**DAY 1**

**Date:13/02/2005**

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 <ctype.h>

#include <string.h>

#define MAX\_ID\_LEN 31

int isOperator(char ch) {

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

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

if (ch == operators[i]) {

return 1;

}

}

return 0;

}

int isKeyword(const char \*str) {

char \*keywords[] = {"int", "float", "char", "if", "else", "while", "for", "return", "void"};

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

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

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

return 1;

}

}

return 0;

}

void lexicalAnalyzer(const char \*code) {

int i = 0;

char token[MAX\_ID\_LEN];

int tokenIndex = 0;

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

if (isspace(code[i])) {

i++; // Ignore spaces, tabs, and new lines

continue;

}

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

tokenIndex = 0;

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

if (tokenIndex < MAX\_ID\_LEN - 1)

token[tokenIndex++] = code[i];

i++;

}

token[tokenIndex] = '\0';

if (isKeyword(token)) {

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

} else {

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

}

}

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

tokenIndex = 0;

while (isdigit(code[i])) {

token[tokenIndex++] = code[i];

i++;

}

token[tokenIndex] = '\0';

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

}

else if (isOperator(code[i])) {

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

i++;

}

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

while (code[i] != '\n' && code[i] != '\0') {

i++;

}

}

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

i += 2;

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

i++;

}

if (code[i] != '\0') {

i += 2;

}

}

else {

printf("Unknown: %c\n", code[i]);

i++;

}

}

}

int main() {

char code[] = "int a = 5; \na = a + 10;\n

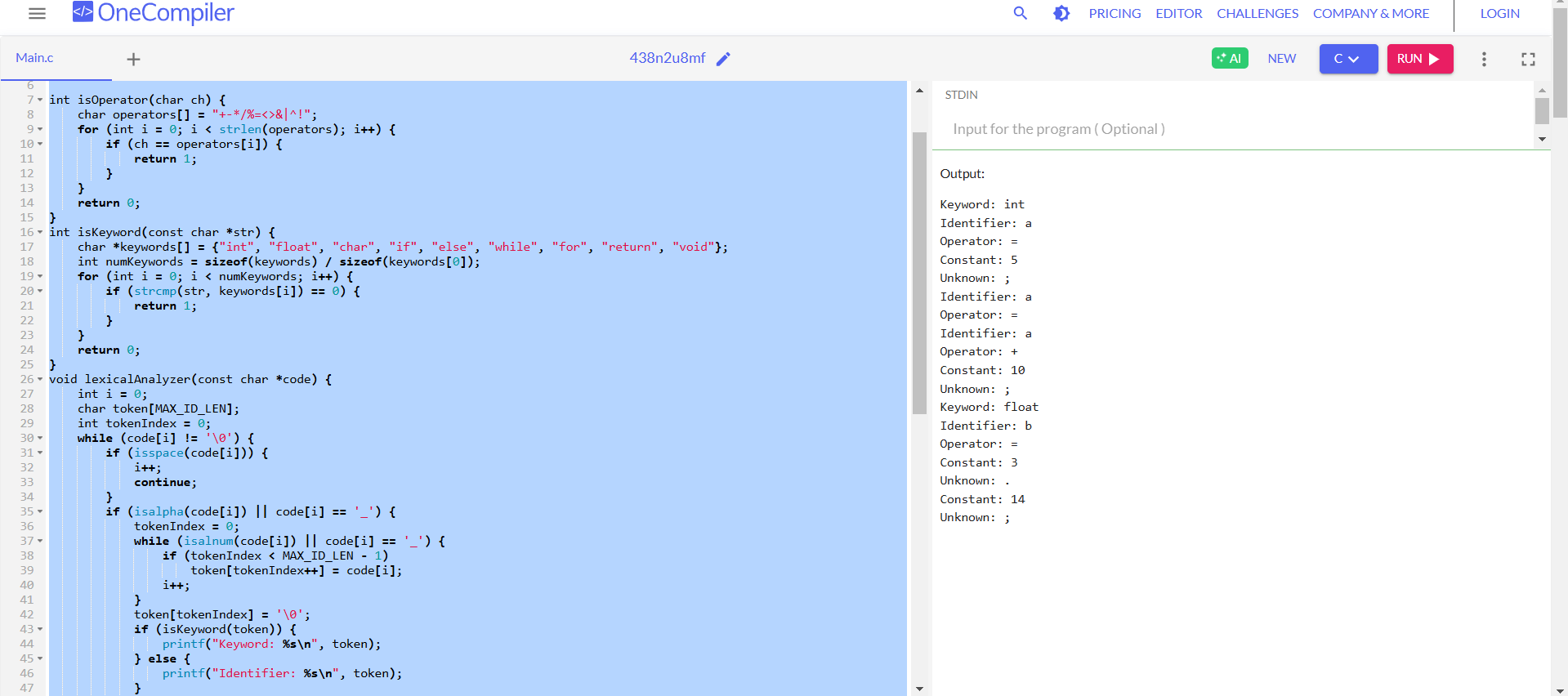
float b = 3.14;\n";

lexicalAnalyzer(code);

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>

#include <stdbool.h>

bool is\_single\_line\_comment(const char \*line) {

return strncmp(line, "//", 2) == 0;

}

bool is\_multi\_line\_comment(const char \*line) {

return strncmp(line, "/\*", 2) == 0 && strstr(line, "\*/") != NULL;

}

const char\* check\_comment(const char \*line) {

if (is\_single\_line\_comment(line)) {

return "Single-line comment";

} else if (is\_multi\_line\_comment(line)) {

return "Multi-line comment";

} else {

return "Not a comment";

}

}

int main() {

const char \*test\_lines[] = {

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

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

"int a = 10; // Variable declaration",

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

"/\* Multi-line\n comment spanning\n multiple lines \*/"

};

int num\_tests = sizeof(test\_lines) / sizeof(test\_lines[0]);

for (int i = 0; i < num\_tests; i++) {

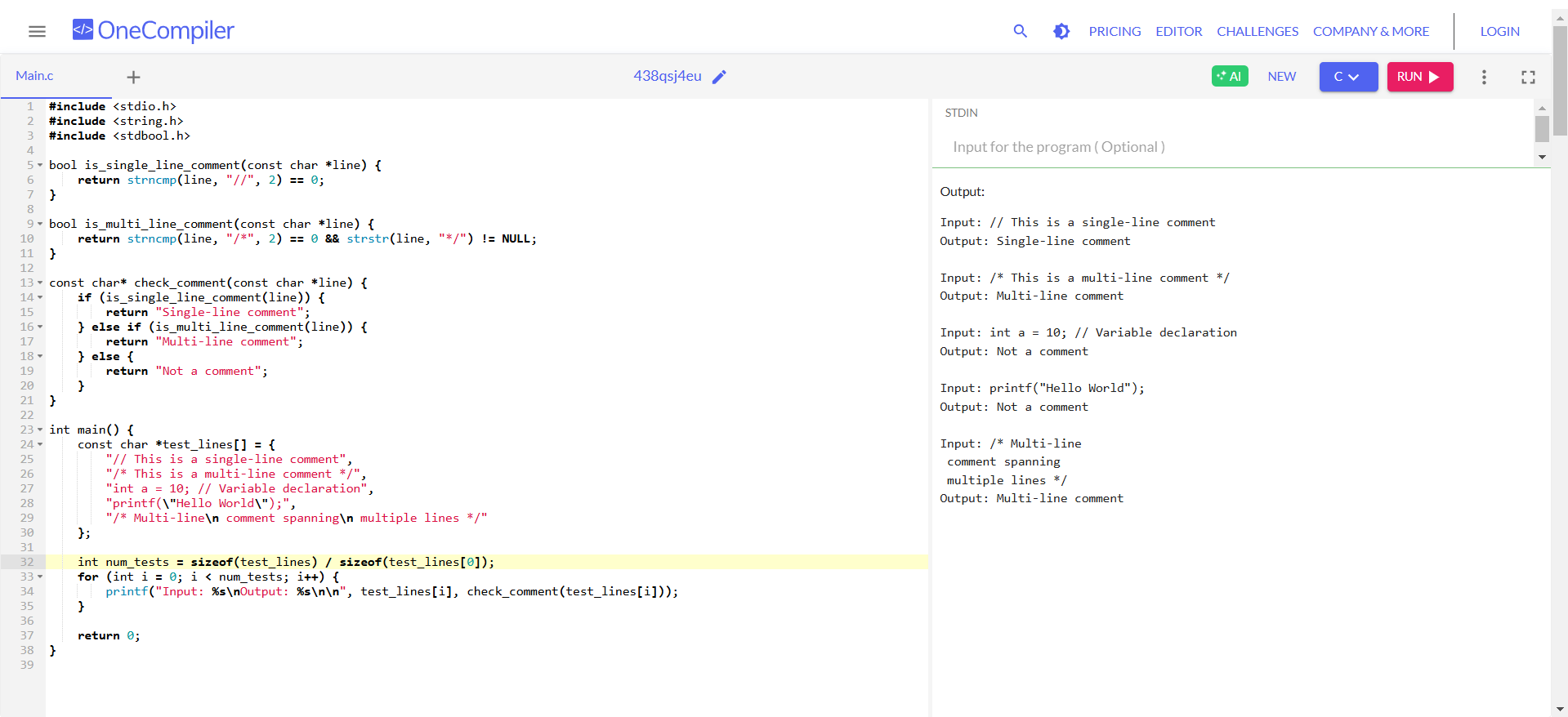
printf("Input: %s\nOutput: %s\n\n", test\_lines[i], check\_comment(test\_lines[i]));

}

return 0;

}

**OUTPUT:**



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

**CODE:**

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

bool is\_operator(const char \*token) {

return (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||

strcmp(token, "\*") == 0 || strcmp(token, "/") == 0);

}

const char\* check\_operator(const char \*token) {

if (is\_operator(token)) {

return "Valid operator";

} else {

return "Not an operator";\

}

}

int main() {

const char \*test\_tokens[] = {

"+", "-", "\*", "/", "%", "&", "x", "++"

};

int num\_tests = sizeof(test\_tokens) / sizeof(test\_tokens[0]);

for (int i = 0; i < num\_tests; i++) {

printf("Input: %s\nOutput: %s\n\n", test\_tokens[i], check\_operator(test\_tokens[i]));

}

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

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

**CODE:**

#include <stdio.h>

void countSpaces(char str[]) {

int i = 0, spaces = 0, newlines = 0;

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

if (str[i] == ' ') spaces++;

if (str[i] == '\n') newlines++;

i++;

}

printf("Spaces: %d, Newlines: %d\n", spaces, newlines);

}

int main() {

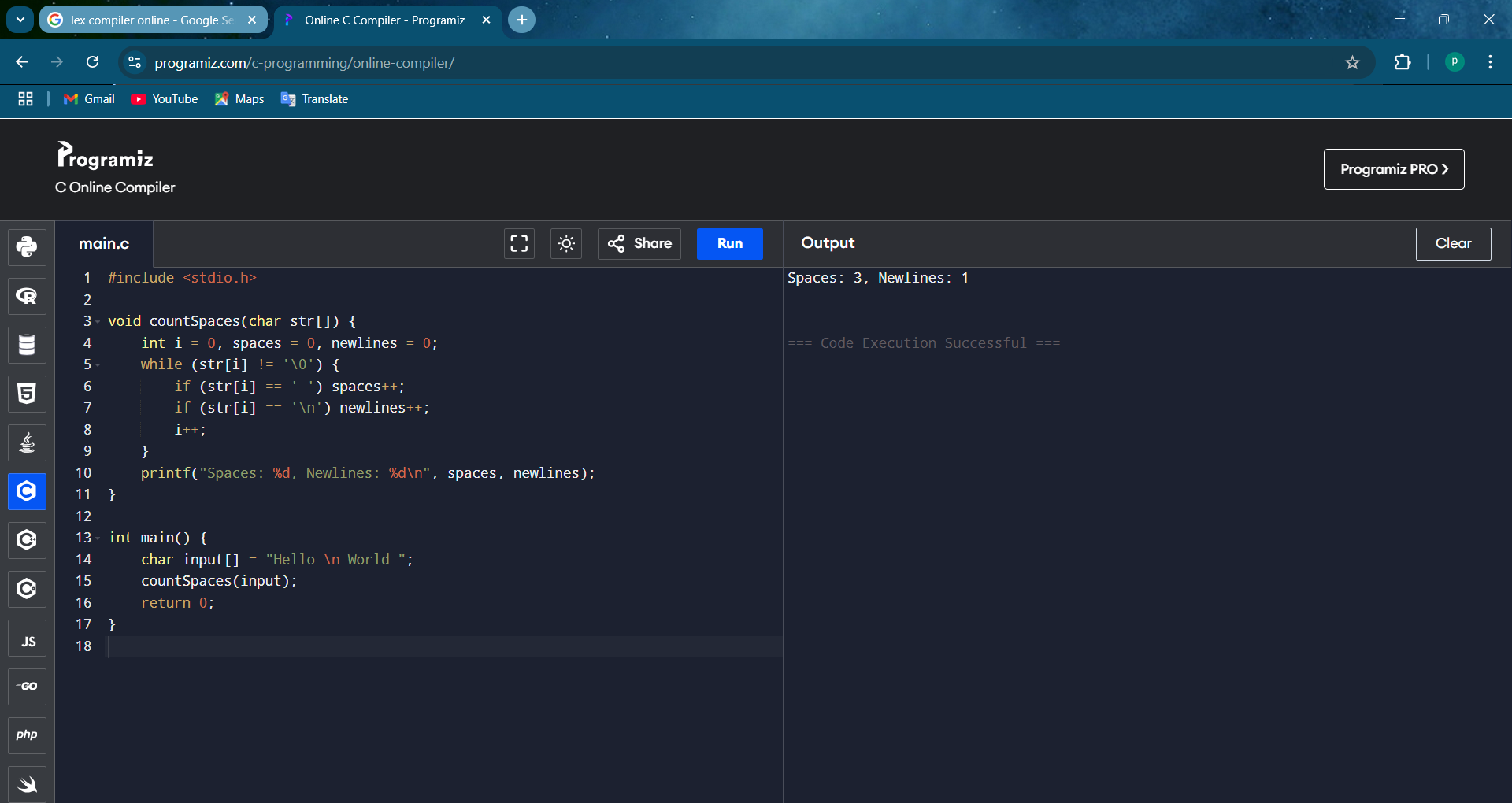
char input[] = "Hello \n World ";

countSpaces(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>

void checkIdentifier(char str[]) {

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

printf("Invalid Identifier\n");

return;

}

int i = 1;

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

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

printf("Invalid Identifier\n");

return;

}

i++;

}

printf("Valid Identifier\n");

}

int main() {

char id[] = "var1";

checkIdentifier(id);

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

6. Implement a C program to eliminate left recursion.

**CODE:**

#include <stdio.h>

#include <string.h>

void eliminateLeftRecursion(char nonTerminal, char alpha[], char beta[]) {

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

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

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

}

int main() {

char nonTerminal = 'A';

char alpha[] = "x";

char beta[] = "y";

printf("Given Grammar:\n");

printf("%c -> %c%s | %s\n", nonTerminal, nonTerminal, alpha, beta);

eliminateLeftRecursion(nonTerminal, alpha, beta);

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

7. Implement a C program to eliminate left factoring.

**CODE:**

#include <stdio.h>

#include <string.h>

void eliminateLeftFactoring(char nonTerminal, char common[], char diff1[], char diff2[]) {

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

printf("%c -> %s%c'\n", nonTerminal, common, nonTerminal);

printf("%c' -> %s | %s\n", nonTerminal, diff1, diff2);

}

int main() {

char nonTerminal = 'A';

char common[] = "x"; // Common part

char diff1[] = "y"; // First alternative

char diff2[] = "z"; // Second alternative

printf("Given Grammar:\n");

printf("%c -> %s%s | %s%s\n", nonTerminal, common, diff1, common, diff2);

eliminateLeftFactoring(nonTerminal, common, diff1, diff2);

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.