**EGE UNIVERSITY**

**FACULTY OF ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

**PROGRAMMING LANGUAGE**

**2023–2024**

**PROJECT-1 REPORT**

**LEXICAL ANALYSIS**

**DELIVERY DATE**

06.05.2024

**PREPARED BY**

05210000220, Bengi İlhan

05220000276, Mahmut Karabulut

**Contents:**

LEXICAL ANALYSIS

1.a Overview

1.a.1 Explanation

1.b Token Types

1.b.1 Token Enumeration

1.b.2 Token Categories

1.b.3 Keywords and Operators

1.c Comment Stripping

1.c.1 Code

1.c.2 Explanation

1.d Token Handling Functions

1.d.1 Code

1.d.2 Output Examples

1.e Line Processing

1.e.1 Code

1.e.2 Explanation

1.f Main Lexical Analysis Function

1.f.1 Code

1.f.2 Overview

1.f.3 Explanation

1.g Conclusion

1.g.1 Summary

1.g.2 Suggestions for Improvement

TEST CASES

1. An example, this not contains error
2. An example, this not contains error
3. Error test, identifiers
4. Error test, integers
5. Error test, strings
6. Error test, all of error case (comments, identifiers, integers, strings)
7. Error test, comments (all of comment cases)

## Overview

The provided C code implements a lexical analyzer that reads an input file, identifies various tokens (such as keywords, operators, identifiers, integers, strings, and others), and writes the identified tokens to an output file. The code follows a typical lexer structure with predefined token types, handling different token categories, and error reporting for invalid tokens.

## Token Types

The code defines several token types using an enumeration (TokenType):

* TOKEN\_IDENTIFIER: Represents an identifier in the code.
* TOKEN\_KEYWORD: Represents a reserved keyword.
* TOKEN\_OPERATOR: Represents mathematical operators.
* TOKEN\_STRING: Represents a string literal.
* TOKEN\_INTEGER: Represents an integer literal.
* TOKEN\_END\_OF\_LINE: Represents an end-of-line token (marked by a period .).
* TOKEN\_COMMA: Represents a comma ,.
* TOKEN\_LEFT\_CURLY\_BRACKET and TOKEN\_RIGHT\_CURLY\_BRACKET: Represent curly brackets {}.
* TOKEN\_ERROR: Represents an error during tokenization.

These types are used to categorize tokens during the lexical analysis.

## Global Definitions

The code contains global definitions for keywords and operators:

* keywords[]: An array of strings representing reserved keywords such as "int", "text", "is", "loop", "times", "read", "write", and "newLine".
* operators[]: A string of characters representing mathematical operators ('+', '-', '\*', '/').

The code provides utility functions to check whether a given character is an operator and whether a given identifier is a keyword.

## Comment Stripping

Before processing a line, the code removes comments with the strip\_comments function. It looks for block comments starting with /\* and ending with \*/. If a comment is unclosed, it returns NULL, indicating an error:

char\* strip\_comments(char\* line) {

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

while (comment\_start) {

char\* comment\_end = strstr(comment\_start, "\*/");

if (!comment\_end) return NULL; // Unclosed comment

memmove(comment\_start, comment\_end + 2, strlen(comment\_end + 2) + 1); // Remove the comment

comment\_start = strstr(comment\_start, "/\*");

}

return line;

}

## Token Handling Functions

The code contains specific functions to handle different token types and write them to the output file. For example:

* handle\_operator: Writes an operator token to the output.
* handle\_end\_of\_line: Writes an end-of-line token to the output.
* handle\_comma: Writes a comma token to the output.
* handle\_curly\_bracket: Handles left or right curly brackets.
* handle\_identifier: Handles identifiers, checking if they are keywords or valid identifiers.
* handle\_integer: Handles integer tokens.
* handle\_string: Handles string tokens, checking for unclosed or invalid strings.

Each function writes the token information to the output file in a structured format. For example, here's the handle\_operator function:

void handle\_operator(FILE\* outputFile, char op) {

fprintf(outputFile, "Operator(%c)\n", op);

}

Similarly, the handle\_identifier function checks if the identifier is a keyword or valid identifier, then writes the appropriate output:

void handle\_identifier(FILE\* outputFile, const char\* start, int length) {

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

strncpy(identifier, start, length);

identifier[length] = '\0';

if (!isalpha(identifier[0])) {

fprintf(outputFile, "Error: Invalid identifier.\n");

} else if (is\_keyword(identifier)) {

fprintf(outputFile, "Keyword(%s)\n", identifier);

} else {

fprintf(outputFile, "Identifier(%s)\n", identifier);

}

}

## Line Processing

The process\_line function processes a line of text, identifying various tokens such as operators, keywords, identifiers, integers, strings, and special characters (like curly brackets). It writes the identified tokens to the provided output file stream. Here's how the function works, broken down into key parts:

## Initial Setup

* Iteration Variables: i = 0 starts at the beginning of the line, and len = strlen(line) represents the length of the line.
* The main loop continues as long as i < len.

## Skip Whitespace

* A nested loop skips whitespace characters. It increments i until it finds a non-whitespace character.

## Operators

* If line[i] is an operator (checked with is\_operator(line[i])), it calls handle\_operator(outputFile, line[i]) and then increments i.

## Special Tokens

A switch-case block handles several predefined tokens:

* End-of-Line: If line[i] is a period '.', it calls handle\_end\_of\_line(outputFile).
* Comma: If line[i] is a comma ',', it calls handle\_comma(outputFile).

Curly Brackets: If line[i] is '{' or '}', it calls handle\_curly\_bracket(outputFile, line[i]).

* In each case, the index i is incremented after handling the token.

## Identifiers and Keywords

* If line[i] is alphabetic, it could be an identifier or keyword. The function:
  + Tracks the start of the identifier (start = i) and the length (length = 0).
  + Uses a while loop to include alphanumeric characters and underscores in the identifier.
  + If length exceeds MAX\_IDENTIFIER\_LENGTH, it logs an error ("Error: Identifier too long.") and skips to the end of the identifier.
  + If within limits, it calls handle\_identifier(outputFile, line + start, length) to identify the keyword or identifier.

## Integer Constants

* If line[i] is numeric (isdigit(line[i])), it may represent an integer constant. The function:
  + Records the start (start = i) and length (length = 0).
  + Uses a while loop to capture the full number.
  + If length exceeds MAX\_INTEGER\_LENGTH, it logs an error ("Error: Integer too long.") and skips to the end of the number.
  + If valid, it calls handle\_integer(outputFile, line + start, length).

## String Constants

* If line[i] is a double quote ('"'), it starts a string literal. The function:
  + Initializes start = i and increments i.
  + Uses a while loop to find the closing double quote, tracking the length of the string.
  + If length exceeds MAX\_STRING\_LENGTH, it logs an error ("Error: String too long.") and skips to the end of the string.
  + If the string is closed properly, it calls handle\_string(outputFile, line + start, length). If unclosed, it logs "Error: Unclosed string.".

## Unrecognized Tokens

If none of the above conditions are met, the function logs "Error: Unrecognized token." and increments i.

// Processes a line of code, identifying and classifying tokens

void process\_line(const char\* line, FILE\* outputFile,int\* is\_comment\_open) {

int i = 0;

int len = strlen(line);

if (\*is\_comment\_open) { // using flag for comments

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

if (comment\_end) {

\*is\_comment\_open = 0; // Comment closed

i = comment\_end - line + 2; // Continue processing

} else {

return; // Comment is still open

}

}

while (i < len) {

while (isspace(line[i])) i++; // Skip whitespace

if (line[i] == '\0') continue; // If at the end of the line

// Handle potential negative integers

if (line[i] == '-' && isdigit(line[i + 1])) {

fprintf(outputFile, "Error: Negative integer.\n");

i++;

while (isdigit(line[i])) i++; // Skip the rest of the integer

continue;

}

// Handle operator tokens

if (is\_operator(line[i])) {

handle\_operator(outputFile, line[i]);

i++;

continue;

}

// Handle various other token types

switch (line[i]) {

case '.': // End-of-line token

handle\_end\_of\_line(outputFile);

i++;

break;

case ',': // Comma token

handle\_comma(outputFile);

i++;

break;

case '{': // Left curly bracket token

case '}': // Right curly bracket token

handle\_curly\_bracket(outputFile, line[i]);

i++;

break;

default:

// Handle identifiers and keywords

if (isalpha(line[i])) {

int start = i;

int length = 0;

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

length++;

if (length > MAX\_IDENTIFIER\_LENGTH) {

fprintf(outputFile, "Error: Identifier too long.\n");

while (isalnum(line[i]) || line[i] == '\_') { // Skip the long identifier

i++;

}

break;

}

i++;

}

if (length <= MAX\_IDENTIFIER\_LENGTH) {

handle\_identifier(outputFile, line + start, length);

}

continue;

}

// Handle integer tokens

if (isdigit(line[i])) {

int start = i;

int length = 0;

while (isdigit(line[i])) {

length++;

if (length > MAX\_INTEGER\_LENGTH) {

fprintf(outputFile, "Error: Integer too long.\n");

while (isdigit(line[i])) i++; // Skip the long integer

break;

}

i++;

}

if (length <= MAX\_INTEGER\_LENGTH) {

handle\_integer(outputFile, line + start, length);

}

continue;

}

// Handle string tokens, check for unclosed or invalid strings

if (line[i] == '"') {

int start = i;

int length = 1;

i++;

int string\_too\_long = 0;

while (line[i] != '"' && i < len) {

length++;

if (length > MAX\_STRING\_LENGTH) {

string\_too\_long = 1;

break;

}

i++;

}

if (string\_too\_long) {

fprintf(outputFile, "Error: String too long.\n");

while (line[i] != '\0' && line[i] != '"') i++; // Skip until end of the string or line

if (line[i] == '"') i++; // Skip the closing quote if present

continue;

}

if (line[i] == '"') {

length++; // Include the closing quote

// Check if the string contains a double quote inside it

int isInvalidString = 0;

handle\_string(outputFile, line + start, length); // Handle the valid string

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

if (line[j] == '"') {

isInvalidString = 1;

break;

}

i = j+2; // Move past the string

}

if (isInvalidString) {

fprintf(outputFile, "Error: String contains double quotes.\n");

continue;

}

} else {

fprintf(outputFile, "Error: Unclosed string.\n"); // If the string isn't closed

}

continue;

}

fprintf(outputFile, "Error: Unrecognized token.\n"); // If none of the conditions were met

i++;

break;

}

}

}

## Main Lexical Analysis Function

The lexicalAnalyzer function performs lexical analysis on a source code file, identifying various tokens and writing them to an output file. Here's a detailed explanation of how it works:

## Opening Files

* Input File: The function attempts to open the specified input file in read mode (fopen(inputFilePath, "r")). If the file cannot be opened, it logs an error ("Error: Could not open input file.") to the standard error stream (stderr) and returns to exit the function.
* Output File: It also tries to open the specified output file in write mode (fopen(outputFilePath, "w")). If this fails, the input file is closed to prevent resource leaks, an error message is logged ("Error: Could not open output file."), and the function returns.

## Processing Lines

* Line Buffer: The function declares a character array line[1024] to hold each line from the input file. It uses fgets to read lines from the input file.
* Strip Comments: Before processing the line, the function calls strip\_comments(line). This function removes block comments (i.e., text between /\* and \*/). If the comment is unclosed, strip\_comments returns NULL, prompting the lexical analyzer to log "Error: Unclosed comment." to the output file and skip further processing of that line.
* Process Line: If there are no unclosed comments, the function calls process\_line(line, outputFile), which identifies tokens and writes them to the output file. This is the core of the lexical analysis, responsible for token classification and error handling (see the explanation of process\_line for more details).

## Closing Files

After processing all lines from the input file:

* Close Input File: The function closes the input file to free resources and prevent leaks.
* Close Output File: It also closes the output file to ensure all data is written and to free resources.

Overall, lexicalAnalyzer manages the entire process of opening files, reading and processing lines, handling comments, and ensuring resources are properly managed by closing files after use. The modular approach allows the function to handle errors effectively, ensuring that incomplete comments or file-related issues don't crash the program.

// Main lexical analysis function, reads from an input file and writes tokens to an output file

void lexicalAnalyzer(const char\* inputFilePath, const char\* outputFilePath) {

FILE\* inputFile = fopen(inputFilePath, "r");

if (!inputFile) {

fprintf(stderr, "Error: Could not open input file.\n");

return;

}

FILE\* outputFile = fopen(outputFilePath, "w");

if (!outputFile) {

fclose(inputFile);

fprintf(stderr, "Error: Could not open output file.\n");

return;

}

int is\_comment\_open = 0; // Flag for comments

char line[1024];

while (fgets(line, sizeof(line), inputFile)) {

if (!strip\_comments(line, &is\_comment\_open)) { // If the comment is not closed, do not continue

continue; //If the comment is not closed, check the next line

}

// If comment is closed, commit current line

if (!is\_comment\_open) {

process\_line(line, outputFile, &is\_comment\_open);

}

}

// If there is still an open comment at the end of the file, print the error message

if (is\_comment\_open) {

fprintf(outputFile, "Error: Unclosed comment.\n");

}

fclose(inputFile);

fclose(outputFile);

}

This function ensures that all lines are processed, and comments are properly handled before identifying tokens.

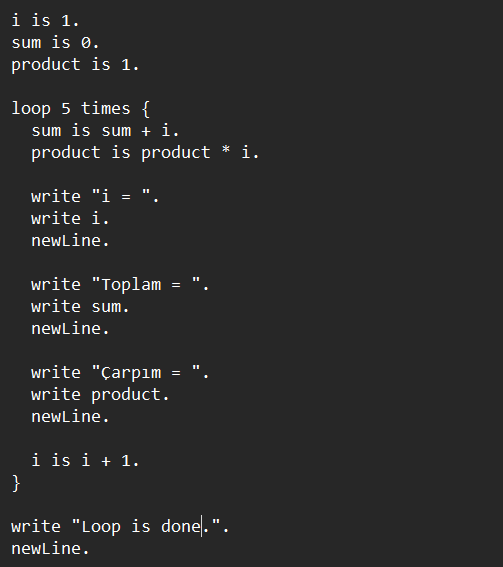
## Conclusion

Overall, this lexical analyzer code provides a robust way to identify and classify tokens from a source file, handling various token types and potential errors. The modular structure of the code allows for easy modification and expansion if additional token types or functionality are required.

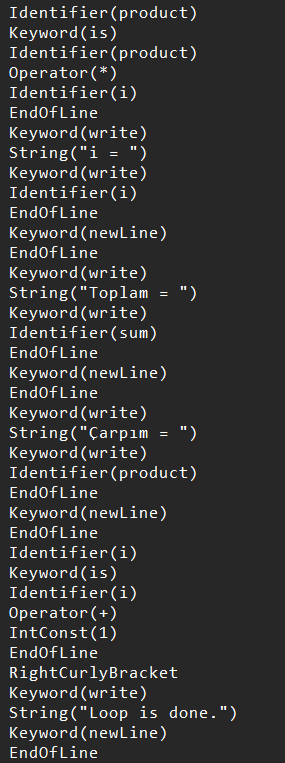
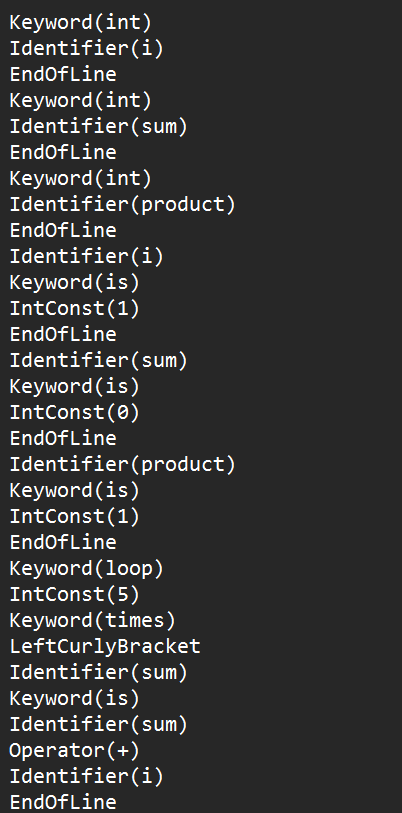
**Test Cases**

**1. (An example, there aren’t errors)**

**Input:**

****

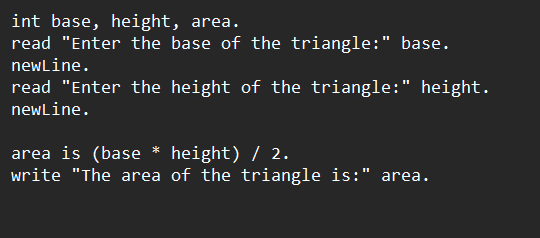
**Output:**

****

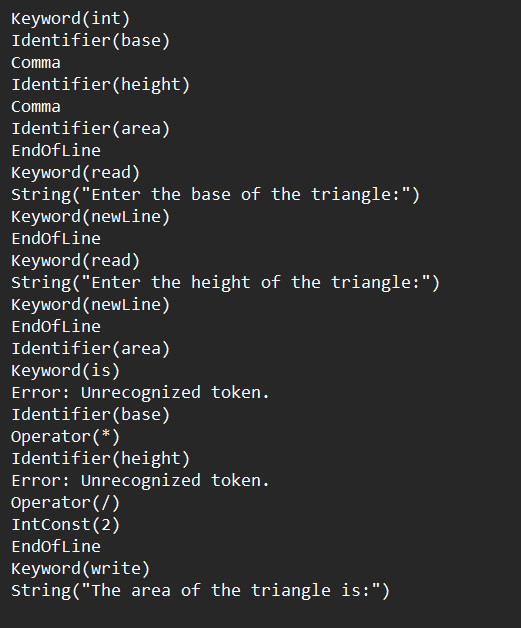
A test case that does not contain errors and contains a loop

**2. (An example, there aren’t errors)**

**Input:**

****

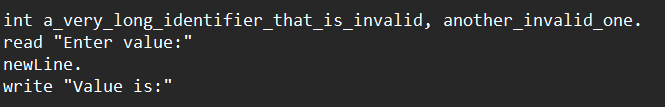
**Output:**

****

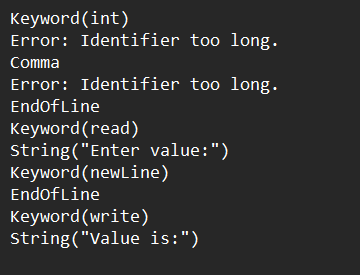
Again, a test case that does not contain any errors and calculates the area of the triangle.

**3. (Identifier Cases)**

**Input:**

****

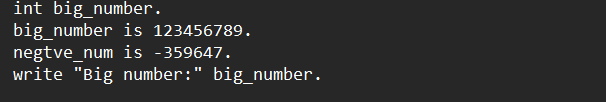
**Output:**

****

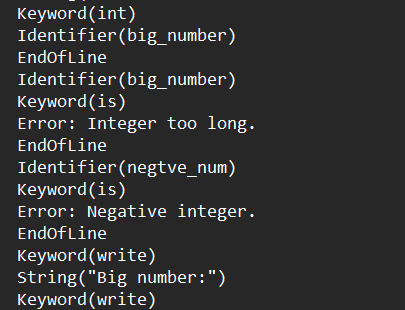
This is a test case that checks the Identifier.

**4. (Integer Cases)**

**Input:**

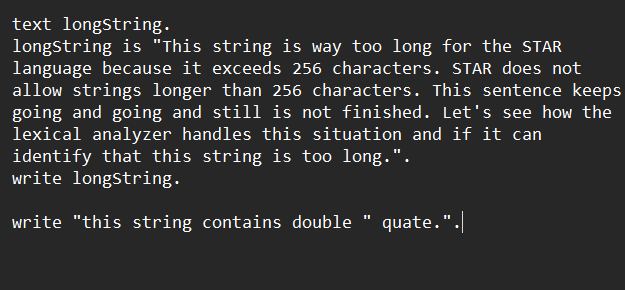
****

**Output:**

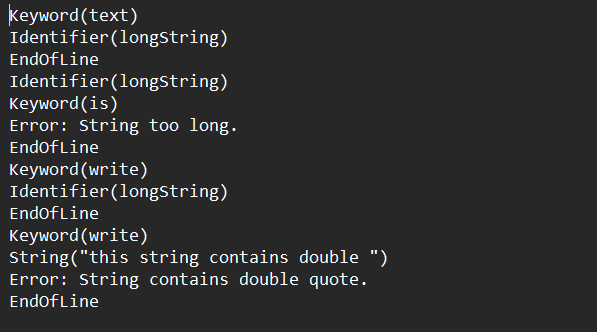
****

This is a test case that performs integer checks.

**5. (String Cases)**

**Input:**

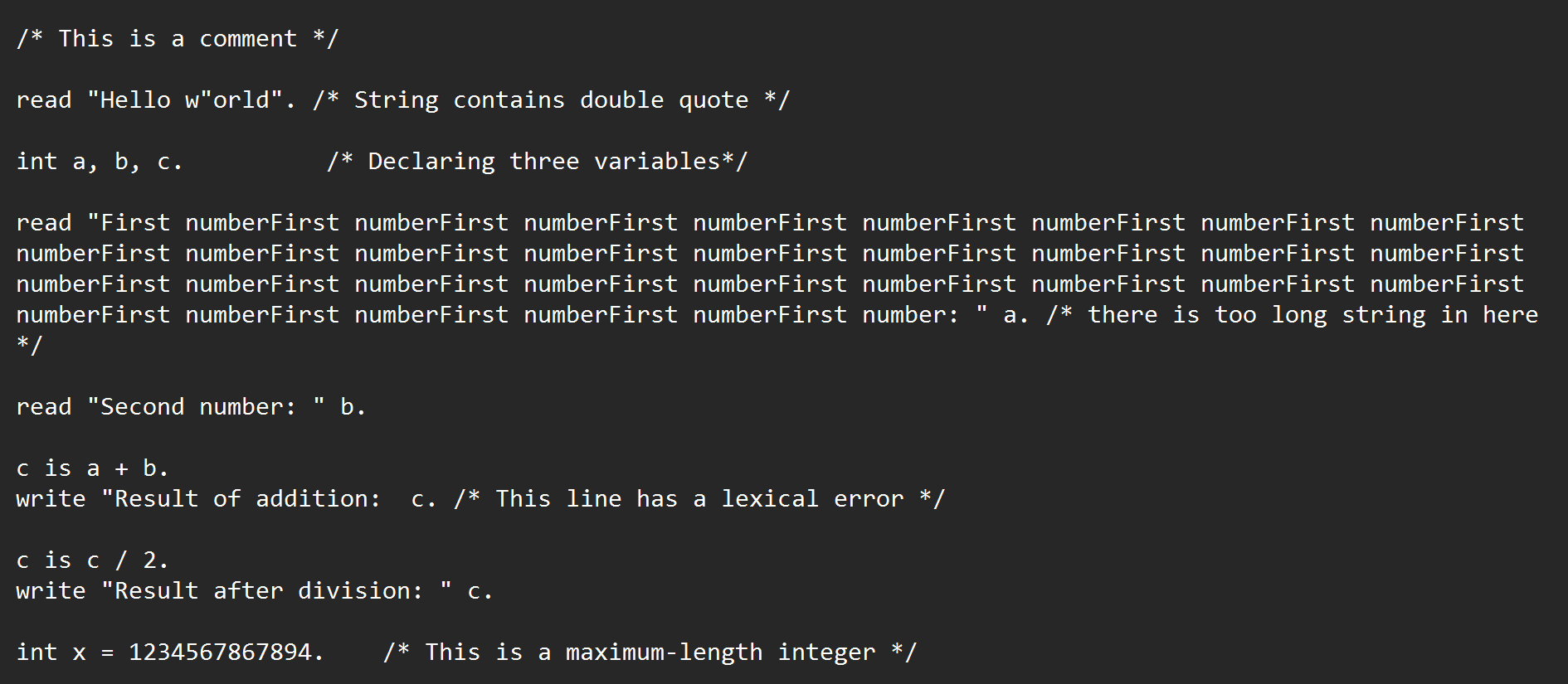
**Output:**

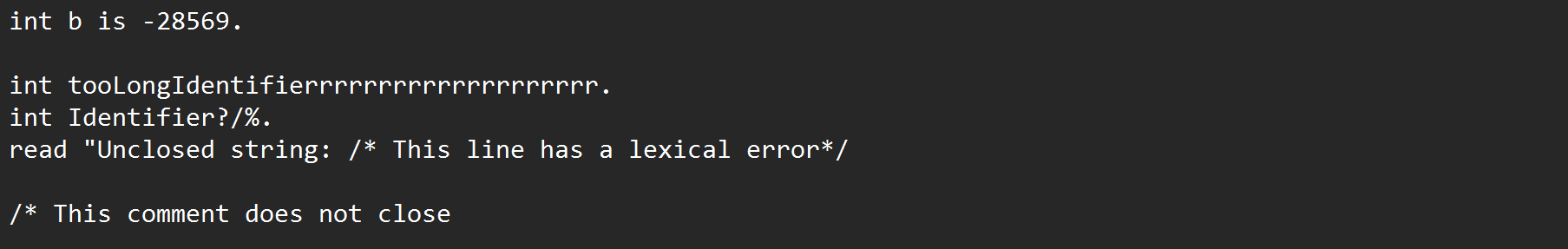
****

A test case that performs string checks.

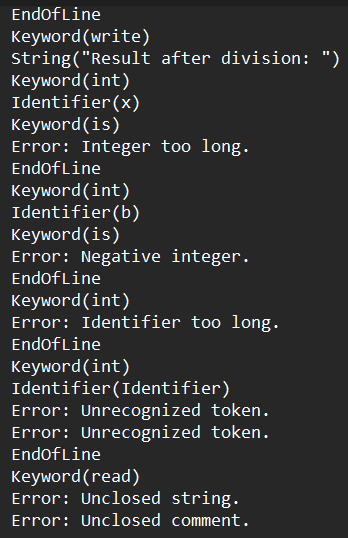
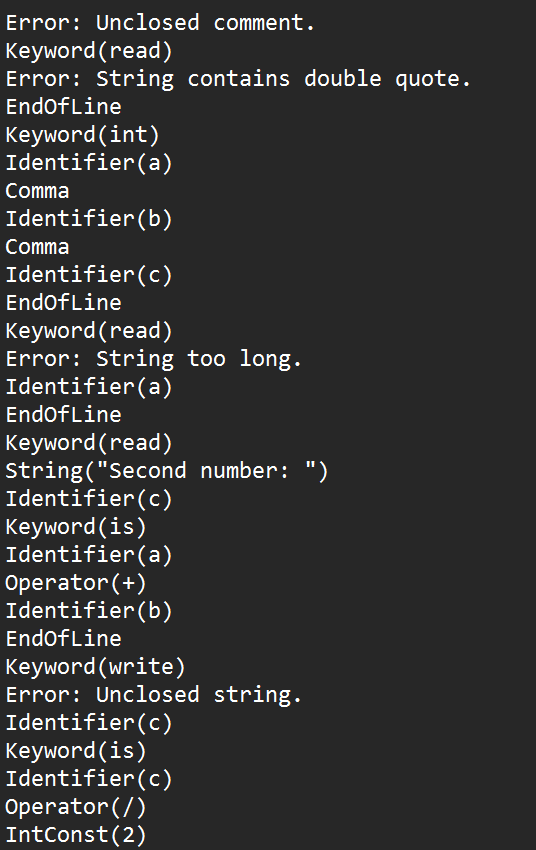
**6. (All Cases)**

**Input:**

****

****

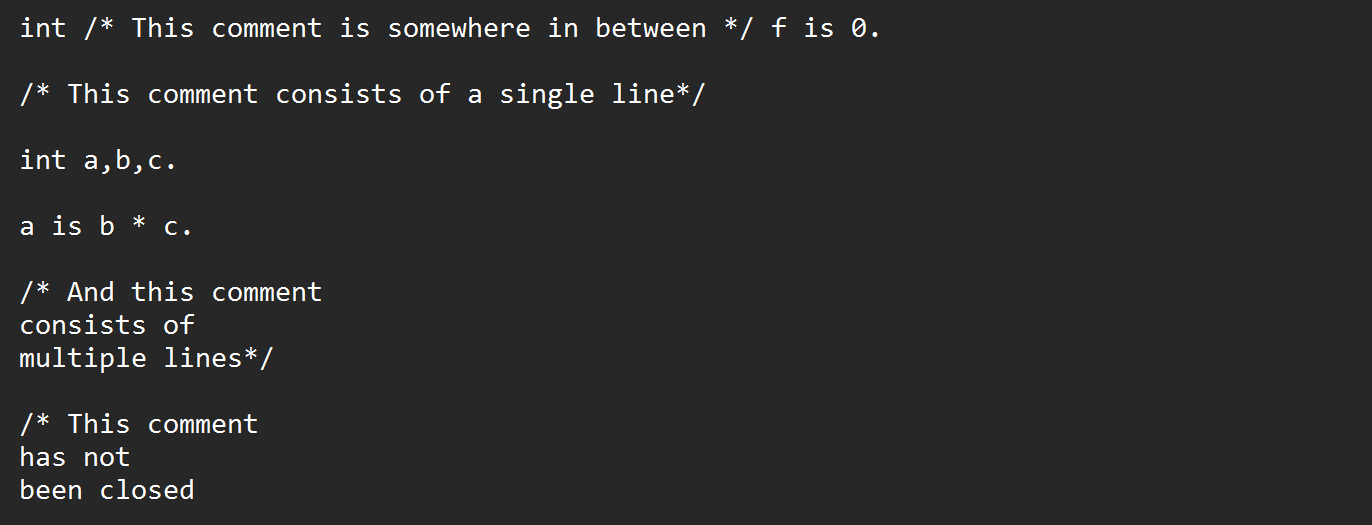
**Output:**

****

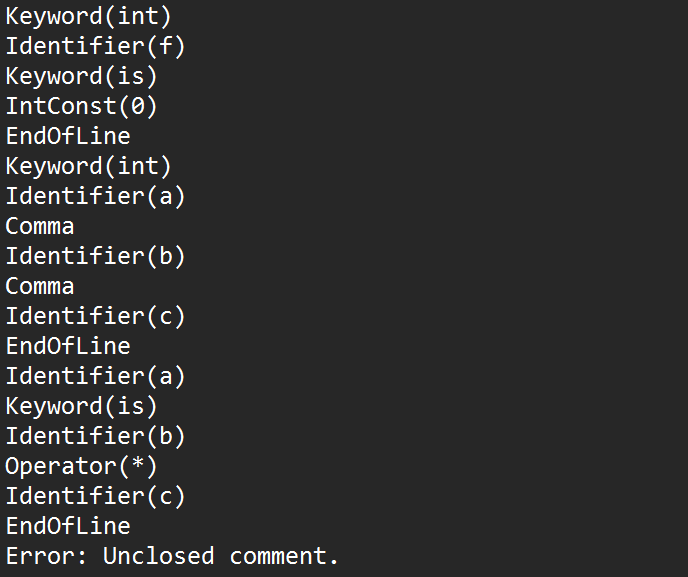
In this example, we tried to check all error cases. Here is an example with the incorrect comment check and all of the other error cases we checked above.

**7. (Comments)**

**Input:**

****

**Output:**

****

As can be seen here, error-free comments are not received as tokens and are ignored, regardless of where they are in the line or code. An error is given for faulty tokens / tokens that have not been closed.