1.The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

**Program/**code**:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

void removeComments(char\* source);

void tokenize(char\* source);

int main() {

char sourceCode[] = "/\* Sample code \*/\n"

"int a = 5; // Initialize a\n"

"a = a + 10;\n";

printf("Original Source Code:\n%s\n", sourceCode);

removeComments(sourceCode);

printf("\nSource Code after Removing Comments:\n%s\n", sourceCode);

printf("\nTokens:\n");

tokenize(sourceCode);

return 0;

}

// Function to remove comments (single-line and multi-line)

void removeComments(char\* source) {

int i = 0, j = 0;

while (source[i]) {

// Check for single-line comment

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

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

}

// Check for multi-line comment

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

i += 2;

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

i += 2;

} else {

source[j++] = source[i++];

}

}

source[j] = '\0';

}

// Function to tokenize the source code

void tokenize(char\* source) {

int i = 0;

while (source[i]) {

// Skip whitespace

if (isspace(source[i])) {

i++;

continue;

}

// Identify an identifier (starts with a letter or underscore)

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

char identifier[MAX\_IDENTIFIER\_LENGTH + 1];

int length = 0;

// Collect identifier characters

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

identifier[length++] = source[i++];

}

identifier[length] = '\0';

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

}

// Identify a constant (integer)

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

int constant = 0;

while (isdigit(source[i])) {

constant = constant \* 10 + (source[i] - '0');

i++;

}

printf("Constant: %d\n", constant);

}

// Identify operators (e.g., +, -, =, \*, etc.)

else if (strchr("+-\*/=<>!&|", source[i])) {

char operator[3] = {source[i], '\0', '\0'};

if (strchr("=<>!&|", source[i]) && source[i + 1] == '=') { // Handle ==, <=, etc.

operator[1] = source[++i];

}

printf("Operator: %s\n", operator);

i++;

}

// Handle other characters (e.g., punctuation)

else {

printf("Symbol: %c\n", source[i]);

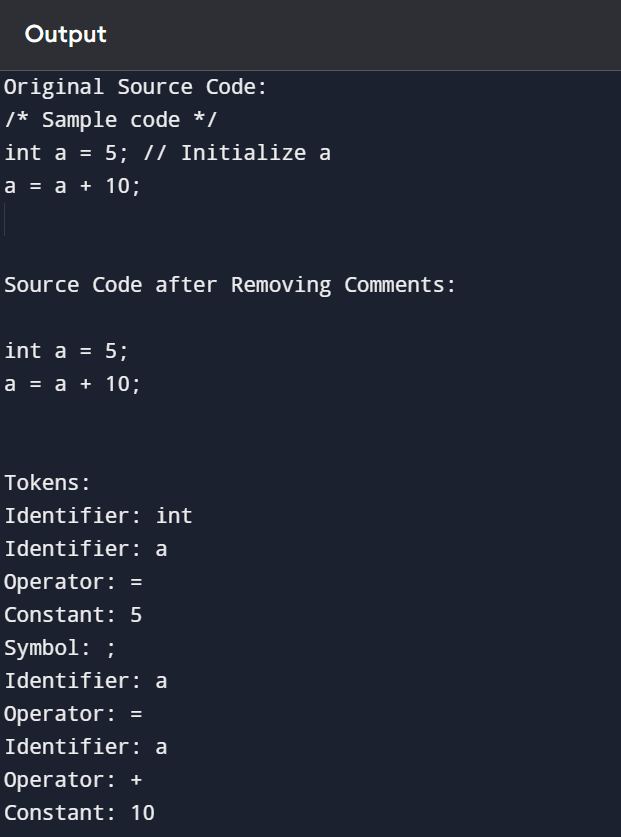
i++;

}

}

}

**Output :**



2.Extend the lexical Analyzer to Check comments, dened as follows in C:

a) A comment begins with // and includes all characters until the end of that line.

b) A comment begins with /\* and includes all characters through the next occurrence of the character sequence \*/Develop a lexical Analyzer to identify whether a given line is a comment or not.

**Program/code:**

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

// Function to check if a given line is a comment

bool isComment(char\* line) {

// Remove leading whitespaces

while (\*line == ' ' || \*line == '\t') line++;

// Check for single-line comment starting with //

if (strncmp(line, "//", 2) == 0) {

return true;

}

// Check for multi-line comment starting with /\*

if (strncmp(line, "/\*", 2) == 0) {

// Look for closing \*/

char\* closing = strstr(line, "\*/");

if (closing) {

return true;

} else {

printf("Warning: Unclosed multi-line comment.\n");

}

}

return false;

}

int main() {

// Example input: an array of lines from the source code

char\* lines[] = {

"int x = 5; // This is a single-line comment",

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

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

"/\* This is an unclosed multi-line comment",

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

NULL // End of the array

};

printf("Checking for comments in the given lines:\n\n");

for (int i = 0; lines[i] != NULL; i++) {

printf("Line %d: %s\n", i + 1, lines[i]);

if (isComment(lines[i])) {

printf(" -> This line is a comment.\n");

} else {

printf(" -> This line is NOT a comment.\n");

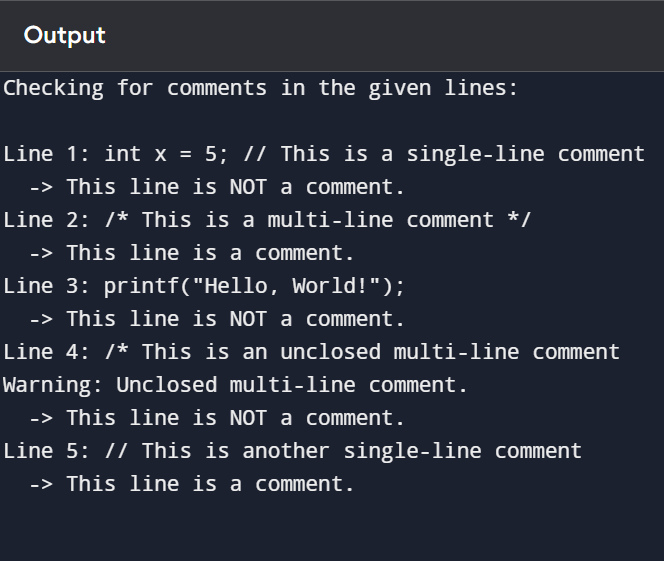
}

}

return 0;

}

**Output:**

****

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

#include <stdio.h>

#include <string.h>

#include <ctype.h>

// Function to check if a character is a valid arithmetic operator

int isArithmeticOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/');

}

// Function to scan the input and recognize arithmetic operators

void recognizeOperators(const char\* source) {

int i = 0;

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

if (isspace(source[i])) {

i++;

continue;

}

if (isArithmeticOperator(source[i])) {

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

}

i++;

}

}

int main() {

char sourceCode[] = "a = b + c \* d - e / f;";

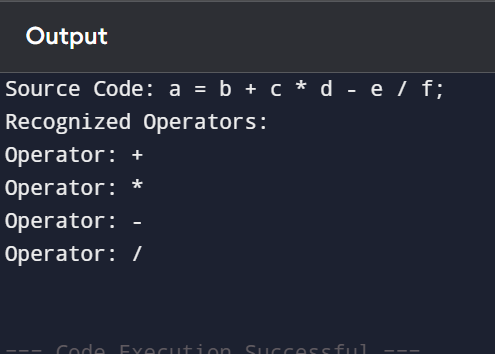
printf("Source Code: %s\n", sourceCode);

printf("Recognized Operators:\n");

recognizeOperators(sourceCode);

return 0;

}



4.Design a lexical Analyzer to find the number of whitespaces and newline characters.

#include <stdio.h>

void countWhitespaceAndNewlines(const char\* source) {

int i = 0, whitespaceCount = 0, newlineCount = 0;

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

if (source[i] == ' ' || source[i] == '\t') {

whitespaceCount++;

} else if (source[i] == '\n') {

newlineCount++;

}

i++;

}

printf("Total Whitespace Characters: %d\n", whitespaceCount);

printf("Total Newline Characters: %d\n", newlineCount);

}

int main() {

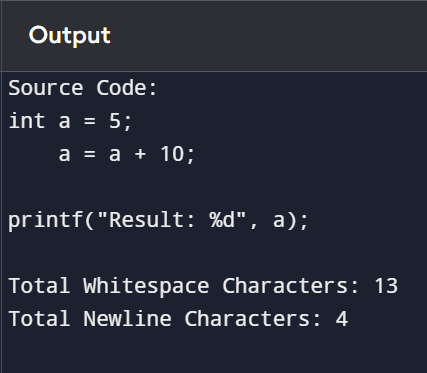
char sourceCode[] = "int a = 5;\n a = a + 10;\n\nprintf(\"Result: %d\", a);\n";

printf("Source Code:\n%s\n", sourceCode);

countWhitespaceAndNewlines(sourceCode);

return 0;

}



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

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <stdbool.h>

bool isKeyword(const char\* str) {

const char\* keywords[] = {

"auto", "break", "case", "char", "const", "continue", "default", "do", "double",

"else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",

"return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",

"union", "unsigned", "void", "volatile", "while", NULL

};

for (int i = 0; keywords[i] != NULL; i++) {

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

return true;

}

}

return false;

}

// Function to check if a given identifier is valid

bool isValidIdentifier(const char\* identifier) {

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

return false; // Must start with a letter or underscore

}

for (int i = 1; identifier[i] != '\0'; i++) {

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

return false; // Must contain only letters, digits, or underscores

}

}

if (isKeyword(identifier)) {

return false; // Must not be a keyword

}

return true;

}

int main() {

char identifiers[][20] = {"\_valid123", "123invalid", "for", "variable\_name", "void", "test\_var", "a1b2"};

int numIdentifiers = sizeof(identifiers) / sizeof(identifiers[0]);

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

printf("Identifier: %s -> ", identifiers[i]);

if (isValidIdentifier(identifiers[i])) {

printf("Valid\n");

} else {

printf("Invalid\n");

}

}

return 0;

}

