Lexical Analyzer – A Small Utility Program

Course: *CEW*

Project Title: *Lexical Analyzer*

Group Members:

- *Osama Hassan -- CS-24120*
- *Rameel Ahmed – CS-24127*
- *Abdul Hadi – CS-24136*

Department: *CIS*

1. Abstract

This project implements a Lexical Analyzer written in C that performs tokenization of C source code. The program reads a C source file, recognizes tokens (keywords, identifiers, numeric literals, operators, punctuation, string and character literals), and prints a formatted table of tokens and their values. The analyzer is modular, performs safe file input, and handles common C language constructs including comments and escape sequences.

2. Problem Statement

Create a compact, reliable utility that performs lexical analysis of C source code files. The tool should:

1. Read a C source file safely.
2. Identify and categorize tokens such as keywords, identifiers, integers (including binary, octal, hexadecimal), floating-point numbers, operators (including multi-character operators), punctuation, string literals, and character literals.
3. Ignore comments and treat string/char literals as single tokens.
4. Present tokens in a clear, tabular output suitable for inspection or for feeding into later compiler phases.

3. Objectives

1. Build a single-file C program that demonstrates modular programming (functions for discrete tasks).
2. Ensure correct handling of comments, string/char literals, escape sequences, and numeric formats.
3. Provide safe file reading with buffer-limit checks to avoid overflow.
4. Produce a readable token list printed to the console.

4. Tools & Environment

Language: *C*
Compiler: *gcc*
Libraries: *<stdio.h>, <stdlib.h>, <stdbool.h>, <ctype.h>, <string.h>*
Buffer limit macro: *#define MAX_LENGTH 10000 — tunable for larger inputs*
Compile & run: `gcc main.c -o main`

5. Program Design & Modular Structure

The implementation is organized as a set of clear, single-purpose functions. Each function encapsulates a small unit of lexical logic and keeps the scanner readable and maintainable.

Core components and responsibilities:

is*Delimeter(char)*
Recognizes characters that separate tokens (spaces, newlines, punctuation, many operators). Used to

determine token boundaries.

isOperator(char)
Identifies one-character operators.

isPunctuation(char)
Identifies punctuation characters printed as separate tokens.

isValidIdentifier(char)*
Validates C-like identifiers: must not start with a digit and may contain letters, digits, and underscores.

isKeyword(char*)
Checks whether a token matches a table of common C keywords and selected library functions (e.g., printf, scanf, main). The list can be adjusted for strict language adherence.

isInteger(char)*
Detects integer literals in multiple bases: binary (0b...), hexadecimal (0x...), octal (leading 0), and decimal. Returns true if the string represents a valid integer literal under these formats.

isFloat(char)*
Detects floating-point literals that contain exactly one decimal point and otherwise digits. Note: scientific notation is not supported in the current implementation.

skipCharLiteral(const char, int)*
Advances the scanning index past a character literal, correctly handling escapes such as ' '.

skipString(char, int)*
Advances the scanning index past string literals while respecting escaped quotes and other escape sequences.

skipComments(char, int)*
Skips both single-line (//) and multi-line (/* ... */) comments.

getSubstring(char, int, int)*
Returns a dynamically allocated copy of a substring for token classification.

readFile(const char)*
Reads the entire file into a dynamically allocated buffer while preventing overflow beyond MAX_LENGTH.

lexiAnalyzer(char)*
The main scanning function. Uses a two-pointer technique (left and right) to identify tokens and classifies them by calling the helper functions above.

6. High-level Algorithm

1. Read entire source file into a buffer.
2. Use two indices left and right to create a sliding window over the buffer.
3. Move right forward until a delimiter is encountered; when found, extract the window buffer[left..right-1] and classify it.

4. Handle delimiter characters (operators, punctuation) directly and classify multi-character operators with lookahead.
5. Special-case handling for string literals, character literals, and comments ensures they are treated as atomic units or skipped (comments).
6. Print each recognized token with its type and value in a formatted table.

This approach results in a linear-time scan over the input with occasional short scans for classification and keyword lookup.

7. Edge Cases & Implementation Notes

The implementation explicitly addresses several tricky cases:

String literals: The scanner collects entire string literals, including escaped quotes, so contents such as " are preserved and the literal is not split.
Character literals: Handles both simple characters ('A') and escaped forms (' ', '\'). The skip routine prevents accidental splitting.
Comments: Both // single-line and /* ... */ multi-line comments are recognized and ignored in token output.
Numeric bases: Integer recognition supports binary (0b/0B), hexadecimal (0x/0X), octal (leading 0), and decimal. Floats are recognized when a single decimal point occurs with digits on both sides.
Multi-character operators: The scanner detects ++, --, ==, !=, <=, and >= using lookahead.

Limitations:

- The float parser does not accept scientific notation (e.g., 1e-3) or leading +/- signs as part of the numeric token. Those can be added with a small extension.
- The keyword list contains printf, scanf, and main to highlight them in output; for strict correctness these are not language keywords and can be removed.
- Preprocessor tokens (e.g., complex # macro lines) are treated simply as punctuation; a dedicated preprocessor stage would be required for full conformance.

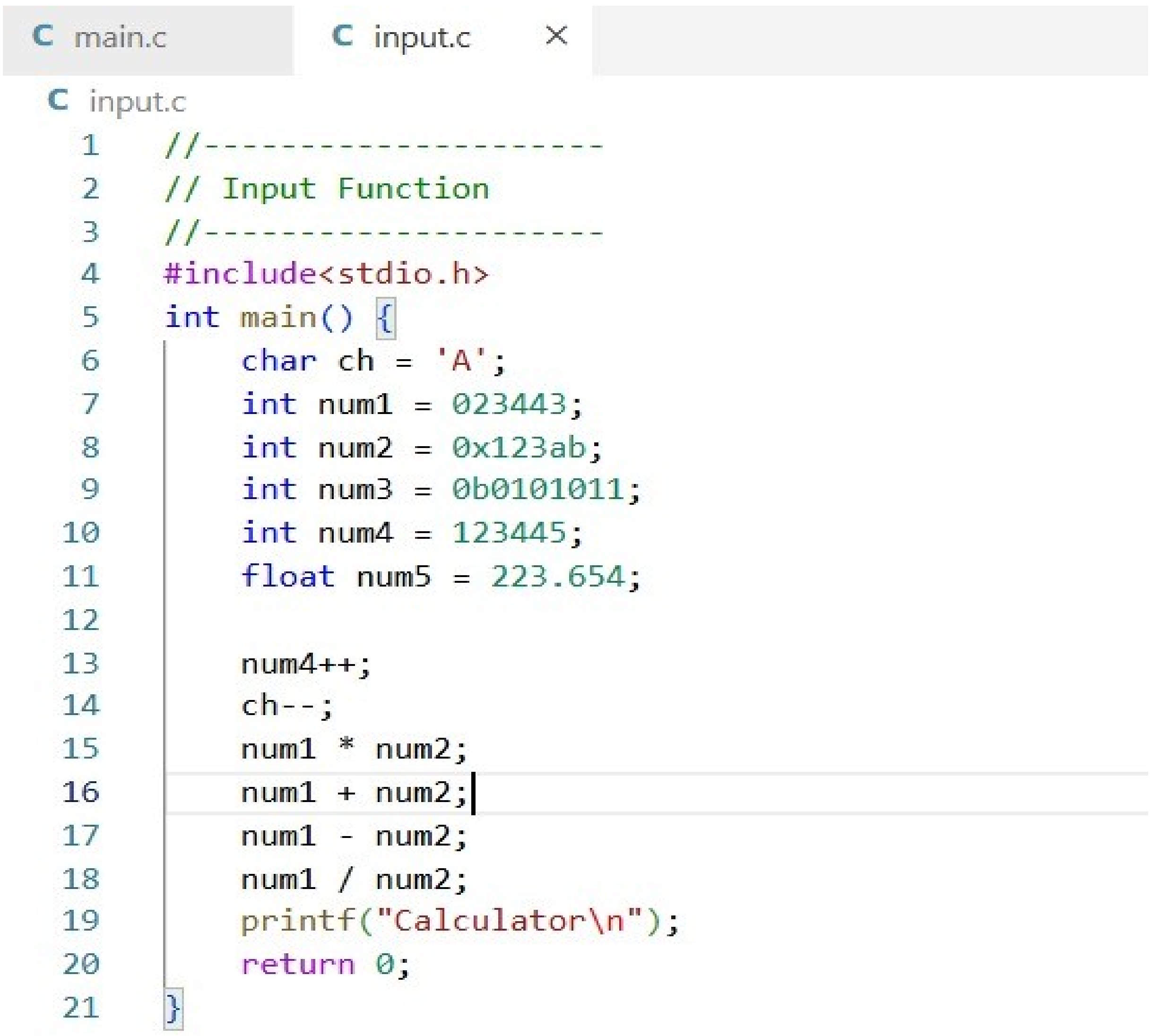
8. Time and Space Complexity

Time complexity: The scanner performs essentially a single pass over the input buffer; each character is examined a constant number of times. Keyword checks are linear in the small fixed keyword list. Overall time complexity is *O(n)* for practical input sizes, where n is the number of characters in the file.

Space complexity: The program stores the entire file in memory *O(n)*, plus temporary allocations for token substrings. Memory usage is bounded by MAX_LENGTH plus a small overhead per token.

9. Sample Input and Output

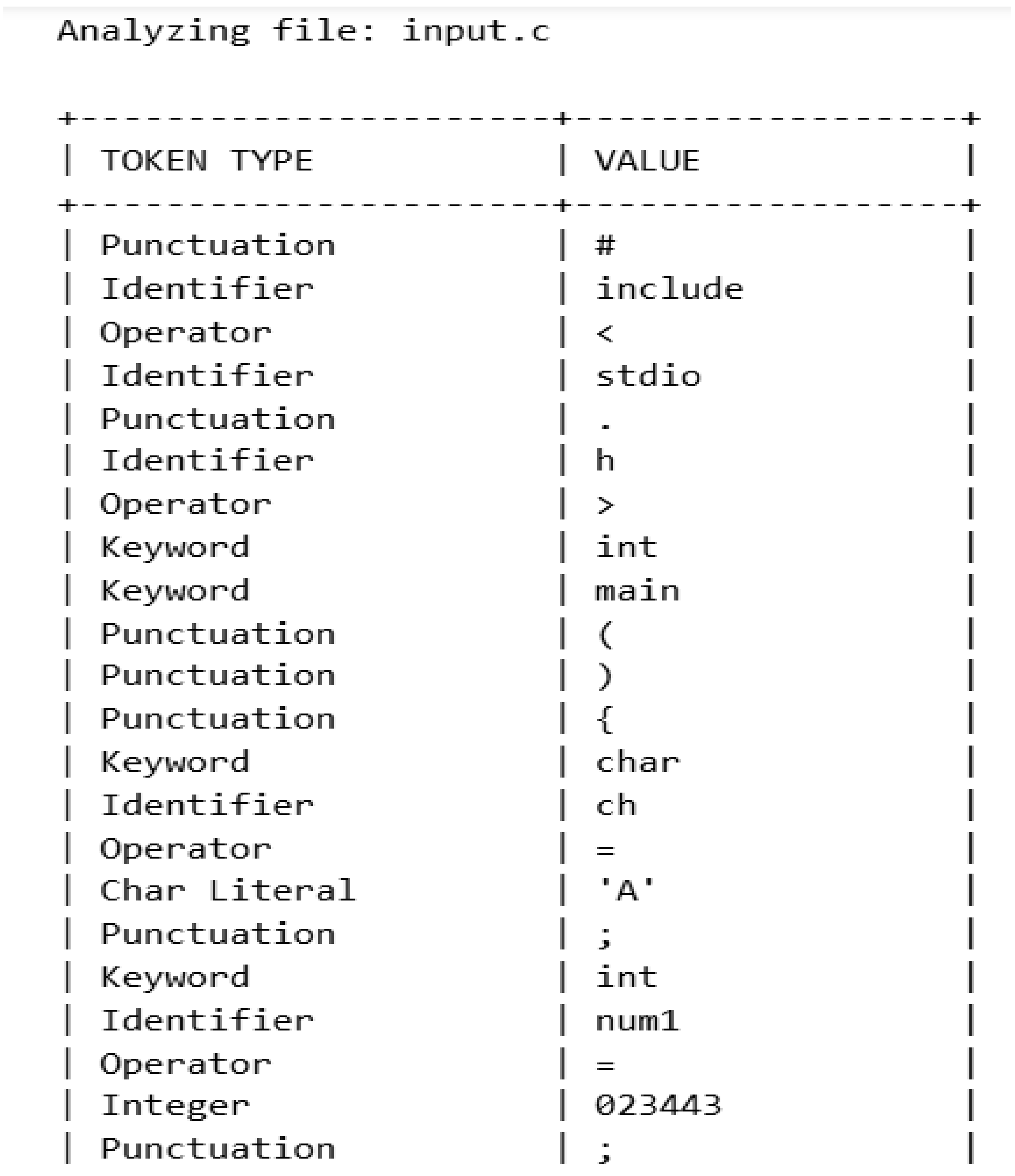
Sample input:



```c
//---------------------
// Input Function
//---------------------
#include<stdio.h>
int main() {
    char ch = 'A';
    int num1 = 023443;
    int num2 = 0x123ab;
    int num3 = 0b0101011;
    int num4 = 123445;
    float num5 = 223.654;

    num4++;
    ch--;
    num1 * num2;
    num1 + num2;
    num1 - num2;
    num1 / num2;
    printf("Calculator\n");
    return 0;
}
```

Sample output:



```
Analyzing file: input.c

+-----------------------+-------------------+
| TOKEN TYPE            | VALUE             |
+-----------------------+-------------------+
| Punctuation           | #                 |
| Identifier            | include           |
| Operator              | <                 |
| Identifier            | stdio             |
| Punctuation           | .                 |
| Identifier            | h                 |
| Operator              | >                 |
| Keyword               | int               |
| Keyword               | main              |
| Punctuation           | (                 |
| Punctuation           | )                 |
| Punctuation           | {                 |
| Keyword               | char              |
| Identifier            | ch                |
| Operator              | =                 |
| Char Literal          | 'A'               |
| Punctuation           | ;                 |
| Keyword               | int               |
| Identifier            | num1              |
| Operator              | =                 |
| Integer               | 023443            |
| Punctuation           | ;                 |
```

10. Error Handling & Safety

1. *readFile()* checks for file open errors and memory allocation failures and reports them using *perror()* before terminating.
2. File content concatenation checks the current buffer length against *MAX_LENGTH* to prevent buffer overflow.
3. Functions use casts to (unsigned char) when passing characters to i*sdigit()* or i*sspace()* to avoid undefined behavior for negative char values on some platforms.
4. Allocated substrings are freed immediately after classification to prevent memory leaks.

11. Possible Improvements and Extensions

1. Accept a filename from command-line arguments instead of using a hard-coded input.c name.
2. Add support for scientific notation in floats and optional leading +/- signs for numbers.

3. Produce token metadata such as line and column numbers for error reporting and diagnostics.
4. Replace temporary substring allocations with views into the main buffer to reduce memory churn.
5. Add a JSON or CSV output mode for programmatic consumption by other tools.
6. Integrate a preprocessor pass to correctly handle #include, macros, and conditional compilation.
7. Build a symbol table to prepare for semantic analysis stages.

12. Conclusion

This Lexical Analyzer provides a compact and practical demonstration of lexical scanning techniques in C. It cleanly separates concerns into helper functions, handles common C language features, and produces a clear token listing. The design is intentionally simple but extensible, making it suitable as a learning tool or as the first stage in a small compiler or static analysis project.