**EX 1 : Develop a lexical Analyzer to identify identifiers, constants, operators using C program.**

**Code**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <stdbool.h>

#define MAX\_IDENTIFIER\_LENGTH 31

// Function to check if a character is an operator

bool isOperator(char ch) {

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

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

if (ch == operators[i]) {

return true;

}

}

return false;

}

// Function to check if a string is a keyword

bool isKeyword(const char \*str) {

const char \*keywords[] = {

"int", "float", "char", "if", "else", "for", "while", "return", "void", "main"

};

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

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

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

return true;

}

}

return false;

}

// Function to recognize identifiers and constants

void lexicalAnalysis(const char \*input) {

int i = 0;

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

// Ignore spaces, tabs, and newlines

if (isspace(input[i])) {

i++;

continue;

}

// Ignore comments (single-line // and multi-line /\* \*/)

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

while (input[i] != '\0' && input[i] != '\n') i++;

continue;

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

i += 2;

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

i += 2;

continue;

}

// Check for identifiers and keywords

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

char buffer[MAX\_IDENTIFIER\_LENGTH + 1] = {0};

int j = 0;

while ((isalnum(input[i]) || input[i] == '\_') && j < MAX\_IDENTIFIER\_LENGTH) {

buffer[j++] = input[i++];

}

buffer[j] = '\0';

if (isKeyword(buffer)) {

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

} else {

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

}

continue;

}

// Check for numeric constants

if (isdigit(input[i])) {

char buffer[32] = {0};

int j = 0;

while (isdigit(input[i])) {

buffer[j++] = input[i++];

}

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

continue;

}

// Check for operators

if (isOperator(input[i])) {

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

i++;

continue;

}

// Skip unrecognized characters

i++;

}

}

int main() {

char input[1024];

printf("Enter the code snippet: \n");

fgets(input, sizeof(input), stdin);

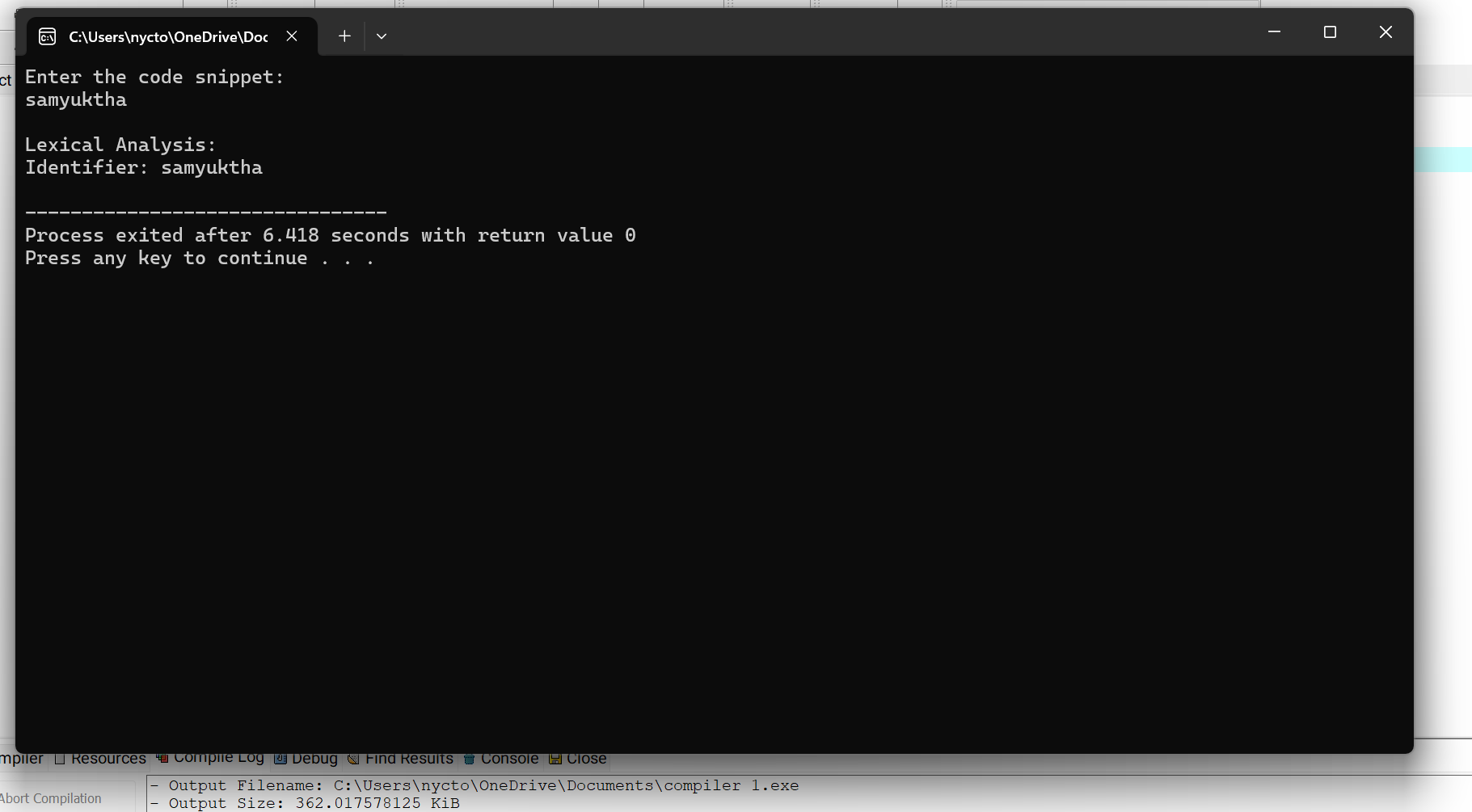
printf("\nLexical Analysis:\n");

lexicalAnalysis(input);

return 0;

}

**OUTPUT**



**EX 2:** **Develop a lexical Analyzer to identify whether a given line is a comment or not.**

CODE

def check\_comment(line: str):

line = line.strip()

if line.startswith("//"):

print("Single-line comment detected.")

elif line.startswith("/\*") and line.endswith("\*/"):

print("Multi-line comment detected.")

else:

print("Not a comment.")

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter a line of code (or comment):")

input\_line = input() # Take user input

check\_comment(input\_line)

**EX 3:** **Design a lexical Analyzer to validate operators to recognize the operators +,-,\*,/ using regular Arithmetic operators**

**CODE**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <ctype.h>**

**#define MAX\_LEN 100**

**void lexicalAnalyzer(const char \*input) {**

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

**if (isspace(input[i])) {**

**continue;**

**}**

**if (input[i] == '+' || input[i] == '-' || input[i] == '\*' || input[i] == '/') {**

**printf("Operator found: %c\n", input[i]);**

**} else {**

**printf("Invalid character: %c\n", input[i]);**

**}**

**}**

**}**

**int main() {**

**char input[MAX\_LEN];**

**printf("Enter an expression: ");**

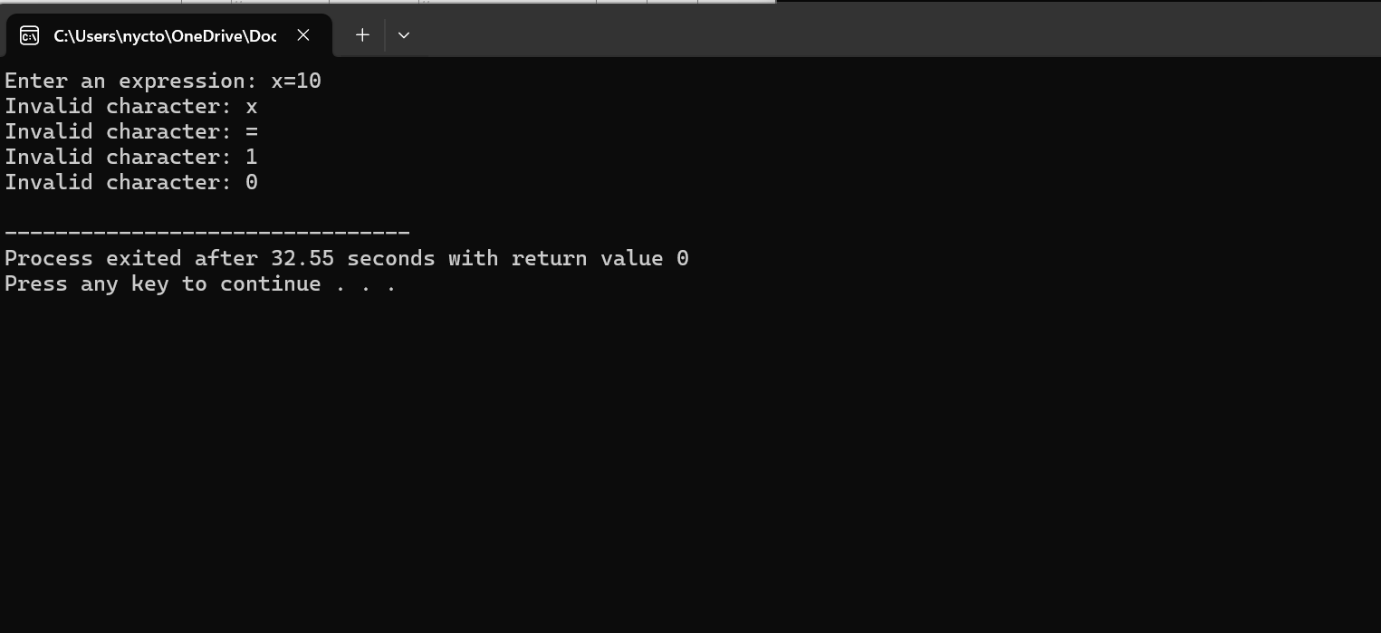
**fgets(input, MAX\_LEN, stdin);**

**lexicalAnalyzer(input);**

**return 0;**

**}**

**OUTPUT:**

****

**EX 4:** **Design a lexical Analyzer to find the number of whitespaces and newline characters**

**CODE:**

#include <stdio.h>

#include <ctype.h>

int main() {

FILE \*file;

char ch;

int whitespace\_count = 0;

int newline\_count = 0;

file = fopen("input.txt", "r");

if (file == NULL) {

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

return 1;

}

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

if (isspace(ch)) {

whitespace\_count++;

if (ch == '\n') {

newline\_count++;

}

}

}

fclose(file);

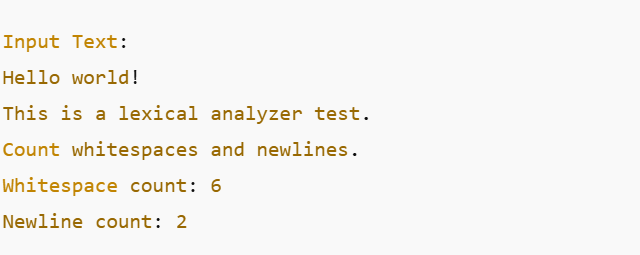
printf("Number of whitespace characters: %d\n", whitespace\_count);

printf("Number of newline characters: %d\n", newline\_count);

return 0;

}

**OUTPUT:**



**EX 5:** **Develop a lexical Analyzer to test whether a given identifier is valid or not.**

**CODE:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX\_IDENTIFIER\_LENGTH 100

int isValidIdentifier(const char \*identifier) {

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

return 0;

}

for (int i = 1; i < strlen(identifier); i++) {

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

return 0;

}

}

return 1;

}

int main() {

char identifier[MAX\_IDENTIFIER\_LENGTH];

printf("Enter an identifier: ");

scanf("%s", identifier);

if (isValidIdentifier(identifier)) {

printf("'%s' is a valid identifier.\n", identifier);

} else {

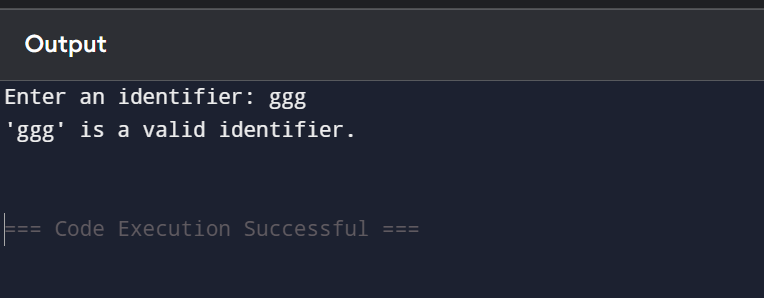
printf("'%s' is not a valid identifier.\n", identifier);

}

return 0;

}

**OUTPUT**

****

**EX 6:** **Implement a C program to eliminate left recursion****.**

**CODE:**

#include <stdio.h>

#include <string.h>

#define MAX 100

void eliminateLeftRecursion(char nonTerminal, char \*production) {

char alpha[MAX], beta[MAX];

int i = 0, j = 0, k = 0;

if (production[0] == nonTerminal) {

i = 1;

while (production[i] != '|' && production[i] != '\0') {

alpha[j++] = production[i++];

}

alpha[j] = '\0';

i++;

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

beta[k++] = production[i++];

}

beta[k] = '\0';

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

printf("%c -> %s%c'\n", nonTerminal, beta, nonTerminal + 1);

printf("%c' -> %s%c' | ε\n", nonTerminal + 1, alpha, nonTerminal + 1);

} else {

printf("The production does not have left recursion.\n");

}

}

int main() {

char nonTerminal;

char production[MAX];

printf("Enter the non-terminal (e.g., A): ");

scanf(" %c", &nonTerminal);

printf("Enter the production (format: Aalpha|beta): ");

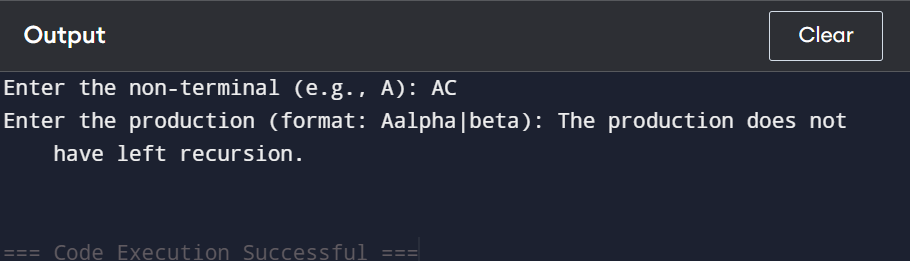
scanf(" %s", production);

eliminateLeftRecursion(nonTerminal, production);

return 0;

}

**OUTPUT:**



EX 7: Implement a C program to eliminate left factoring.

Code:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX 10

void leftFactor(char nonTerminal, char \*productions[], int count) {

char prefix[MAX], newSymbol, newProduction[MAX][MAX];

int i, j, k, prefixLength = 0, flag = 1;

for (i = 0; i < strlen(productions[0]); i++) {

char ch = productions[0][i];

for (j = 1; j < count; j++) {

if (productions[j][i] != ch) {

flag = 0;

break;

}

}

if (!flag) break;

Code:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX 10

void leftFactor(char nonTerminal, char \*productions[], int count) {

char prefix[MAX], newSymbol, newProduction[MAX][MAX];

int i, j, k, prefixLength = 0, flag = 1;

for (i = 0; i < strlen(productions[0]); i++) {

char ch = productions[0][i];

for (j = 1; j < count; j++) {

if (productions[j][i] != ch) {

flag = 0;

break;

}

}

if (!flag) break;

int i, count;

char \*productions[MAX];

printf("Enter the non-terminal: ");

scanf(" %c", &nonTerminal);

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

scanf("%d", &count);

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

productions[i] = (char \*)malloc(MAX \* sizeof(char));

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

scanf("%s", productions[i]);

}

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

leftFactor(nonTerminal, productions, count);

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

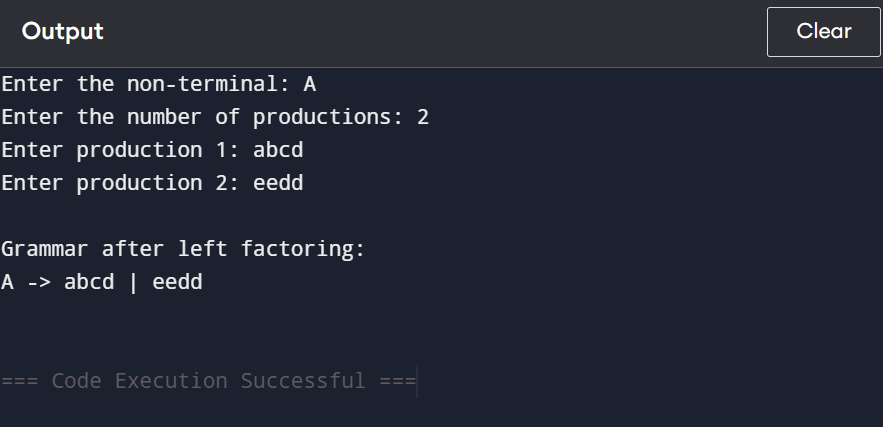
free(productions[i]);

}

return 0;

}

Output:



EX 8: Implement a C program to perform symbol table operations.

Code:

#include <stdio.h>

#include <string.h>

#define MAX 100 // Maximum number of symbols

// Structure to store a symbol

struct Symbol {

char name[50];

char type[20];

int address;

} table[MAX];

int count = 0; // Count of symbols

// Function to insert a symbol

void insert() {

if (count >= MAX) {

printf("Symbol Table is Full!\n");

return;

}

char name[50], type[20];

int address, i;

printf("Enter Symbol Name: ");

scanf("%s", name);

printf("Enter Type: ");

scanf("%s", type);

printf("Enter Address: ");

scanf("%d", &address);

// Check if symbol already exists

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

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

printf("Error: Symbol already exists!\n");

return;

}

}

// Insert the new symbol

strcpy(table[count].name, name);

strcpy(table[count].type, type);

table[count].address = address;

count++;

printf("Symbol Inserted Successfully!\n");

}

// Function to search for a symbol

void search() {

char name[50];

printf("Enter Symbol Name to Search: ");

scanf("%s", name);

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

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

printf("Symbol Found: Name: %s, Type: %s, Address: %d\n",

table[i].name, table[i].type, table[i].address);

return;

}

}

printf("Symbol Not Found!\n");

}

// Function to display the symbol table

void display() {

if (count == 0) {

printf("Symbol Table is Empty!\n");

return;

}

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

printf("---------------------------------------------------\n");

printf("Index\tName\t\tType\t\tAddress\n");

printf("---------------------------------------------------\n");

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

printf("%d\t%-10s\t%-10s\t%d\n", i + 1, table[i].name, table[i].type, table[i].address);

}

printf("---------------------------------------------------\n");

}

// Main function

int main() {

int choice;

while (1) {

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

printf("1. Insert Symbol\n");

printf("2. Search Symbol\n");

printf("3. Display Symbol Table\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: insert(); break;

case 2: search(); break;

case 3: display(); break;

case 4: return 0;

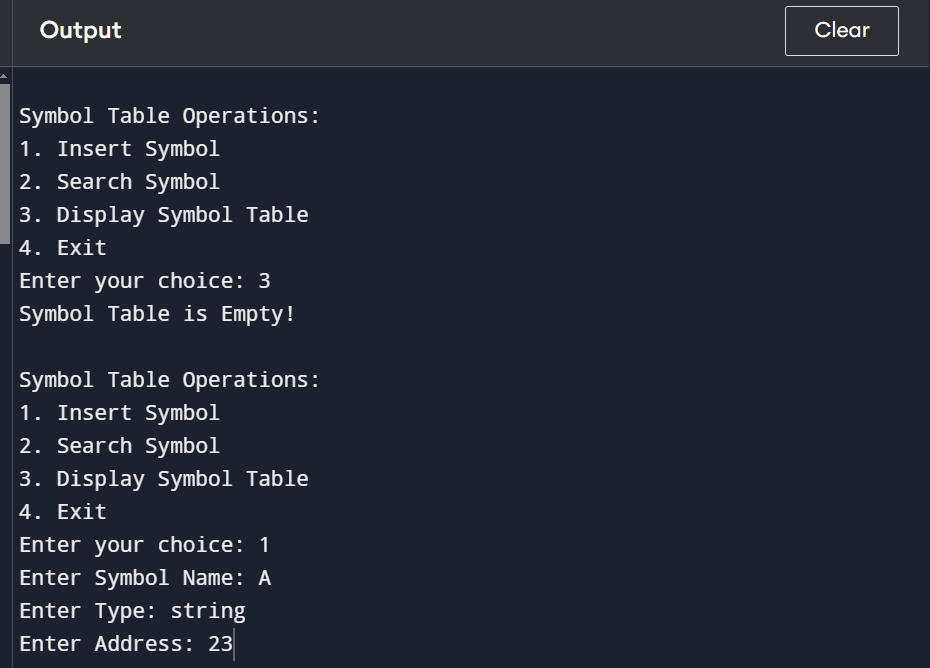
default: printf("Invalid Choice! Try Again.\n");

}

}

}

OUTPUT:



EX 9: Implement a C program that checks whether a given input string follows a predefined grammar.

CODE:

#include <stdio.h>

#include <string.h>

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

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

const char \*objects[] = {"food", "cricket", "book"};

#define SUBJECTS\_SIZE 3

#define VERBS\_SIZE 3

#define OBJECTS\_SIZE 3

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;

}

return 0;

}

void checkGrammar(char \*sentence) {

char \*subject, \*verb, \*object;

subject = strtok(sentence, " ");

verb = strtok(NULL, " ");

object = strtok(NULL, " ");

if (subject == NULL || verb == NULL || object == NULL || strtok(NULL, " ") != NULL) {

printf("Invalid Sentence!\n");

return;

}

if (isInList(subject, subjects, SUBJECTS\_SIZE) &&

isInList(verb, verbs, VERBS\_SIZE) &&

isInList(object, objects, OBJECTS\_SIZE)) {

printf("Valid Sentence!\n");

} else {

printf("Invalid Sentence!\n");

}

}

int main() {

char sentence[100];

printf("Enter a sentence: ");

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

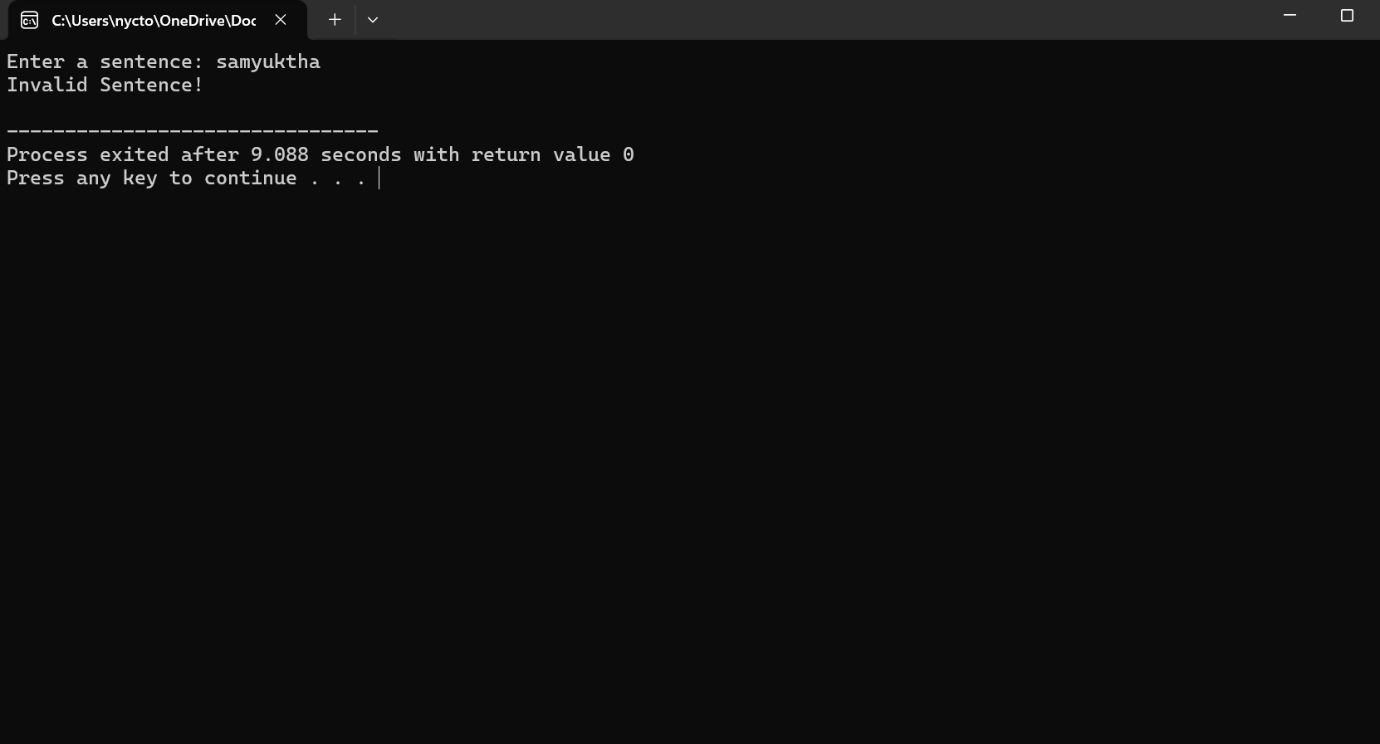
sentence[strcspn(sentence, "\n")] = '\0';

checkGrammar(sentence);

return 0;

}

OUTPUT:



**EX 10:** **Write a C program to construct recursive descent parsing.**

**CODE:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

char input[100]; // Input string

int pos = 0; // Current position in input

// Function prototypes

void E();

void T();

void F();

// Function to match a character and move to the next

void match(char expected) {

if (input[pos] == expected)

pos++;

else {

printf("Error: Unexpected character '%c'\n", input[pos]);

exit(1);

}

}

// Recursive function for E → T + E | T

void E() {

T(); // Parse T

if (input[pos] == '+') { // Check for +

match('+');

E(); // Parse E recursively

}

}

// Recursive function for T → F \* T | F

void T() {

F(); // Parse F

if (input[pos] == '\*') { // Check for \*

match('\*');

T(); // Parse T recursively

}

}

// Recursive function for F → (E) | id

void F() {

if (isalpha(input[pos])) { // If it's an identifier (id)

match(input[pos]);

} else if (input[pos] == '(') { // If it's a left parenthesis

match('(');

E(); // Parse E inside parentheses

match(')'); // Match closing parenthesis

} else {

printf("Error: Invalid character '%c'\n", input[pos]);

exit(1);

}

}

// Main function

int main() {

printf("Enter an expression: ");

scanf("%s", input);

E(); // Start parsing from E

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

printf("Valid Expression!\n");

else

printf("Error: Unexpected characters after parsing!\n");

return 0;

}

OUTPUT:

