Session # 04: Building a Hand-Coded Lexer in C

1. Introduction

The lexical analyzer, or **lexer**, is the first stage of a compiler. Its task is to read the source code and produce a sequence of **tokens**, each describing a meaningful unit such as a keyword, identifier, number, or operator.

The objective of this series of exercises is to build a lexer manually in C, step by step, in order to understand how lexical analysis works before using automatic tools like *Lex* or *Flex*. The construction process is organized into progressive exercises:

- **Exercises 1–2** introduce file handling and the token data structure.
- **Exercises 3–5** focus on helper functions for classifying characters and keywords.
- **Exercise 6** develops the ability to skip whitespace and comments (since whitespace and comments are not meaningful tokens, the lexer must be able to ignore them while maintaining correct line counting).
- **Exercises 7–10** gradually implement token recognition (identifiers, numbers, operators, delimiters, strings, and character literals).
- **Exercises 11–12** integrate all parts into a complete lexer that reads a source file and produces a token table with lexeme, type, value, and line number.
- **Exercises 13–16** design and implement a symbol table that will be used by the lexer to store and manage identifiers.

At the end of this project, the student will have constructed a working hand-coded lexer for a small programming language, gaining practical insight into how lexical analysis is performed in compilers.

2. Preliminary Requirements

2.1. Token Specification

The lexer must be able to detect and classify the following categories of tokens:

- **Keywords**: reserved words of the language (int, if, else, while, return).
- **Identifiers**: sequences beginning with a letter or underscore, followed by letters, digits, or

underscores. *Examples*: x, count1, _temp.

- **Numbers**: integer constants composed of digits (*e.g.*, 42, 0, 1234).
- **Operators**: arithmetic operators (+, -, *, /), assignment operators (=, +=, -=), and relational operators (==, !=, <, >, <=, >=).
- **Delimiters**: separators such as ; , ( ) { }.
- **String literals**: sequences enclosed in double quotes ("Hello"), possibly with escape characters (\", \\).
- **Character literals**: single characters enclosed in single quotes ('a', '\\n').
- **Comments**: ignored by the lexer but must be skipped correctly. Two forms are recognized:
    - ✓ **Single-line comments** beginning with //,
    - ✓ **Multi-line comments** enclosed in /* ... */.
- **Whitespace**: spaces, tabs, and newlines are not tokens, but newlines are important for line numbering.

2.2. Token Data Structure

Each token must be represented by a C structure that contains four pieces of information:

```
typedef struct {
    char lexeme[100];   // the exact text of the token
    char type[20];      // the token category (e.g., KEYWORD, IDENTIFIER)
    char value[100];   // the token value if applicable (e.g., numeric value, character, string)
    int line;           // the line number where the token appears
} Token;
```

- ➢ **The lexeme** stores the raw string as it appears in the source code.
- ➢ **The type** specifies the category of the token.
- ➢ **The value** provides additional information when relevant: for numbers, it stores the integer value as a string; for character or string literals, it stores the literal value without quotes. If not applicable, the field can be set to "-".
- ➢ **The line** records the line number in the source file where the token was found.

2.3. Error Handling

The lexer must be able to detect and report **invalid inputs**. When an error occurs, a descriptive message should be printed, including the line number where the error was found. Examples of errors include:

- ✓ Invalid identifiers: starting with a digit (*e.g.*, 123abc).
- ✓ Unrecognized symbols: characters not part of the language (*e.g.*, @, #).
- ✓ Unterminated string or character literals: a missing closing quote.
- ✓ Unclosed comments: /* without a matching */.

For each error, the lexer should:

- Print a clear error message (*e.g.*, Error: Invalid identifier '123abc' at line 5).
- Continue scanning the rest of the file (do not stop at the first error).

3. Exercises

Phase 1: File Handling and Data Structures

Exercise 1: File Handling and Line Counting

Write a program that opens a source file using *fopen()*, reads it character by character with *fgetc()*, and counts the number of lines. Print the final line count at the end.

Exercise 2:Define a Token Structure

Define a *struct* **Token** that contains the fields **lexeme**, **type**, **value**, and **line**. Write a small test program that initializes a token and prints its contents.

Phase 2: Character Classification Functions

Exercise 3: Keyword Detection

Write a function `int isKeyword(const char *word)` that returns 1 if the word is one of the reserved keywords (int, if, else, while, return), and 0 otherwise.

Exercise 4: Identifier Rules

In most programming languages, the rule for identifiers distinguishes between the first character and the following characters.

- The first character must be a letter or underscore.
- The following characters may be letters, digits, or underscores.

For this reason, implement two functions:

- `int isIdentifierStart(char c)` to check if a character can start an identifier (letter or underscore).
- `int isIdentifierChar(char c)` to check if a character can be part of an identifier (letter, digit, underscore).

These functions will be used later when constructing the lexer to correctly recognize identifiers.

Exercise 5: Operator Rules

Write a function `int isOperatorChar(char c)` that returns 1 if the character is one of + - * / = < > !, and 0 otherwise.

Phase 3: Skipping and Preprocessing

Exercise 6: Skipping Whitespace and Comments

Write a function that ignores spaces, tabs, and newlines (while updating line numbers), as well as comments. Single-line comments begin with // and multi-line comments are enclosed between /* ... */. Test this by printing only the meaningful characters that remain.

Phase 4: Token Recognition Functions

Exercise 7: Identifiers and Keywords

Write a function that recognizes identifiers. Use `isKeyword()` to decide whether the lexeme should be classified as a keyword or as an identifier.

Exercise 8: Numbers

Write a function that recognizes integers. The token must include both the lexeme and its numeric value stored in the value field.

Exercise 9: Operators and Delimiters

Extend recognition to operators (+ - * / = == != < > <= >=) and delimiters (; , ( ) { }).

Exercise 10: Strings and Character Literals

Write a function that recognizes string literals ("...") and character literals ('c'). Handle escape sequences such as \" or \\.

Phase 5: Integration

Exercise 11: Implement getNextToken()

Write the main function `Token getNextToken(FILE *fp)` that integrates all previous parts. It should skip whitespace and comments, recognize identifiers/keywords, numbers, operators, delimiters, and literals, and return a token each time it is called.

Exercise 12: Complete the Lexer

In the `main()` function, repeatedly call `getNextToken(fp)` until the end of the file is reached. Print each token in a table showing its **Lexeme**, **Token Type**, **Value** (if applicable) and **Line Number**.

4. Test Cases

4.1. Example 01

- ***Input for Lexer :***

```
int main() {
    int x = 10;
    char c = 'A';
    if (x > 0) {
        // Print message
        x = x - 1;
        printf("Hello World");
        return c;
    }
}
```

- ***Expected Tokens (Lexer Output)***

Line	Lexeme	Token Type	Value
1	int	KEYWORD	-

1	main	IDENTIFIER	-
1	(	DELIMITER	-
1	)	DELIMITER	-
1	{	DELIMITER	-
2	int	KEYWORD	-
2	x	IDENTIFIER	-
2	=	OPERATOR	-
2	10	NUMBER	10
2	;	DELIMITER	-
3	char	KEYWORD	-
3	c	IDENTIFIER	-
3	=	OPERATOR	-
3	'A'	CHAR_LITERAL	A
3	;	DELIMITER	-
4	if	KEYWORD	-
4	(	DELIMITER	-
4	x	IDENTIFIER	-
4	>	OPERATOR	-
4	0	NUMBER	0
4	)	DELIMITER	-
4	{	DELIMITER	-
6	x	IDENTIFIER	-
6	=	OPERATOR	-
6	x	IDENTIFIER	-
6	-	OPERATOR	-
6	1	NUMBER	1
6	;	DELIMITER	-
7	printf	IDENTIFIER	-
7	(	DELIMITER	-
7	"Hello World"	STRING_LITERAL	Hello World
7	)	DELIMITER	-
7	;	DELIMITER	-
8	return	KEYWORD	-
8	c	IDENTIFIER	-
8	;	DELIMITER	-
9	}	DELIMITER	-
10	}	DELIMITER	-

4.2. Example 02

- ***Input for Lexer :***

```
int 123abc = 5;
char d = 'Z;
```

```
float @value = 3.14;
```

➢ Runtime Error Messages (during scanning)

```
Error: Invalid identifier '123abc' at line 1
Error: Unterminated character literal at line 2
Error: Invalid symbol '@' at line 3
Error: Unsupported number format '3.14' at line 3
```

➢ *Expected Tokens (Lexer Output)*

Line	Lexeme	Token Type	Value
1	int	KEYWORD	-
1	123abc	***ERROR***	Invalid identifier
1	=	OPERATOR	-
1	5	NUMBER	5
1	;	DELIMITER	-
2	char	KEYWORD	-
2	d	IDENTIFIER	-
2	=	OPERATOR	-
2	'Z;	***ERROR***	Unterminated char literal
3	float	IDENTIFIER	-
3	@	***ERROR***	Invalid symbol
3	value	IDENTIFIER	-
3	=	OPERATOR	-
3	3.14	***ERROR***	Floating numbers not supported
3	;	DELIMITER	-

Notes:

- 123abc and 'Z; are flagged as ERROR tokens with descriptive messages.
- @ is reported as an invalid symbol.
- 3.14 is an unsupported numeric format (since the lexer only handles integers for now).
- The lexer continues scanning after errors instead of stopping, so the full table is still generated.