1. Implement a C program to perform symbol table operations.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 10

// Structure for a symbol table entry

typedef struct Symbol {

char identifier[50];

char type[20];

struct Symbol\* next;

} Symbol;

// Hash table

Symbol\* symbolTable[TABLE\_SIZE];

// Hash function

int hashFunction(char\* identifier) {

int hash = 0;

while (\*identifier)

hash = (hash + \*identifier++) % TABLE\_SIZE;

return hash;

}

// Insert into the symbol table

void insert(char\* identifier, char\* type) {

int index = hashFunction(identifier);

// Check if the identifier already exists

Symbol\* temp = symbolTable[index];

while (temp) {

if (strcmp(temp->identifier, identifier) == 0) {

printf("Identifier '%s' already exists.\n", identifier);

return;

}

temp = temp->next;

}

// Create a new symbol

Symbol\* newSymbol = (Symbol\*)malloc(sizeof(Symbol));

strcpy(newSymbol->identifier, identifier);

strcpy(newSymbol->type, type);

newSymbol->next = symbolTable[index]; // Insert at the beginning of the chain

symbolTable[index] = newSymbol;

printf("Inserted: %s -> %s\n", identifier, type);

}

// Search for an identifier

void search(char\* identifier) {

int index = hashFunction(identifier);

Symbol\* temp = symbolTable[index];

while (temp) {

if (strcmp(temp->identifier, identifier) == 0) {

printf("Found: %s -> %s\n", temp->identifier, temp->type);

return;

}

temp = temp->next;

}

printf("Identifier '%s' not found.\n", identifier);

}

// Display the symbol table

void display() {

printf("\nSymbol Table:\n");

for (int i = 0; i < TABLE\_SIZE; i++) {

Symbol\* temp = symbolTable[i];

if (temp) {

printf("Index %d: ", i);

while (temp) {

printf("(%s, %s) -> ", temp->identifier, temp->type);

temp = temp->next;

}

printf("NULL\n");

}

}

}

int main() {

int choice;

char identifier[50], type[20];

while (1) {

printf("\n1. Insert\n2. Search\n3. Display\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter identifier: ");

scanf("%s", identifier);

printf("Enter type: ");

scanf("%s", type);

insert(identifier, type);

break;

case 2:

printf("Enter identifier to search: ");

scanf("%s", identifier);

search(identifier);

break;

case 3:

display();

break;

case 4:

exit(0);

default:

printf("Invalid choice!\n");

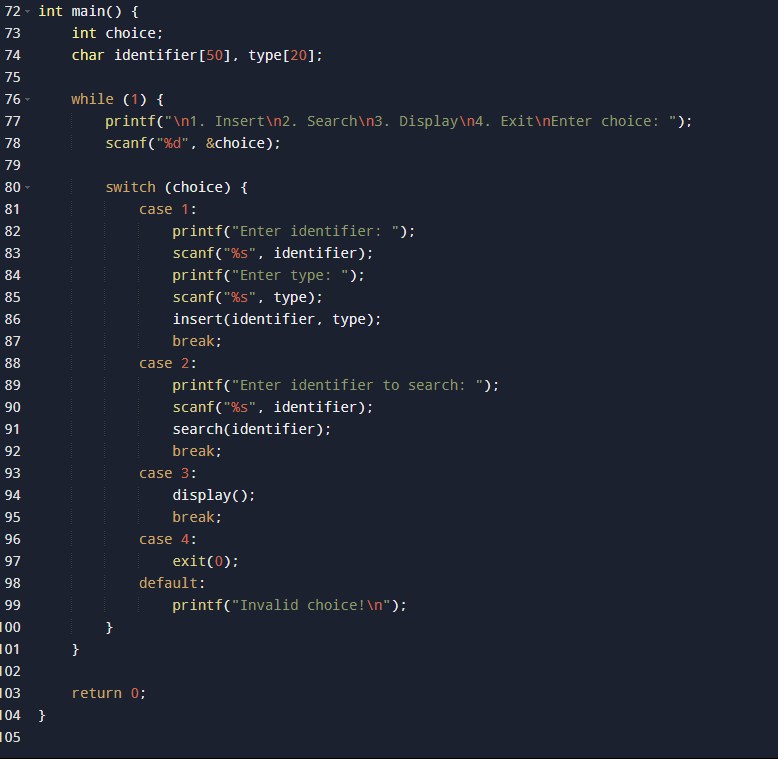
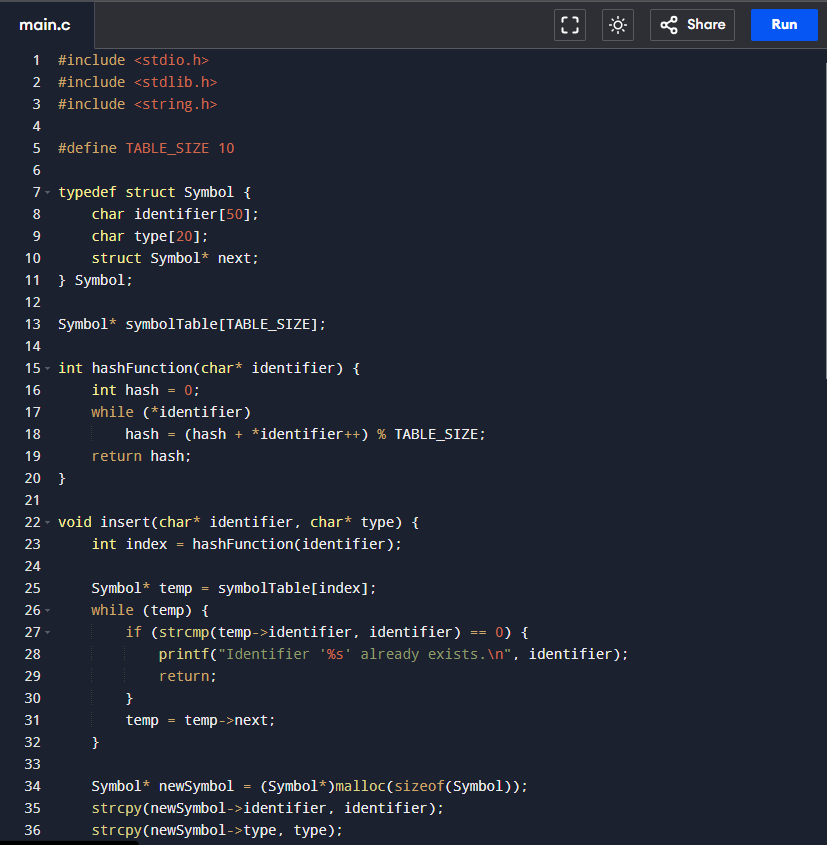
}

}

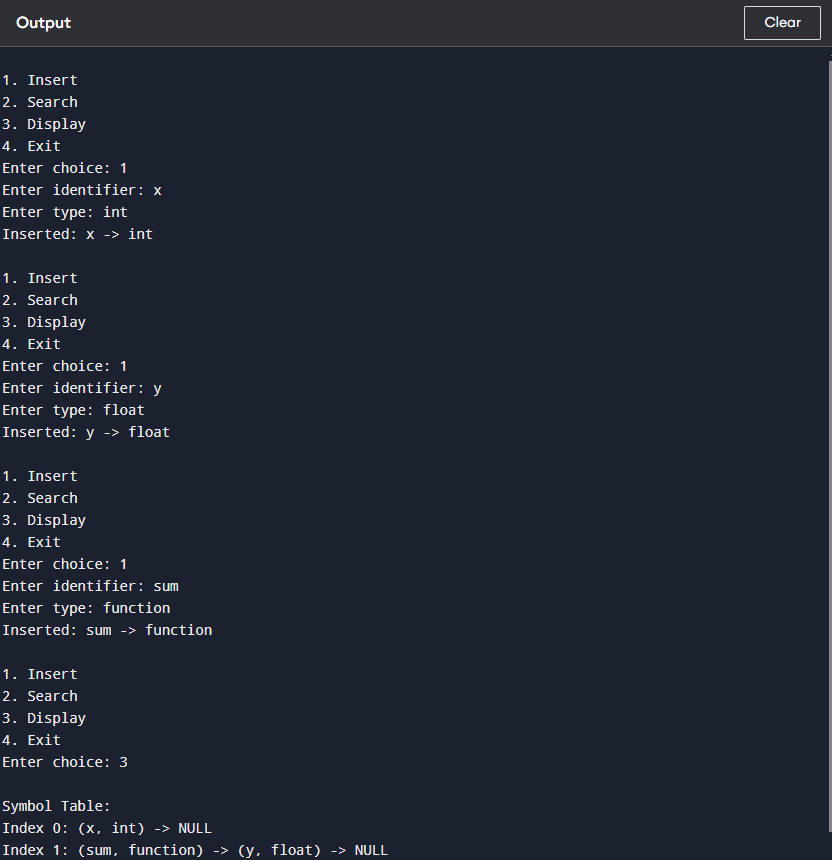
return 0;

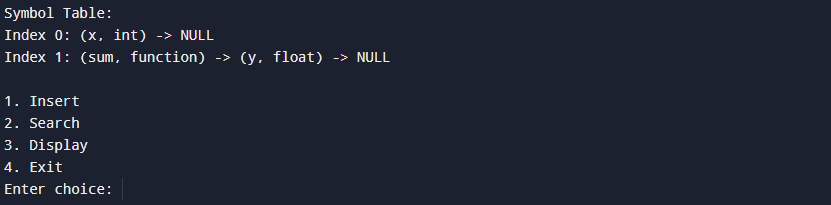
}

**CODE**

****

**OUTPUT:**

****

****

1. Grammar Validation

**CODE:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX\_WORDS 10

#define MAX\_LENGTH 20

// Predefined words in grammar

const char\* subjects[] = {"I", "You", "He", "She", "It", "We", "They"};

const char\* verbs[] = {"eat", "play", "read", "write", "drink"};

const char\* objects[] = {"apple", "book", "game", "water", "story"};

// Function to check if a word belongs to a given list

int isInList(const char\* word, const char\* list[], int size) {

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

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

return 1; // Found

}

}

return 0; // Not Found

}

// Function to check if a sentence follows the Subject-Verb-Object structure

int checkGrammar(char\* sentence) {

char\* words[MAX\_WORDS];

int count = 0;

// Tokenizing the sentence into words

char\* token = strtok(sentence, " ");

while (token != NULL) {

words[count++] = token;

token = strtok(NULL, " ");

}

// Sentence must have exactly 3 words: Subject Verb Object

if (count != 3) {

printf("Invalid sentence: Incorrect structure!\n");

return 0;

}

// Checking grammar rules

if (!isInList(words[0], subjects, sizeof(subjects) / sizeof(subjects[0]))) {

printf("Invalid sentence: Incorrect subject!\n");

return 0;

}

if (!isInList(words[1], verbs, sizeof(verbs) / sizeof(verbs[0]))) {

printf("Invalid sentence: Incorrect verb!\n");

return 0;

}

if (!isInList(words[2], objects, sizeof(objects) / sizeof(objects[0]))) {

printf("Invalid sentence: Incorrect object!\n");

return 0;

}

printf("Valid sentence: Follows grammar!\n");

return 1;

}

int main() {

char sentence[MAX\_LENGTH \* MAX\_WORDS];

printf("Enter a sentence (Subject Verb Object): ");

fgets(sentence, sizeof(sentence), stdin);

// Remove newline character from input

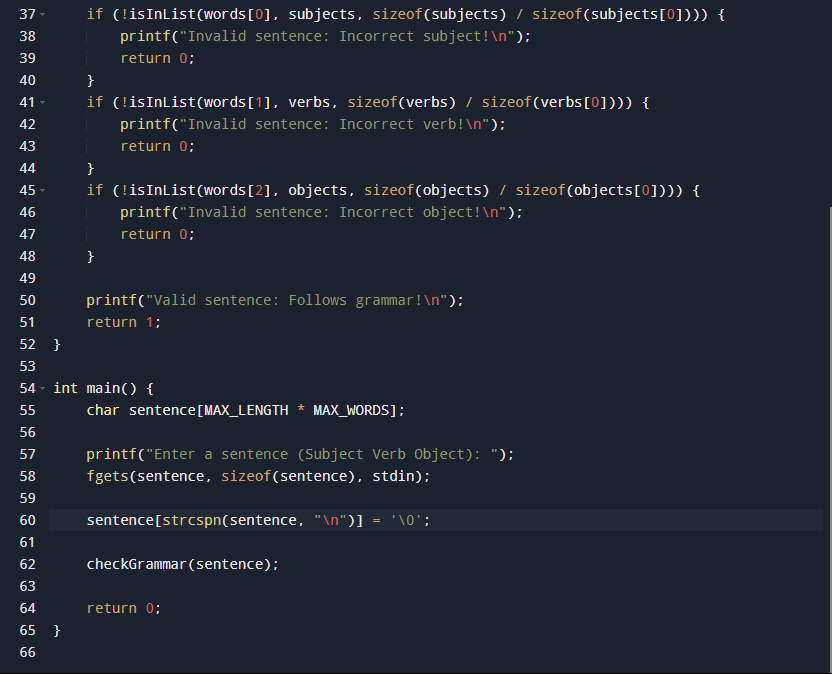
sentence[strcspn(sentence, "\n")] = '\0';

checkGrammar(sentence);

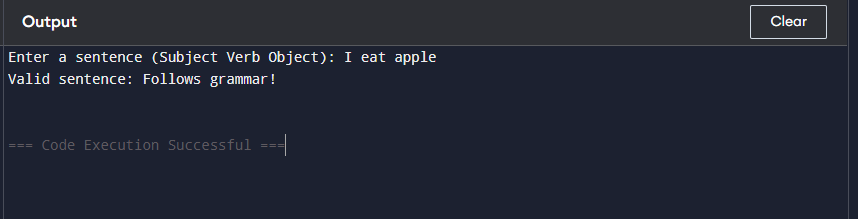
return 0;

}





**OUTPUT:**

****

1. **Recursive Descent Parsing**

#include <stdio.h>

#include <ctype.h>

char \*input; // Input expression

// Function to match and move to the next character

void match(char expected) {

if (\*input == expected) {

input++; // Move to next character

} else {

printf("Syntax Error: Unexpected character '%c'\n", \*input);

exit(1);

}

}

// Forward declarations

void E(); // Expression

void T(); // Term

void F(); // Factor

// Expression: E → T { ('+' | '-') T }

void E() {

T();

while (\*input == '+' || \*input == '-') {

char op = \*input;

match(op);

T();

printf("Processed Operator: %c\n", op);

}

}

// Term: T → F { ('\*' | '/') F }

void T() {

F();

while (\*input == '\*' || \*input == '/') {

char op = \*input;

match(op);

F();

printf("Processed Operator: %c\n", op);

}

}

// Factor: F → (E) | number

void F() {

if (isdigit(\*input)) {

printf("Processed Number: %c\n", \*input);

match(\*input);

} else if (\*input == '(') {

match('(');

E();

match(')');

} else {

printf("Syntax Error: Unexpected character '%c'\n", \*input);

exit(1);

}

}

// Main function to parse an expression

int main() {

char expr[100];

printf("Enter an arithmetic expression: ");

scanf("%s", expr); // Read input without spaces

input = expr; // Initialize input pointer

E(); // Start parsing from Expression

if (\*input == '\0') {

printf("Parsing Successful: Expression is Valid!\n");

} else {

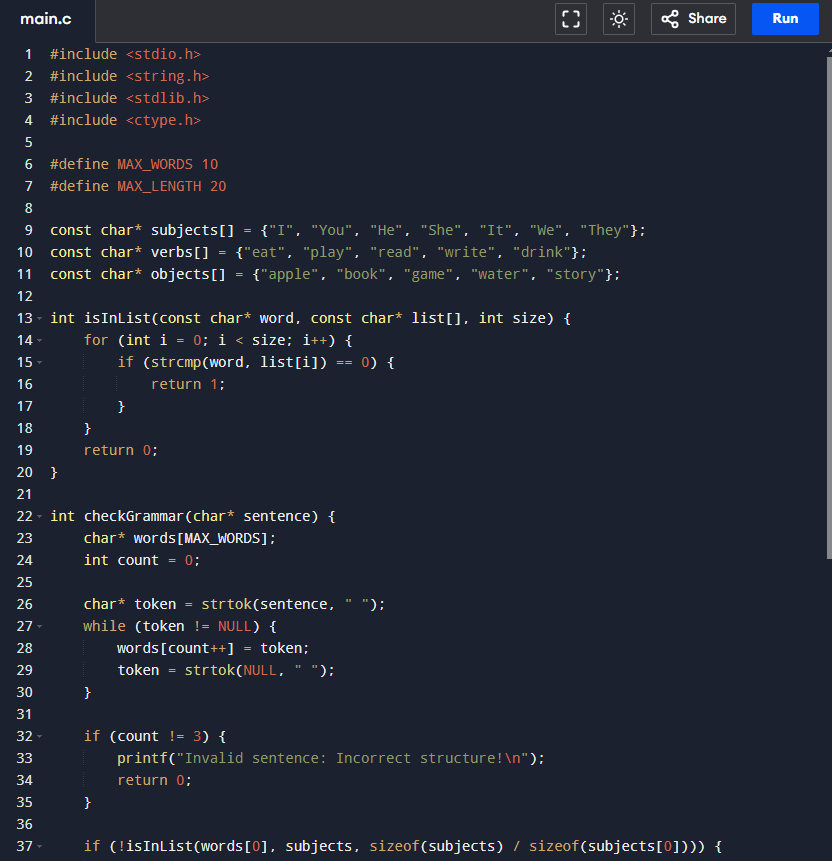
printf("Syntax Error: Unexpected characters after valid expression!\n");

}

return 0;

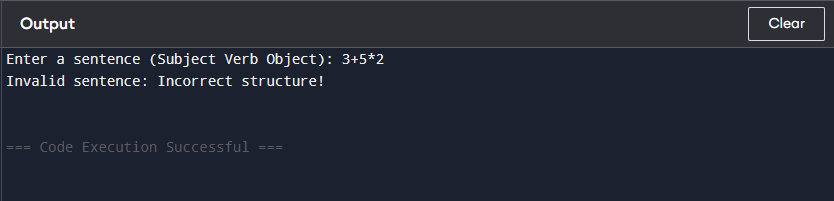
}

**CODE**:





**OUTPUT**:



**11. Operator Precedence Parsing – PEMDAS**:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

char \*input; // Input expression pointer

// Function prototypes

double E(); // Expression

double T(); // Term

double F(); // Factor

double P(); // Parentheses

// Function to match the expected character and move forward

void match(char expected) {

if (\*input == expected) {

input++;

}

}

// Factor: F → Number | Parentheses | Exponentiation

double F() {

double value;

if (\*input == '(') {

match('(');

value = E(); // Solve the expression inside parentheses

match(')');

} else if (isdigit(\*input)) {

value = strtod(input, &input); // Convert number string to double

} else {

printf("Syntax Error: Unexpected character '%c'\n", \*input);

exit(1);

}

// Handle exponentiation (^)

while (\*input == '^') {

match('^');

value = pow(value, F()); // Compute exponentiation

}

return value;

}

// Term: T → F { ('\*' | '/') F }

double T() {

double value = F();

while (\*input == '\*' || \*input == '/') {

char op = \*input;

match(op);

if (op == '\*') {

value \*= F();

} else {

value /= F();

}

}

return value;

}

// Expression: E → T { ('+' | '-') T }

double E() {

double value = T();

while (\*input == '+' || \*input == '-') {

char op = \*input;

match(op);

if (op == '+') {

value += T();

} else {

value -= T();

}

}

return value;

}

// Main function to evaluate an expression

int main() {

char expr[100];

printf("Enter an arithmetic expression: ");

scanf("%s", expr); // Read input expression without spaces

input = expr; // Initialize input pointer

double result = E(); // Evaluate the expression

if (\*input == '\0') {

printf("Result: %.2f\n", result);

} else {

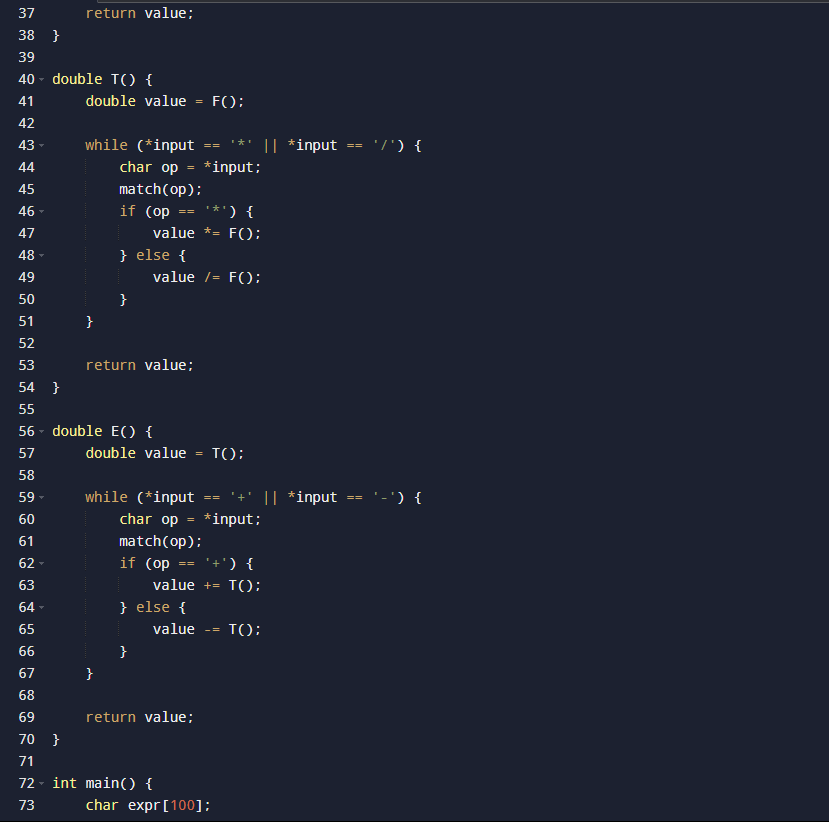
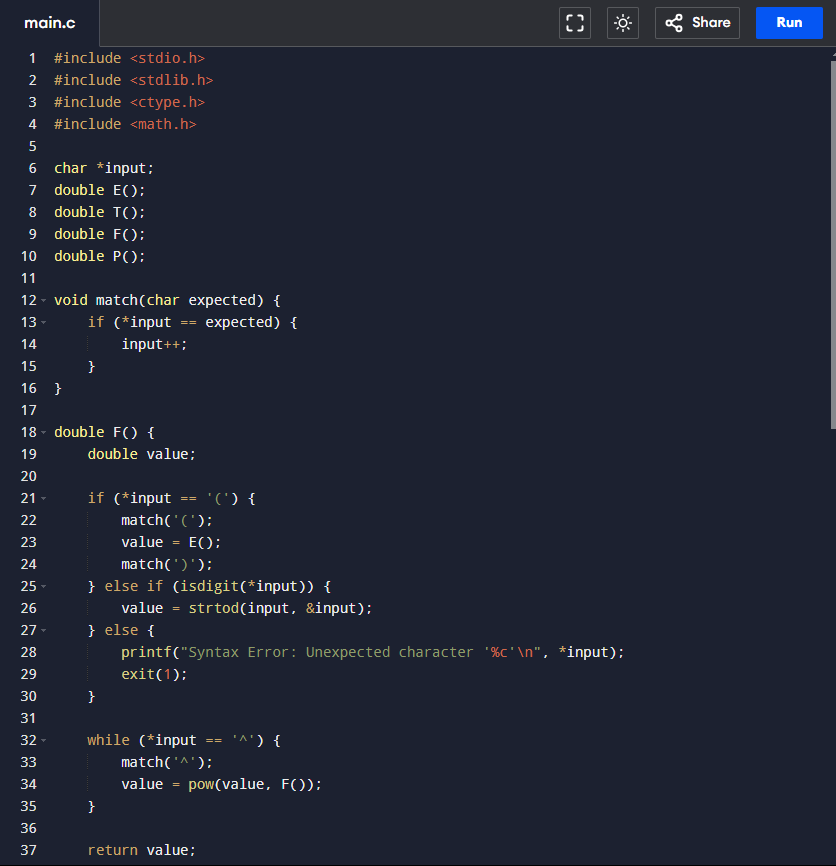
printf("Syntax Error: Unexpected characters after valid expression!\n");

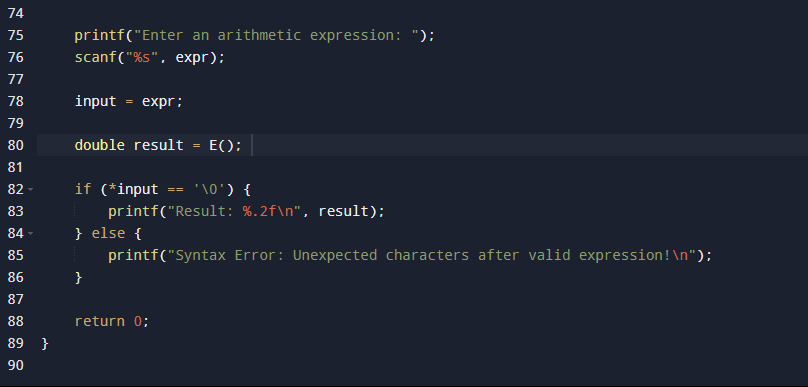
}

return 0;

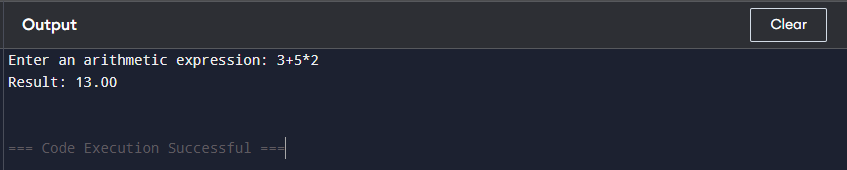
}

**CODE**:





**OUTPUT**:



**12.Three-Address Code Generation**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX\_LEN 100

char expr[MAX\_LEN]; // Input expression

char \*input; // Pointer to traverse expression

int tempVar = 1; // Temporary variable counter

// Function prototypes

void E(); // Expression

void T(); // Term

void F(); // Factor

void generateTAC(char op, char operand1[], char operand2[], char result[]);

// Structure to store TAC instructions

typedef struct {

char op;

char arg1[MAX\_LEN];

char arg2[MAX\_LEN];

char result[MAX\_LEN];

} TAC;

TAC tacList[MAX\_LEN];

int tacCount = 0;

// Generate a Three-Address Code statement

void generateTAC(char op, char operand1[], char operand2[], char result[]) {

tacList[tacCount].op = op;

strcpy(tacList[tacCount].arg1, operand1);

strcpy(tacList[tacCount].arg2, operand2);

strcpy(tacList[tacCount].result, result);

tacCount++;

}

// Parse Factor: F → (E) | id

void F(char result[]) {

if (\*input == '(') {

input++; // Skip '('

E(result);

input++; // Skip ')'

} else if (isalnum(\*input)) {

sprintf(result, "%c", \*input); // Store variable

input++;

} else {

printf("Syntax Error: Unexpected character '%c'\n", \*input);

exit(1);

}

}

// Parse Term: T → F { ('\*' | '/') F }

void T(char result[]) {

char left[MAX\_LEN], right[MAX\_LEN], temp[MAX\_LEN];

F(left);

while (\*input == '\*' || \*input == '/') {

char op = \*input;

input++; // Skip operator

F(right);

sprintf(temp, "t%d", tempVar++); // Generate temp variable

generateTAC(op, left, right, temp);

strcpy(left, temp);

}

strcpy(result, left);

}

// Parse Expression: E → T { ('+' | '-') T }

void E(char result[]) {

char left[MAX\_LEN], right[MAX\_LEN], temp[MAX\_LEN];

T(left);

while (\*input == '+' || \*input == '-') {

char op = \*input;

input++; // Skip operator

T(right);

sprintf(temp, "t%d", tempVar++); // Generate temp variable

generateTAC(op, left, right, temp);

strcpy(left, temp);

}

strcpy(result, left);

}

// Main function

int main() {

char result[MAX\_LEN];

printf("Enter an arithmetic expression: ");

scanf("%s", expr); // Read input without spaces

input = expr; // Initialize input pointer

E(result); // Start parsing

if (\*input == '\0') {

printf("\nThree-Address Code Representation:\n");

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

printf("%s = %s %c %s\n", tacList[i].result, tacList[i].arg1, tacList[i].op, tacList[i].arg2);

}

} else {

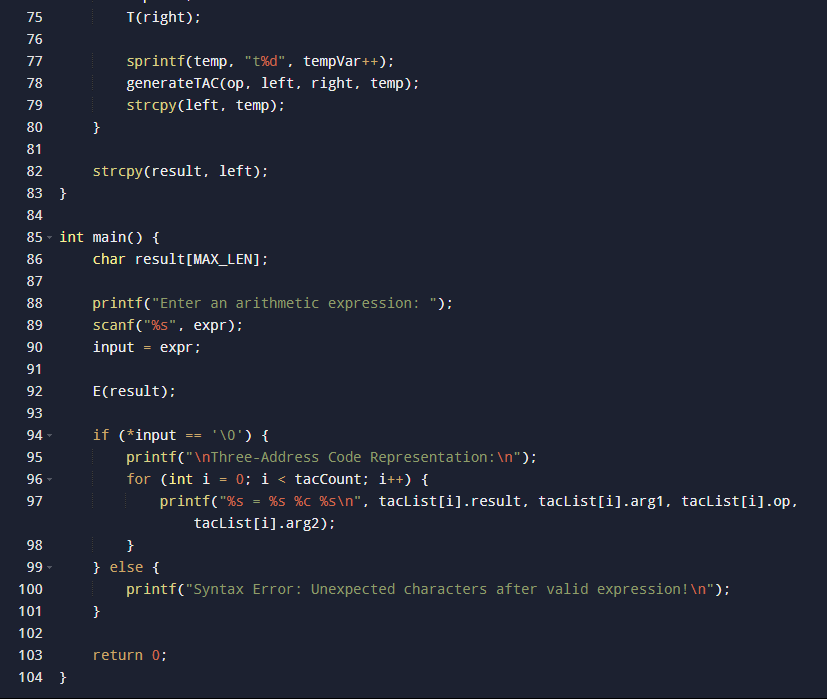
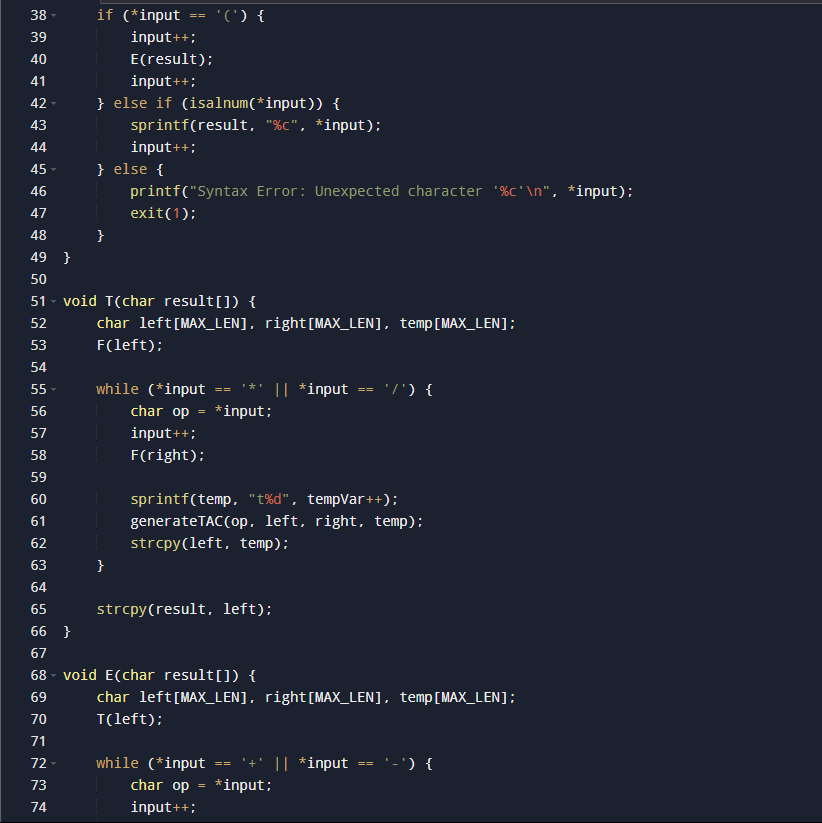
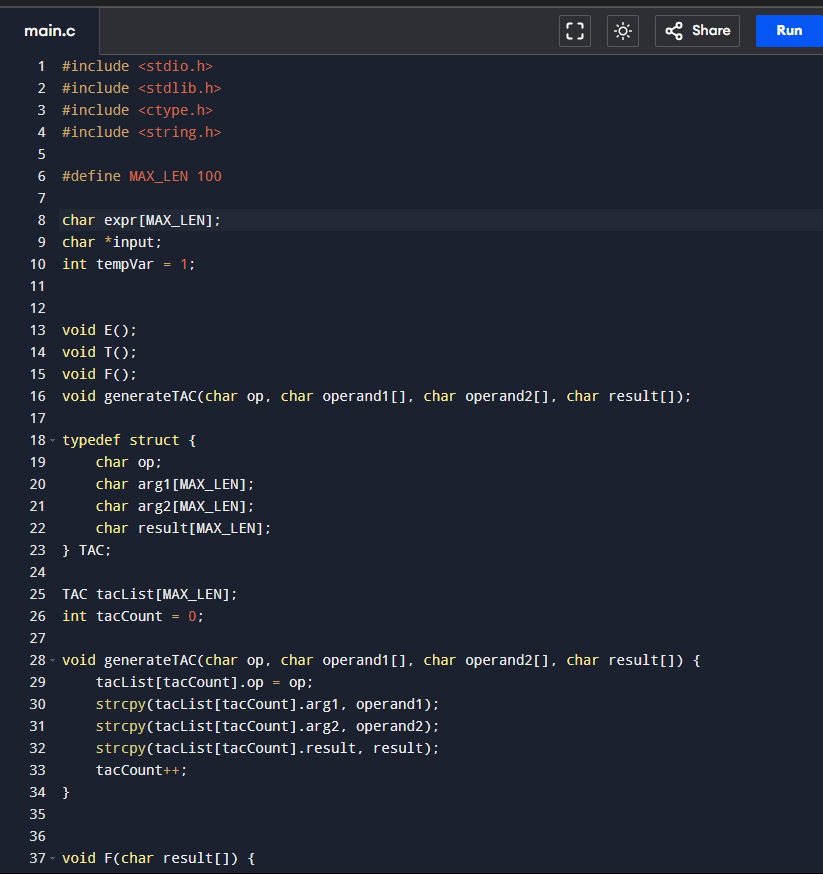
printf("Syntax Error: Unexpected characters after valid expression!\n");

}

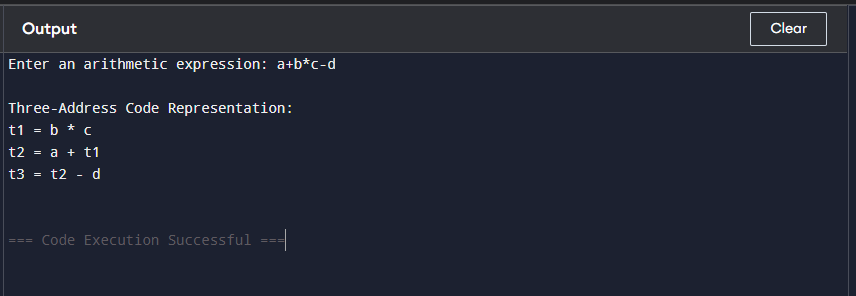
return 0;

}

**CODE**:



**OUTPUT**:



**13. Lexical Analyser: Counting characters, words and lines**

#include <stdio.h>

#include <ctype.h>

int main() {

FILE \*file;

char filename[100], ch;

int characters = 0, words = 0, lines = 0;

int inWord = 0; // Flag to track if we are inside a word

// Get the filename from the user

printf("Enter the filename: ");

scanf("%s", filename);

// Open the file in read mode

file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file.\n");

return 1;

}

// Process file character by character

while ((ch = fgetc(file)) != EOF) {

characters++; // Count every character

if (ch == '\n') {

lines++; // Count new lines

}

// Check if the character is part of a word

if (isspace(ch) || ch == '\n' || ch == '\t') {

inWord = 0; // End of a word

} else if (!inWord) {

inWord = 1;

words++; // Start of a new word

}

}

// Close the file

fclose(file);

// Display the results

printf("\nFile Analysis:\n");

printf("Characters: %d\n", characters);

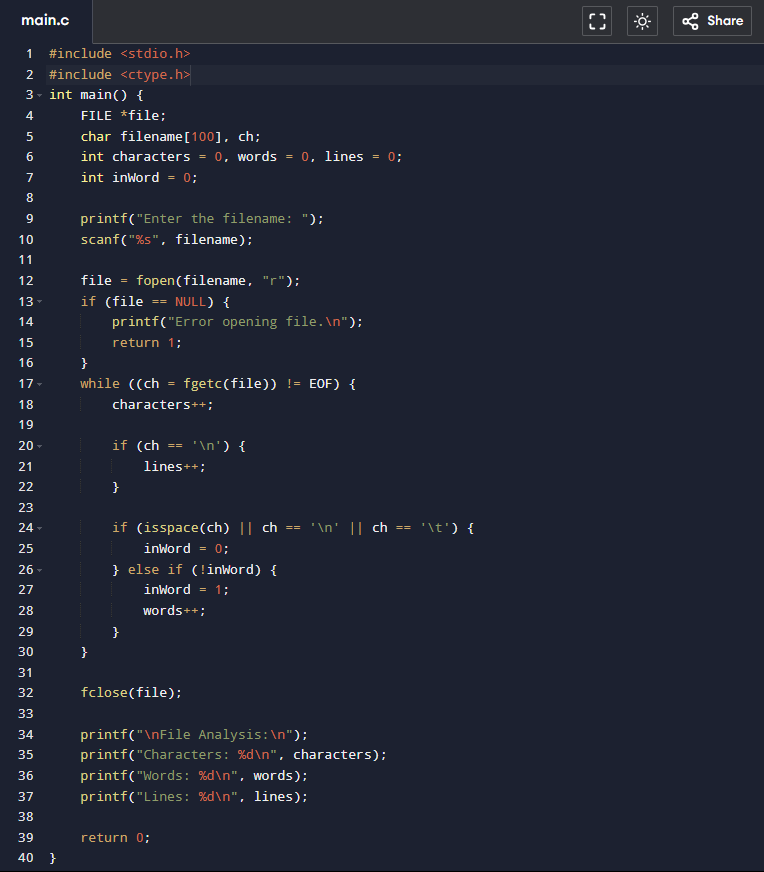
printf("Words: %d\n", words);

printf("Lines: %d\n", lines);

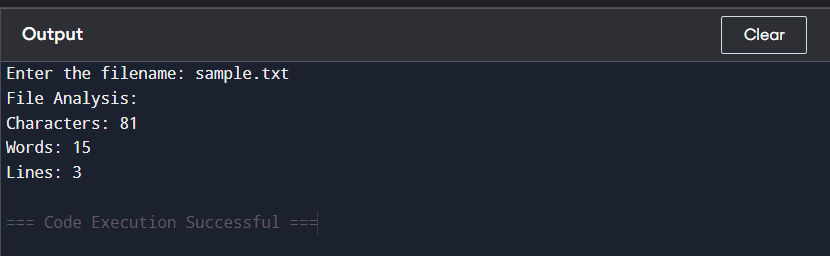
return 0;

}

**CODE**:



**OUTPUT**:



14. Eliminating Common Subexpression

#include <stdio.h>

#include <string.h>

#define MAX 100

// Structure to store expressions

typedef struct {

char expr[MAX];

char tempVar[10]; // Stores temporary variable name

} Expression;

Expression exprList[MAX]; // Array to store expressions

int exprCount = 0; // Number of expressions

// Function to check if an expression already exists

int findCommonSubexpression(char \*expr) {

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

if (strcmp(exprList[i].expr, expr) == 0) {

return i; // Return index of existing expression

}

}

return -1; // Not found

}

// Function to optimize expressions

void optimizeExpression(char \*lhs, char \*rhs) {

int index = findCommonSubexpression(rhs);

if (index != -1) {

// If common subexpression exists, reuse temp variable

printf("%s = %s\n", lhs, exprList[index].tempVar);

} else {

// Create a new temporary variable

char tempVar[10];

sprintf(tempVar, "t%d", exprCount + 1);

// Store the expression

strcpy(exprList[exprCount].expr, rhs);

strcpy(exprList[exprCount].tempVar, tempVar);

exprCount++;

// Print optimized expression

printf("%s = %s\n", tempVar, rhs);

printf("%s = %s\n", lhs, tempVar);

}

}

// Main function

int main() {

int n;

printf("Enter the number of expressions: ");

scanf("%d", &n);

getchar(); // Consume newline

char lhs[MAX], rhs[MAX];

printf("Enter expressions in the form (lhs = rhs):\n");

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

printf("Expression %d: ", i + 1);

scanf("%s = %s", lhs, rhs);

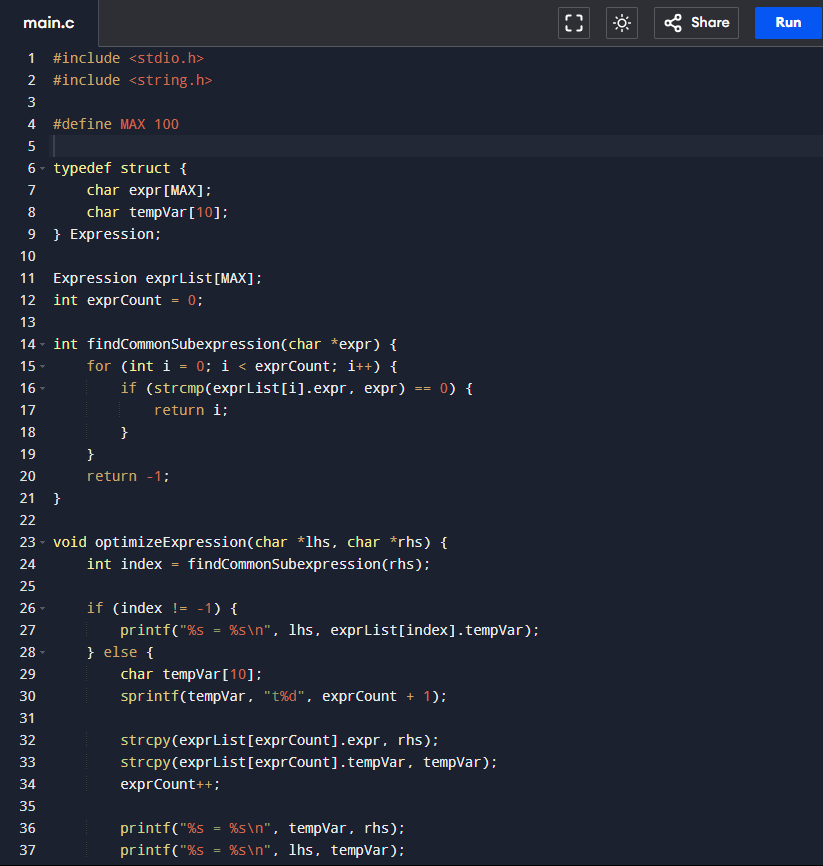
optimizeExpression(lhs, rhs);

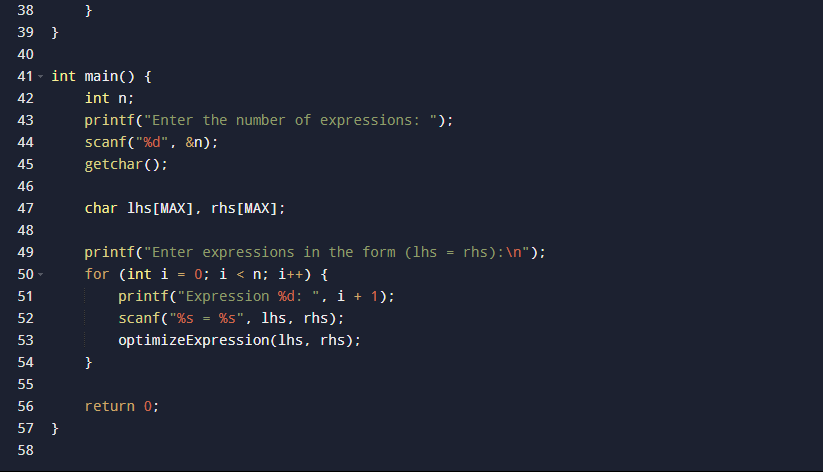
}

return 0;

}

**CODE**:





**OUPUT**:

