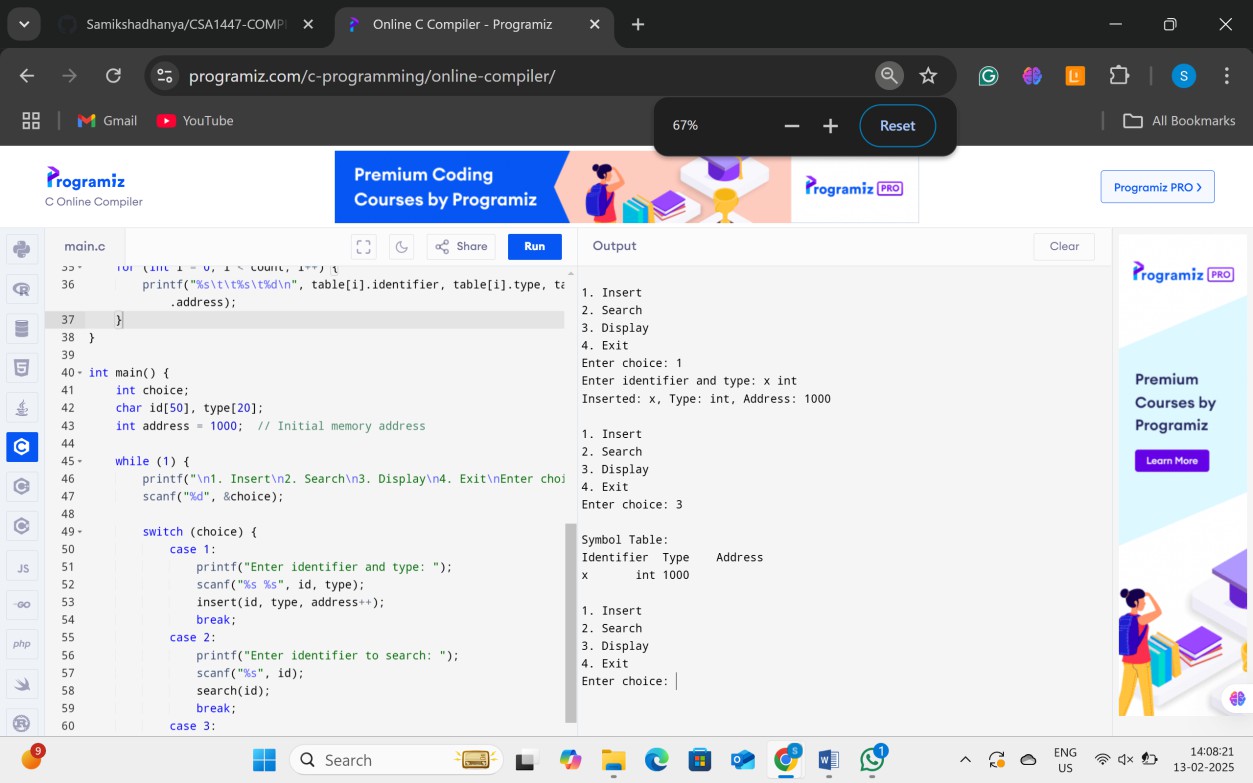
return 0;

}

}

}

**OUTPUT:**



**9. Grammar Checking#include <stdio.h> #include <string.h>**

int checkGrammar(char \*str, int left, int right) {

if (left > right) return 1; // Empty string is valid if (str[left] == 'a' && str[right] == 'b')

return checkGrammar(str, left + 1, right - 1);

return 0;

}

int main() {

char input[50]; printf("Enter a string: "); scanf("%s", input);

if (checkGrammar(input, 0, strlen(input) - 1)) printf("Valid according to the grammar.\n");

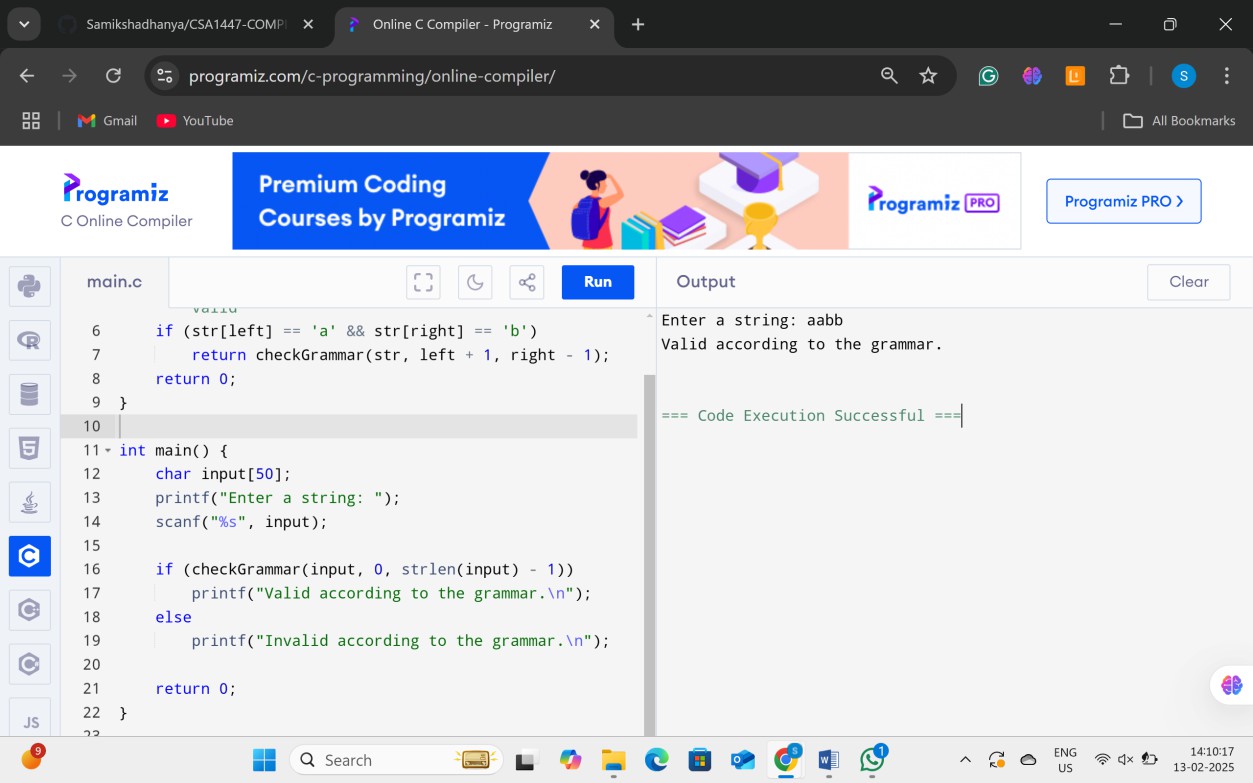
else

printf("Invalid according to the grammar.\n");

return 0;

}

**OUTPUT:**



**10. Recursive Descent Parser**

#include <stdio.h> #include <stdlib.h> #include <string.h>

char input[100]; // Input string

int pos = 0; // Pointer to track parsing position

void E(); // Expression void EPrime();

void T(); // Term void TPrime(); void F(); // Factor

// Function to handle parsing errors void error() {

printf("Error in parsing!\n"); exit(0);

}

// Function to match a character and move to the next void match(char expected) {

if (input[pos] == expected) pos++;

else

error();

}

// E -> T E'

void E() {

T();

EPrime();

}

// E' -> + T E' | ε void EPrime() {

if (input[pos] == '+') { // If '+' is found match('+');

T();

EPrime();

}

}

// T -> F T'

void T() {

F();

TPrime();

}

// T' -> \* F T' | ε void TPrime() {

if (input[pos] == '\*') { // If '\*' is found match('\*');

F();

TPrime();

}

}

// F -> (E) | id (assuming 'id' starts with 'i') void F() {

if (input[pos] == '(') { // If '(' is found match('(');

E();

match(')');

} else if (input[pos] == 'i') { // Assuming 'id' is represented as 'i' match('i');

} else {

error();

}

}

int main() {

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

E(); // Start parsing with E

// If the entire input is parsed successfully if (input[pos] == '\0')

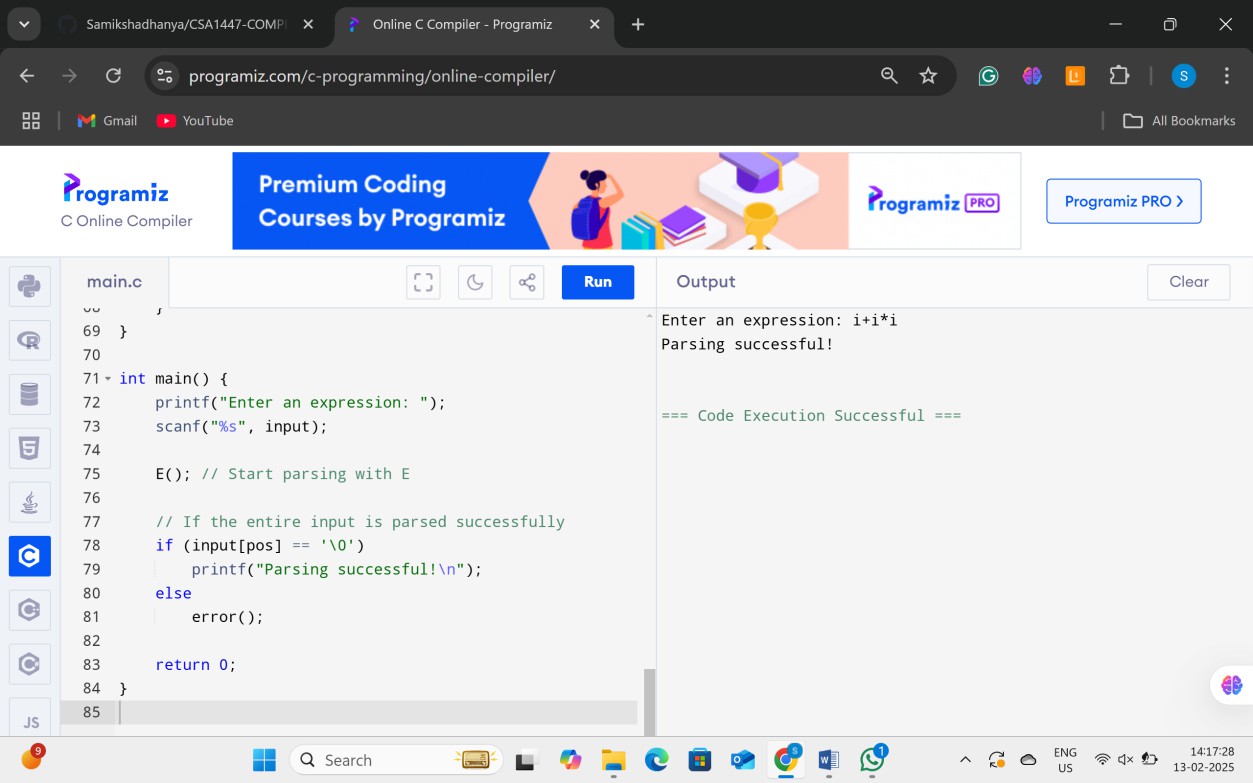
printf("Parsing successful!\n"); else

error();

return 0;

}

**OUTPUT:**



**11. Expression Evaluation Using Operator Precedence**

#include <stdio.h> #include <ctype.h> #include <stdlib.h>

int precedence(char op) {

if (op == '+' || op == '-') return 1; if (op == '\*' || op == '/') return 2; if (op == '^') return 3;

return 0;

}

int applyOp(int a, int b, char op) { switch (op) {

case '+': return a + b; case '-': return a - b; case '\*': return a \* b;

case '/': return a / b; case '^': {

int res = 1;

for (int i = 0; i < b; i++) res \*= a; return res;

}

}

return 0;

}

int evaluateExpression(char\* expr) { int values[100], valTop = -1;

char ops[100]; int opsTop = -1;

for (int i = 0; expr[i] != '\0'; i++) { if (isdigit(expr[i])) {

int val = 0;

while (isdigit(expr[i])) {

val = (val \* 10) + (expr[i] - '0'); i++;

}

i--;

values[++valTop] = val;

} else if (expr[i] == '(') { ops[++opsTop] = expr[i];

} else if (expr[i] == ')') {

while (opsTop != -1 && ops[opsTop] != '(') { int b = values[valTop--];

int a = values[valTop--]; char op = ops[opsTop--];

values[++valTop] = applyOp(a, b, op);

}

opsTop--;

} else {

while (opsTop != -1 && precedence(ops[opsTop]) >= precedence(expr[i])) { int b = values[valTop--];

int a = values[valTop--]; char op = ops[opsTop--];

values[++valTop] = applyOp(a, b, op);

}

ops[++opsTop] = expr[i];

}

}

while (opsTop != -1) {

int b = values[valTop--]; int a = values[valTop--]; char op = ops[opsTop--];

values[++valTop] = applyOp(a, b, op);

}

return values[valTop];

}

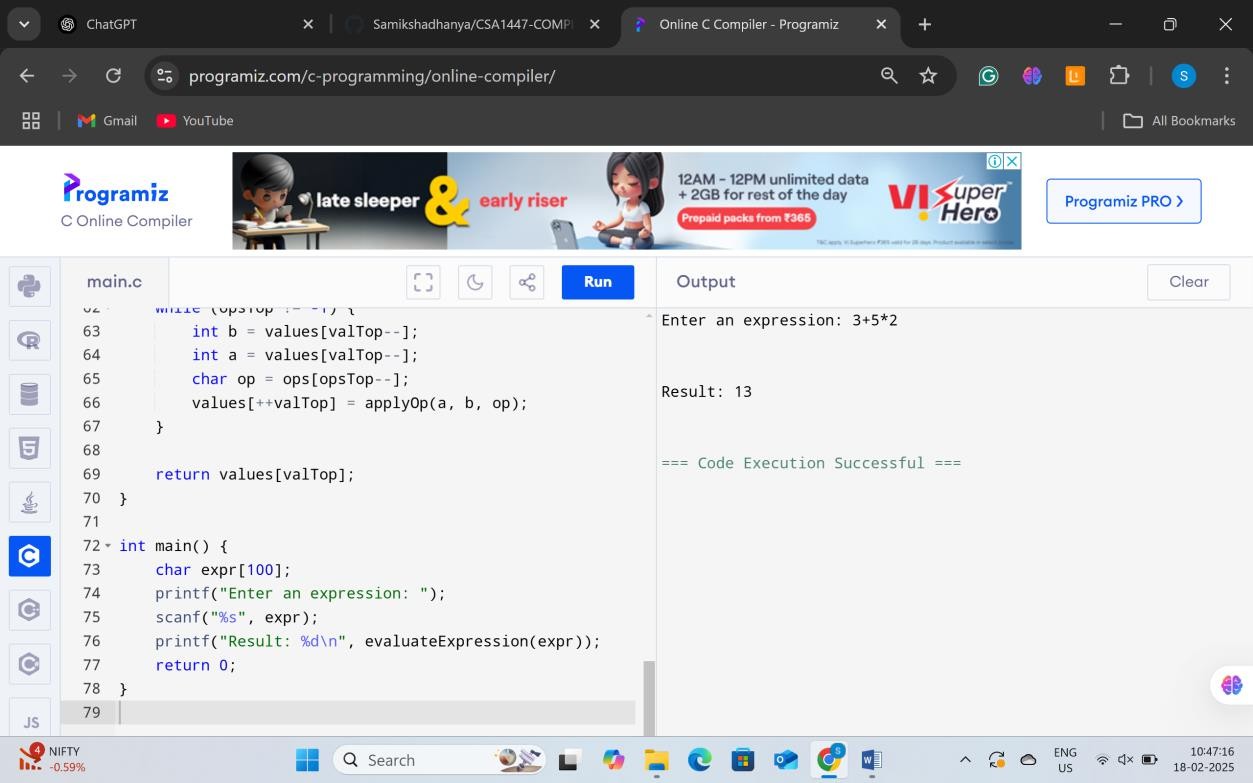
int main() {

char expr[100];

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

printf("Result: %d\n", evaluateExpression(expr)); return 0;

}

**OUTPUT**:

**12. Three Address Code (TAC) Generation**

#include <stdio.h> #include <string.h>

void generateTAC(char expr[]) { char tempVar = 'T';

int tempIndex = 1; char op;

int i, len = strlen(expr);

printf("Three Address Code:\n");

for (i = 0; i < len; i++) {

if (expr[i] == '+' || expr[i] == '-' || expr[i] == '\*' || expr[i] == '/') { op = expr[i];

printf("%c%d = %c %c %c\n", tempVar, tempIndex, expr[i - 1], op, expr[i + 1]);

expr[i + 1] = tempVar + tempIndex - '0'; tempIndex++;

}

}

}

int main() {

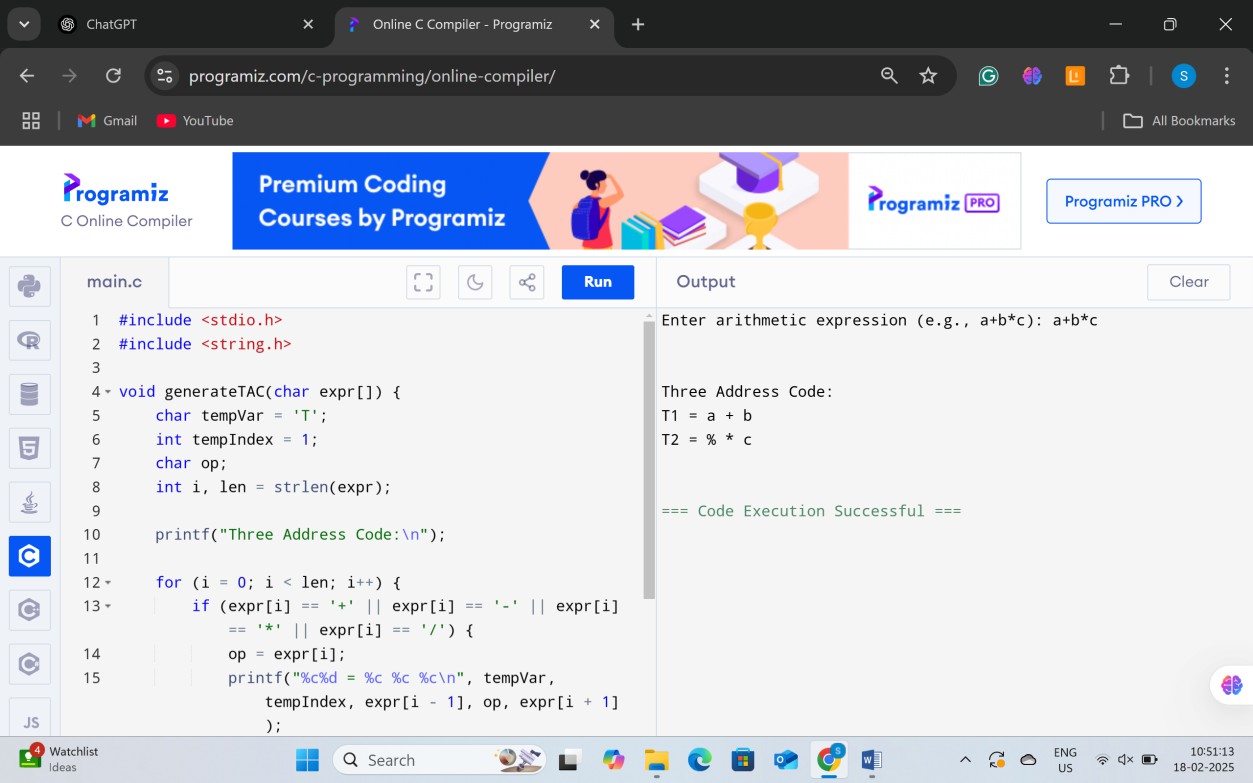
char expr[50];

printf("Enter arithmetic expression (e.g., a+b\*c): "); scanf("%s", expr);

generateTAC(expr); return 0;

}

**OUTPUT:**



**13. Character, Word, and Line Counter**

#include <stdio.h>

int main() {

char str[] = "Hello World\nThis is a test"; int chars = 0, words = 1, lines = 1;

for (int i = 0; str[i] != '\0'; i++) { chars++;

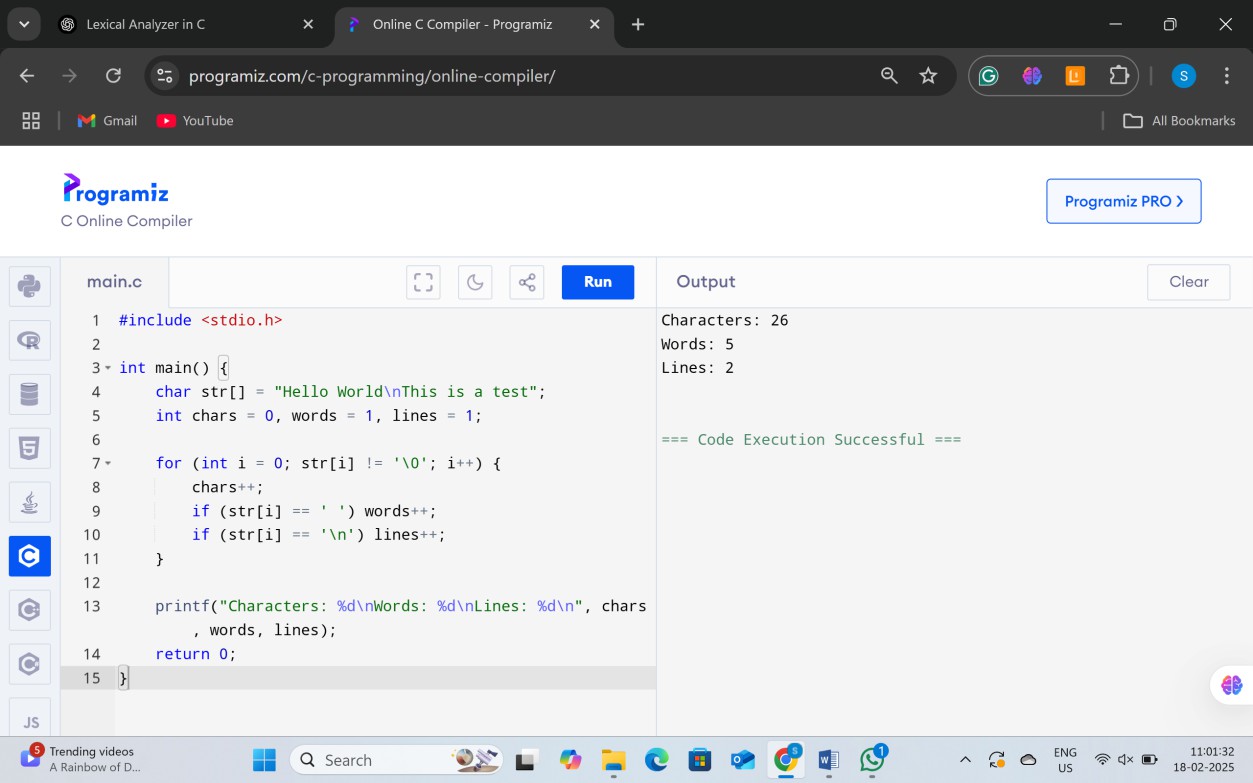
if (str[i] == ' ') words++; if (str[i] == '\n') lines++;

}

printf("Characters: %d\nWords: %d\nLines: %d\n", chars, words, lines); return 0;

}

**OUTPUT:**



**14.** **Intermediate Code Generation**

#include <stdio.h>

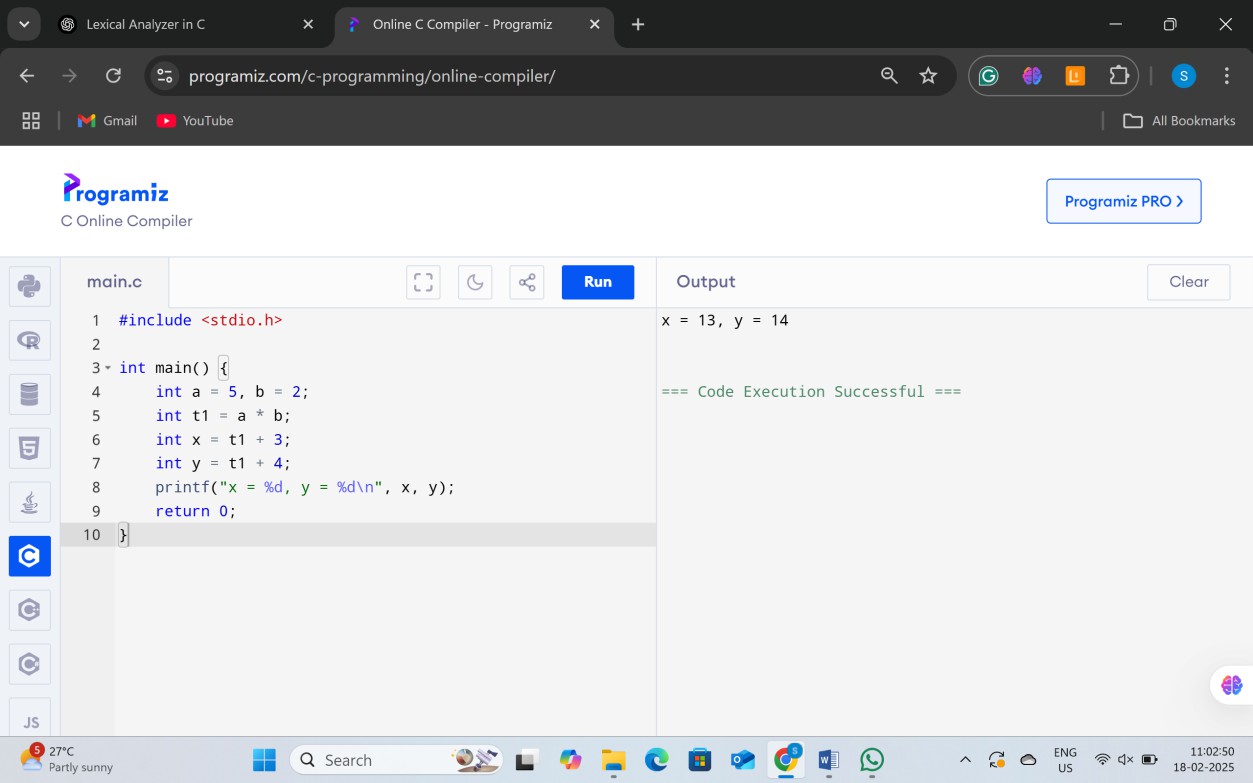
int main() {

int a = 5, b = 2; int t1 = a \* b; int x = t1 + 3; int y = t1 + 4;

printf("x = %d, y = %d\n", x, y); return 0;

}

**OUTPUT:**



**15. Character, Word, and Line Counter in a Source Program**

#include <stdio.h> #include <ctype.h>

int main() {

char str[] = "#include <stdio.h>\nint main() {\n int a = 10;\n printf(\"Hello World\");\n return 0;\n}\n";

int char\_count = 0, word\_count = 0, line\_count = 1; int in\_word = 0;

printf("Input Source Program:\n%s\n\n", str);

for (int i = 0; str[i] != '\0'; i++) { char\_count++;

if (str[i] == '\n') line\_count++;

if (isspace(str[i])) { in\_word = 0;

} else if (!in\_word) { in\_word = 1; word\_count++;

}

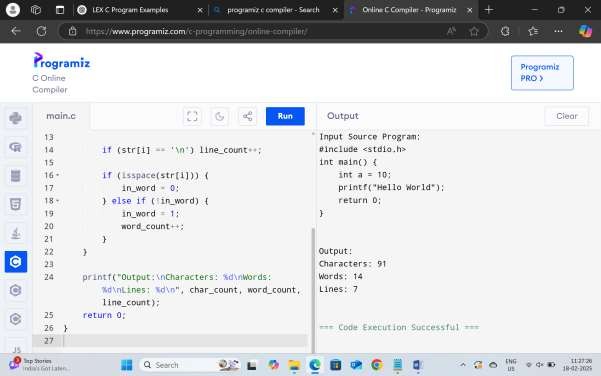
}

printf("Output:\nCharacters: %d\nWords: %d\nLines: %d\n", char\_count, word\_count, line\_count);

return 0;

}

**OUTPUT:**



**16. Extracting Numeric Constants from a Source Program**

#include <stdio.h> #include <ctype.h> #include <stdlib.h>

int main() {

char str[] = "#define PI 3.14\n#include<stdio.h>\nint main() {\n int a = 10, b = 20;\n printf(\"%d\", a + b);\n return 0;\n}\n";

printf("Input Source Program:\n%s\n\n", str); printf("Output:\n");

for (int i = 0; str[i] != '\0'; i++) { if (isdigit(str[i])) {

while (isdigit(str[i]) || str[i] == '.') { printf("%c", str[i]);

i++;

}

printf("\n");

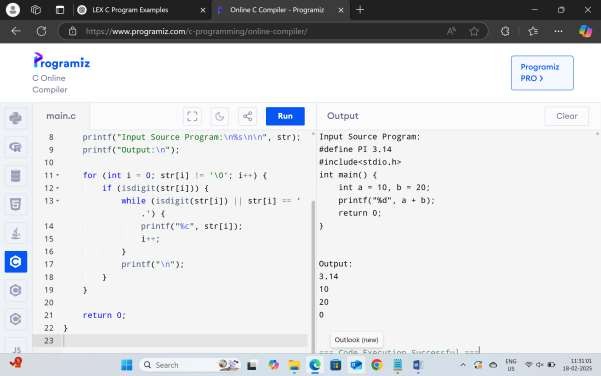
}

}

return 0;

}

**OUTPUT:**



**17. Counting Macros and Header Files in a Source Program**

#include <stdio.h> #include <string.h>

int main() {

char str[] = "#define PI 3.14\n#include<stdio.h>\n#include<conio.h>\nint main() { return 0; }\n"; int macro\_count = 0, header\_count = 0;

printf("Input Source Program:\n%s\n\n", str);

char \*line = strtok(str, "\n"); while (line) {

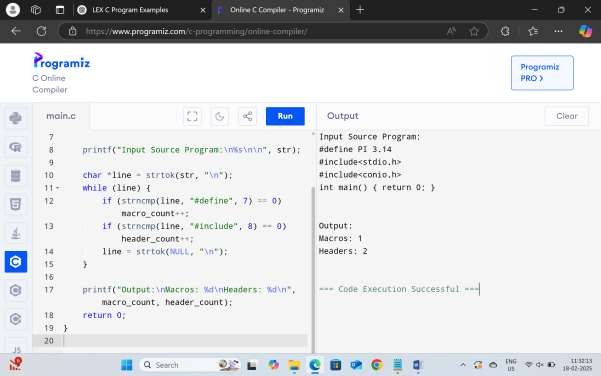
if (strncmp(line, "#define", 7) == 0) macro\_count++; if (strncmp(line, "#include", 8) == 0) header\_count++; line = strtok(NULL, "\n");

}

printf("Output:\nMacros: %d\nHeaders: %d\n", macro\_count, header\_count); return 0;

}

**OUTPUT:**



**18. Counting Macros and Header Files (Duplicate of Experiment 17)**

#include <stdio.h> #include <string.h>

int main() {

char str[] = "#define PI 3.14\n#include<stdio.h>\n#include<conio.h>\nint main() { return 0; }\n"; int macro\_count = 0, header\_count = 0;

printf("Input Source Program:\n%s\n\n", str);

char \*line = strtok(str, "\n"); while (line) {

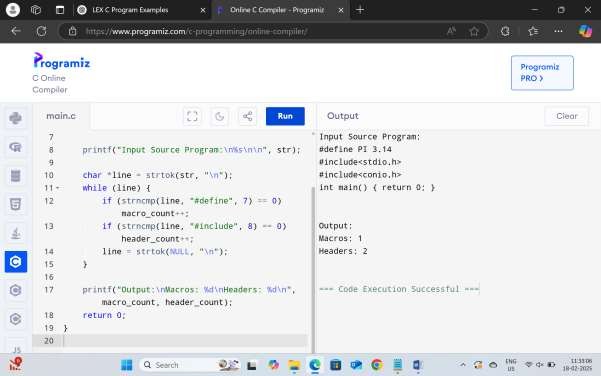
if (strncmp(line, "#define", 7) == 0) macro\_count++; if (strncmp(line, "#include", 8) == 0) header\_count++; line = strtok(NULL, "\n");

}

printf("Output:\nMacros: %d\nHeaders: %d\n", macro\_count, header\_count); return 0;

}

**OUTPUT:**



**19. Extracting HTML Tags from an HTML File**

#include <stdio.h>

int main() {

char str[] = "<html>\n<body>\n<h1>Title</h1>\n<p>Paragraph</p>\n</body>\n</html>\n";

printf("Input HTML File:\n%s\n\n", str); printf("Output:\n");

int inside\_tag = 0;

for (int i = 0; str[i] != '\0'; i++) { if (str[i] == '<') {

inside\_tag = 1; printf("Tag: ");

}

if (inside\_tag) { printf("%c", str[i]);

}

if (str[i] == '>') { inside\_tag = 0; printf("\n");

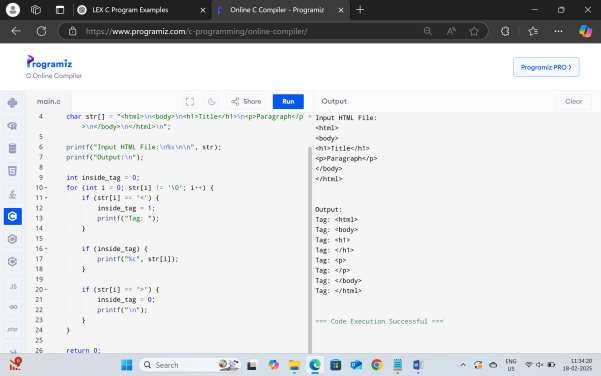
}

}

return 0;

}

**OUTPUT:**



**20. Adding Line Numbers to a C Program**

#include <stdio.h>

int main() {

char str[] = "#define PI 3.14\n#include<stdio.h>\nint main() {\n int a = 10;\n printf(\"Hello\");\n return 0;\n}\n";

int line\_num = 1;

printf("Input C Program:\n%s\n\n", str); printf("Output:\n");

printf("%d: ", line\_num++); for (int i = 0; str[i] != '\0'; i++) {

printf("%c", str[i]);

if (str[i] == '\n' && str[i + 1] != '\0') {

printf("%d: ", line\_num++);

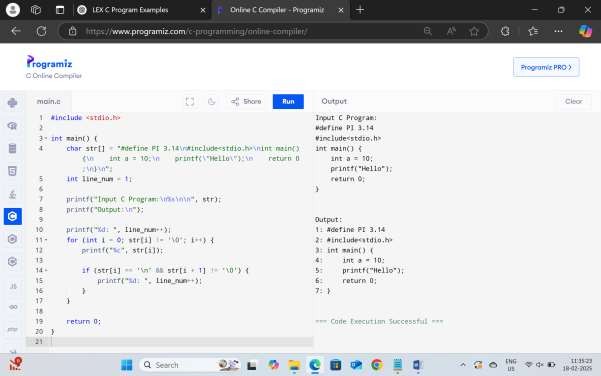
}

}

return 0;

}

**OUTPUT:**



**21. Character, Word, and Line Counter (Similar to Experiment 15)**

#include <stdio.h> #include <ctype.h>

int main() {

// Sample Input: A small C program stored in a string char input[] =

"#include <stdio.h>\n" "int main() {\n"

" int a = 10, b = 20;\n"

" printf(\"Hello, World!\");\n" " return 0;\n"

"}\n";

int char\_count = 0, word\_count = 0, line\_count = 1; int in\_word = 0;

printf("Input Source Program:\n%s\n\n", input);

for (int i = 0; input[i] != '\0'; i++) { char\_count++;

if (input[i] == '\n') line\_count++;

if (isspace(input[i])) {

in\_word = 0; // End of a word

} else if (!in\_word) { in\_word = 1;

word\_count++; // Start of a new word

}

}

// Print results printf("Output:\n");

printf("Characters: %d\n", char\_count); printf("Words: %d\n", word\_count); printf("Lines: %d\n", line\_count);

return 0;

}

**OUTPUT:**

