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.

Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_IDENTIFIER\_LENGTH 100

#define MAX\_CONSTANT\_LENGTH 100

void ignoreWhitespace(FILE \*fp) {

char ch;

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

if (!isspace(ch)) {

ungetc(ch, fp);

break;

}

}

}

void identifyToken(FILE \*fp) {

char identifier[MAX\_IDENTIFIER\_LENGTH];

char constant[MAX\_CONSTANT\_LENGTH];

char ch;

int i = 0;

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

if (isalpha(ch)) {

i = 0;

do {

if (i < MAX\_IDENTIFIER\_LENGTH - 1) {

identifier[i++] = ch;

}

ch = fgetc(fp);

} while (isalnum(ch) || ch == '\_');

identifier[i] = '\0';

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

ungetc(ch, fp);

} else if (isdigit(ch)) {

i = 0;

do {

if (i < MAX\_CONSTANT\_LENGTH - 1) {

constant[i++] = ch;

}

ch = fgetc(fp);

} while (isdigit(ch));

constant[i] = '\0';

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

ungetc(ch, fp);

} else if (ch == '/' && (ch = fgetc(fp)) == '/') {

while ((ch = fgetc(fp)) != EOF && ch != '\n');

} else if (ch == '/' && ch == '\*') {

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

if (ch == '\*' && (ch = fgetc(fp)) == '/') {

break;

}

}

} else if (ch == '+' || ch == '-' || ch == '\*' || ch == '/') {

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

}

}

}

int main() {

FILE \*fp = fopen("input.txt", "r");

if (fp == NULL) {

fprintf(stderr, "Could not open file\n");

return 1;

}

ignoreWhitespace(fp);

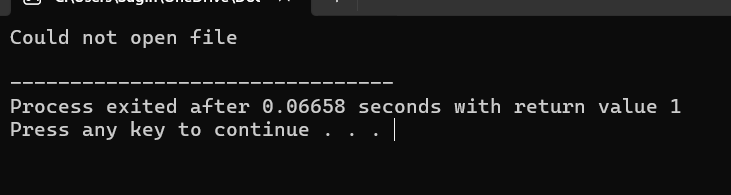
identifyToken(fp);

fclose(fp);

return 0;

}

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.

Code:

#include <stdio.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 1024

int is\_comment(const char \*line) {

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

return 1;

}

if (strstr(line, "/\*") != NULL && strstr(line, "\*/") != NULL) {

return 1;

}

return 0;

}

int main() {

char line[MAX\_LINE\_LENGTH];

printf("Enter a line of code: ");

fgets(line, sizeof(line), stdin);

if (is\_comment(line)) {

printf("The line is a comment.\n");

} else {

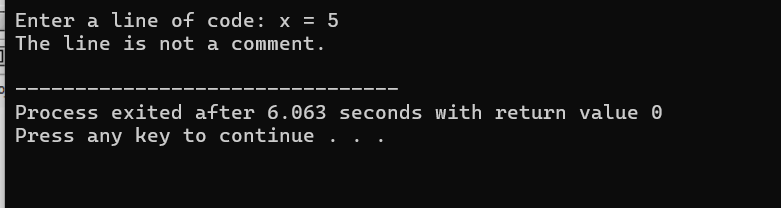
printf("The line is not a comment.\n");

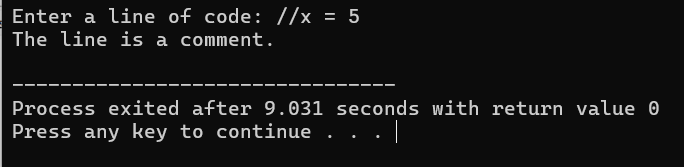
}

return 0;

}

Output:





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

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

} else {

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

}

}

}

int main() {

char input[MAX\_LEN];

printf("Enter an arithmetic expression: ");

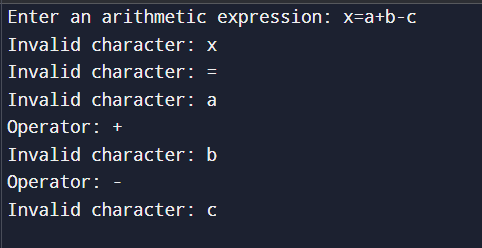
fgets(input, MAX\_LEN, stdin);

lexicalAnalyzer(input);

return 0;

}

Output:



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

Code:

#include <stdio.h>

#include <ctype.h>

#include <string.h> // Added this line

// Function to count whitespaces and newline characters

void count\_whitespace\_newline(char \*input) {

int whitespace\_count = 0;

int newline\_count = 0;

// Iterate over each character in the input

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

// Check if the character is a whitespace

if (isspace(input[i])) {

// Check if the character is a newline

if (input[i] == '\n') {

newline\_count++;

} else {

whitespace\_count++;

}

}

}

// Print the counts

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

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

}

int main() {

char input[1000];

printf("Enter a string: ");

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

// Remove the trailing newline character added by fgets

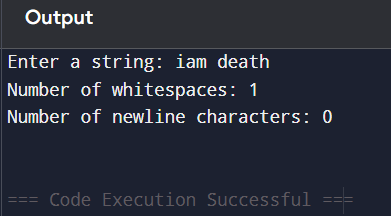
input[strcspn(input, "\n")] = 0;

count\_whitespace\_newline(input);

return 0;

}

Output:



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>

int isValidIdentifier(const char \*identifier) {

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

return 0;

}

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

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

return 0;

}

}

return 1;

}

int main() {

char identifier[100];

printf("Enter an identifier: ");

scanf("%s", identifier);

if (isValidIdentifier(identifier)) {

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

} else {

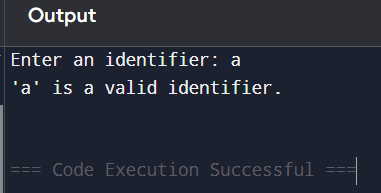
printf("'%s' is not a valid identifier.\n", identifier);

}

return 0;

}

Output:



6.Implement a C program to eliminate left recursion.

Code:

#include <stdio.h>

#include <string.h>

#define MAX 100

// Structure to represent a production rule

typedef struct {

char lhs[MAX];

char rhs[MAX];

} Production;

// Function to eliminate left recursion

void eliminate\_left\_recursion(Production productions[], int n) {

// Iterate over each production rule

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

// Check if the production rule is left recursive

if (productions[i].rhs[0] == productions[i].lhs[0]) {

// Eliminate left recursion

printf("Left recursion found in production rule: %s -> %s\n", productions[i].lhs, productions[i].rhs);

printf("Eliminating left recursion...\n");

// Find the non-recursive part of the production rule

int j = 1;

while (j < strlen(productions[i].rhs) && productions[i].rhs[j] == productions[i].lhs[0]) {

j++;

}

// Create a new production rule without left recursion

char new\_rhs[MAX];

strcpy(new\_rhs, productions[i].rhs + j);

// Add the new production rule

printf("New production rule: %s' -> %s\n", productions[i].lhs, new\_rhs);

// Update the original production rule

productions[i].rhs[j] = '\0';

printf("Updated production rule: %s -> %s%s'\n", productions[i].lhs, productions[i].rhs, productions[i].lhs);

}

}

}

int main() {

// Define the production rules

Production productions[] = {

{"E", "E+T"},

{"E", "T"},

{"T", "T\*F"},

{"T", "F"},

{"F", "(E)"},

{"F", "id"}

};

int n = sizeof(productions) / sizeof(productions[0]);

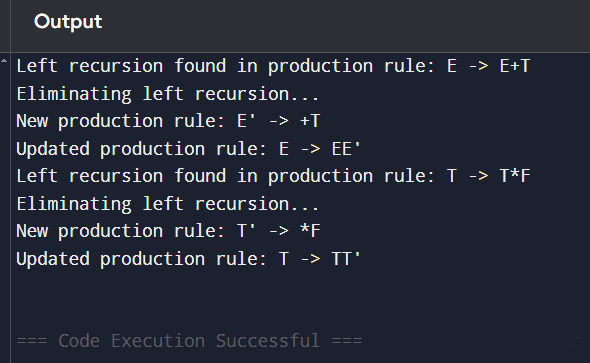
// Eliminate left recursion

eliminate\_left\_recursion(productions, n);

return 0;

}

Output:



7.Implement a C program to eliminate left factoring.

Code:

#include <stdio.h>

#include <string.h>

#define MAX 100

// Structure to represent a production rule

typedef struct {

char lhs[MAX];

char rhs[MAX];

} Production;

// Function to eliminate left factoring

void eliminate\_left\_factoring(Production productions[], int n) {

// Iterate over each production rule

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

// Check if the production rule has left factoring

for (int j = i + 1; j < n; j++) {

if (strcmp(productions[i].lhs, productions[j].lhs) == 0) {

// Find the common prefix of the two production rules

int k = 0;

while (k < strlen(productions[i].rhs) && k < strlen(productions[j].rhs) && productions[i].rhs[k] == productions[j].rhs[k]) {

k++;

}

// Check if the common prefix is not empty

if (k > 0) {

// Eliminate left factoring

printf("Left factoring found in production rules: %s -> %s and %s -> %s\n", productions[i].lhs, productions[i].rhs, productions[j].lhs, productions[j].rhs);

printf("Eliminating left factoring...\n");

// Create a new production rule with the common prefix

char new\_lhs[MAX];

strcpy(new\_lhs, productions[i].lhs);

char new\_rhs[MAX];

strncpy(new\_rhs, productions[i].rhs, k);

new\_rhs[k] = '\0';

// Create new production rules without the common prefix

char new\_rhs1[MAX];

strcpy(new\_rhs1, productions[i].rhs + k);

char new\_rhs2[MAX];

strcpy(new\_rhs2, productions[j].rhs + k);

// Add the new production rules

printf("New production rule: %s -> %s%s'\n", new\_lhs, new\_rhs, new\_lhs);

printf("New production rule: %s' -> %s | %s\n", new\_lhs, new\_rhs1, new\_rhs2);

// Update the original production rules

productions[i].rhs[0] = '\0';

productions[j].rhs[0] = '\0';

}

}

}

}

}

int main() {

// Define the production rules

Production productions[] = {

{"E", "T+T"},

{"E", "T-T"},

{"T", "F\*F"},

{"T", "F/F"},

{"F", "(E)"},

{"F", "id"}

};

int n = sizeof(productions) / sizeof(productions[0]);

// Eliminate left factoring

eliminate\_left\_factoring(productions, n);

return 0;

}

Output:

