A logo with letters and numbers

Description automatically generated

**IUBAT- International University of Business Agriculture and Technology**

**Assaignment**

**Topic: Design & Develop an Open-Ended Lexical Analyzer**

**Course No.: CSC 437**

**Course Title: Compiler Design**

**Section: E**

**Submitted To**

**Rafiqul Islam Munna**

**Lecturer,**

**Dept. Of Computer Science and Engineering (CSE)**

**IUBAT**

**Submitted By**

| **Student Name** | **Student ID** |
| --- | --- |
| Md Fojailibne Nur | 22203222 |
|  | ID2 |
|  | ID3 |

Submission Date: 25/09/2025

**Design & Develop an Open-Ended Lexical Analyzer**

# 1. Introduction

This project implements a simplified lexical analyzer (tokenizer) in the C programming language. The analyzer processes an input program, splits it into tokens, and classifies each token according to defined categories such as keywords, identifiers, operators, numbers, strings, comments, and errors.  
  
The analyzer includes special rules for:  
- Recognizing three student names (Fojael, Kawser, Foyjur) as special identifiers.  
- Validating student IDs using their last three digits (222, 221, 167).  
- Handling strings marked with $...$ and both single-line (// ...) and multi-line (/\* ... \*/) comments.  
  
This tool provides practical understanding of compiler front-end design, focusing on the lexical analysis phase.

# 2. Conflict Analysis & Decisions

- Identifiers starting with digits vs. Numbers  
 \* Issue: Initially, numbers like 123 were being classified as identifiers since digit-check came before number-check.  
 \* Decision: Numbers are checked before identifiers to avoid misclassification.

- Special Student IDs  
 \* Issue: Full IDs (e.g., 22203222) could not be matched since only last 3 digits were stored (222).  
 \* Decision: Allowed prefix-matching so full IDs starting with last-3 digits are recognized.

- Multi-word Strings with Spaces  
 \* Issue: $Hello World$ broke into two tokens due to space-based splitting.  
 \* Decision: Use character-based scanning instead of space-based splitting to capture entire strings.

- Nested Comments  
 \* Issue: /\* outer /\* inner \*/ still outer \*/ caused issues.  
 \* Decision: Nested comments not supported; flagged as error.

- Operators Not in Rule Set  
 \* Issue: Example: <> is not a valid operator in the defined grammar.  
 \* Decision: Classified as <ERROR, Invalid symbol: <>.

# 3. Hand-Written Diagrams

Insert scanned/photo diagrams here. Suggested diagrams:  
- Flowchart of lexical analysis process  
- Token classification decision tree  
- Example of input → tokens mapping

A diagram of a diagram

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a transition diagram

Description automatically generated

A diagram of a flowchart

Description automatically generated

# 4. Code Overview

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LEN 1024

// -------------------- RULES --------------------

// Keywords

const char \*KEYWORDS[] = {"if", "else", "while", "return", "func"};

int KEYWORDS\_COUNT = 5;

// Special Identifiers (Three Students)

const char \*SPECIAL\_NAMES[] = {

"Fojael",

"Kawser",

"Foyjur"

};

const char \*SPECIAL\_IDS[] = {

"222", // Fojael → 22203222

"221", // Kawser → 22203221

"167" // Foyjur → 22203167

};

int SPECIAL\_COUNT = 3;

// Operators

const char \*OPERATORS[] = {"+", "-", "\*", "/", "==", "!=", "<", ">", "<=", ">="};

int OPERATORS\_COUNT = 10;

// -------------------- HELPERS --------------------

int isKeyword(const char \*lex) {

for (int i = 0; i < KEYWORDS\_COUNT; i++) {

if (strcmp(lex, KEYWORDS[i]) == 0) return 1;

}

return 0;

}

int isOperator(const char \*lex) {

for (int i = 0; i < OPERATORS\_COUNT; i++) {

if (strcmp(lex, OPERATORS[i]) == 0) return 1;

}

return 0;

}

int isSpecialName(const char \*lex) {

for (int i = 0; i < SPECIAL\_COUNT; i++) {

if (strcmp(lex, SPECIAL\_NAMES[i]) == 0) return 1;

}

return 0;

}

int isSpecialID(const char \*lex) {

for (int i = 0; i < SPECIAL\_COUNT; i++) {

if (strcmp(lex, SPECIAL\_IDS[i]) == 0) return 1;

}

return 0;

}

// -------------------- TOKEN CLASSIFIER --------------------

void classifyToken(const char \*lexeme) {

if (strlen(lexeme) == 0) return;

// --- Special Names ---

if (isSpecialName(lexeme)) {

printf("<IDENTIFIER\_NAME, %s>\n", lexeme);

return;

}

// --- Special IDs ---

if (isSpecialID(lexeme)) {

printf("<ID\_LAST3, %s>\n", lexeme);

return;

}

// --- Keyword ---

if (isKeyword(lexeme)) {

printf("<KEYWORD, %s>\n", lexeme);

return;

}

// --- Identifier ---

if (isdigit(lexeme[0])) {

// must start with one of the last-3 ID digits

int valid = 0;

for (int i = 0; i < SPECIAL\_COUNT; i++) {

if (strncmp(lexeme, SPECIAL\_IDS[i], strlen(SPECIAL\_IDS[i])) == 0) {

valid = 1;

break;

}

}

if (valid) {

printf("<IDENTIFIER, %s>\n", lexeme);

} else {

printf("<ERROR, Invalid identifier: %s>\n", lexeme);

}

return;

}

// --- Number ---

if (isdigit(lexeme[0]) || lexeme[0] == '.') {

int dotCount = 0, expCount = 0;

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

if (lexeme[i] == '.') dotCount++;

if (lexeme[i] == 'e' || lexeme[i] == 'E') expCount++;

}

if (dotCount > 1 || expCount > 1) {

printf("<ERROR, Malformed number: %s>\n", lexeme);

} else {

printf("<NUMBER, %s>\n", lexeme);

}

return;

}

// --- String Literal ---

if (lexeme[0] == '$') {

int len = strlen(lexeme);

if (len >= 2 && lexeme[len - 1] == '$') {

char strVal[MAX\_LEN];

strncpy(strVal, lexeme + 1, len - 2);

strVal[len - 2] = '\0';

printf("<STRING, %s>\n", strVal);

} else {

printf("<ERROR, Unterminated string: %s>\n", lexeme);

}

return;

}

// --- Operators ---

if (isOperator(lexeme)) {

printf("<OPERATOR, %s>\n", lexeme);

return;

}

// --- Comments ---

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

printf("<COMMENT, %s>\n", lexeme);

return;

}

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

if (strlen(lexeme) >= 4 &&

lexeme[strlen(lexeme) - 2] == '\*' &&

lexeme[strlen(lexeme) - 1] == '/') {

printf("<COMMENT, %s>\n", lexeme);

} else {

printf("<ERROR, Unterminated multi-line comment: %s>\n", lexeme);

}

return;

}

// --- Invalid ---

printf("<ERROR, Invalid symbol: %s>\n", lexeme);

}

// -------------------- MAIN --------------------

int main() {

char input[MAX\_LEN];

printf("Enter your program (end with a blank line):\n");

while (fgets(input, sizeof(input), stdin)) {

if (strcmp(input, "\n") == 0) break; // blank line ends input

// remove newline

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

// Simple space-based token split

char \*tok = strtok(input, " ");

while (tok != NULL) {

classifyToken(tok);

tok = strtok(NULL, " ");

}

}

return 0;

}

Key functions in the program:  
- isKeyword(): Checks if a token matches a reserved keyword (if, else, while, return, func).  
- isOperator(): Validates against the operator list (+, -, \*, /, ==, !=, <, >, <=, >=).  
- isSpecialName(): Detects special student names.  
- isSpecialID(): Validates student IDs based on last-3 digits.  
- classifyToken(): Core classifier function that categorizes tokens into keywords, identifiers, numbers, strings, operators, comments, or errors.  
- main(): Reads input line by line, splits into tokens, and calls classifyToken() for each.

# 5. Verification Result (Name & ID)

Insert screenshot of program output showing:

A screen shot of a computer program

Description automatically generated

A computer screen with white text

Description automatically generated  
Input: Fojael 22203222  
Output: <IDENTIFIER\_NAME, Fojael>  
 <IDENTIFIER, 22203222>

# 6. Test Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Input Snippet | Expected Output | Actual Output | Pass/Fail |
| 1. Leading underscore in identifier | \_abc | <ERROR, Invalid symbol: \_abc> |  | Fail |
| 2. func as keyword & identifier | func func123 | <KEYWORD, func> <ERROR, Invalid identifier: func123> |  | Pass |
| 3. .5 as a number literal | .5 | <NUMBER, .5> |  | Pass |
| 4. Multi-line string literal | $Hello World$ | <ERROR, Unterminated string: $Hello> <IDENTIFIER, World$> |  | Fail |
|  |  |  |  |  |
| 5. <> operator | a <> b | <ERROR, Invalid symbol: <> |  | Fail |
| 6. Nested multi-line comments | /\* outer /\* inner \*/ still outer \*/ | <ERROR, Unterminated multi-line comment: ...> |  | Fail |
| 7. Name & ID | Fojael 22203222 | <IDENTIFIER\_NAME, Fojael> <IDENTIFIER, 22203222> |  | Pass |

# 7. Conclusion

This project successfully implemented a basic lexical analyzer with additional rules for handling custom identifiers (names & IDs). Through testing, the analyzer demonstrated its ability to correctly identify keywords, numbers, operators, and strings, while also detecting errors in malformed tokens.  
  
Key learnings:  
- Importance of tokenization order (numbers before identifiers).  
- Handling multi-line and special-case tokens requires careful state management.  
- Not all language features (e.g., nested comments) can be supported in a simple scanner.  
  
This work provided practical exposure to compiler construction concepts, strengthening understanding of lexical analysis and error handling in programming languages.

# 8. PO Justification

|  |  |
| --- | --- |
| **PO(b) Requirement** | **Justification** |
| **P1: Depth of Knowledge Required** | **K3:** Students apply core engineering concepts of compiler design, especially lexical analysis, by designing input buffering, tokenization, and error handling using formal methods. **K4:** Students explore advanced topics such as DFA-based token recognition, error recovery strategies, and ambiguous lexical rules, reflecting real-world compiler design challenges and demonstrating expert knowledge. |
| **P2: Range of Conflicting Requirements** | Lexical rules are intentionally incomplete and conflicting (e.g., identifier rules, keyword ambiguity, multi-line comments). Students analyze and resolve these conflicts, balancing trade-offs to create a consistent lexical analyzer design. |
| **P3: Depth of Analysis Required** | With no fixed solution, students think abstractly to model lexical tokenization and error management. They justify design choices and create flexible solutions adaptable to language evolution. |