Assignment 01: Lexical Analyzer Comparison Report

Team: 23I-0604, 23I-0508
Section: CS-D

February 15, 2026

1 Side-by-Side Output Comparison

The following tables compare the token generation of the Manual Scanner (Java) and the JFlex Scanner on identical test files.

1.1 Test Case 1: Valid Code (`test1.lang`)

Both scanners successfully identified all tokens correctly. The output is identical for valid inputs.

Manual Scanner Output	JFlex Scanner Output

`<KEYWORD, "start", Line: 1...>`
`<KEYWORD, "declare", Line: 2...>`
`<IDENTIFIER, "X", Line: 2...>`
`<STRING_LITERAL, ""Hello\nWorld"", ...>`
`<FLOAT_LITERAL, "3.14", ...>`
`<EOF, "EOF", Line: 13...>`
	`<KEYWORD, "start", Line: 1...>`
	`<KEYWORD, "declare", Line: 2...>`
	`<IDENTIFIER, "X", Line: 2...>`
	`<STRING_LITERAL, ""Hello\nWorld"", ...>`
	`<FLOAT_LITERAL, "3.14", ...>`
	`<EOF, "EOF", Line: 13...>`

Status: PASS	**Status: PASS**

Table 1: Output comparison for valid code

1.2 Test Case 4: Error Handling (`test4.lang`)

Significant differences arise in how errors are reported. The JFlex scanner uses specific regex rules to identify *types* of errors, while the Manual scanner relies on fallback logic.

Feature / Input	Scanner Behavior
Lowercase Identifier `declare count`	**Manual:** `[ERROR] INVALID_TOKEN` *(Generic error: token not recognized)* **JFlex:** `[ERROR] INVALID_IDENTIFIER` *Reason: Identifiers must start with Uppercase.*
Bad Float `1.2345678`	**Manual:** `[ERROR] INVALID_TOKEN` *(Generic error)* **JFlex:** `[ERROR] MALFORMED_LITERAL` *Reason: Float exceeds 6 decimal places.*
Bad Char Literal `'TooLong'`	**Manual:** Splits into 3 tokens: 1. `MALFORMED_LITERAL` ("'Too") 2. `IDENTIFIER` ("Long") 3. `MALFORMED_LITERAL` ("'") **JFlex:** Catches as single error: `[ERROR] MALFORMED_LITERAL` *Reason: Character literal too long.*

2 Explanation of Differences

2.1 A. Error Specificity

- **Manual Scanner:** The manual implementation uses a "longest match" approach. If no valid token matches, it falls through to an error state. It categorizes these simply as `INVALID_TOKEN` because it detects *that* a match failed, but not *why*.
- **JFlex Scanner:** In JFlex, we defined specific "Error Rules" (e.g., `{LOWER}[a-z0-9]*`). This allows the scanner to affirmatively match an invalid pattern and provide a precise reason to the user.

2.2 B. Token Recovery

- **Manual Scanner:** In cases like `'TooLong'`, the manual scanner stopped parsing the character literal as soon as it became invalid (length > 1), leaving the remaining characters (`Long`) to be parsed as a separate Identifier. This causes cascading errors.
- **JFlex Scanner:** The regex `MALFORMED_CHAR` allowed JFlex to consume the entire invalid chunk as a single error token. This prevents cascading errors and provides a cleaner error log.

3 Performance Comparison

3.1 Theoretical Analysis

3.1.1 1. Manual Scanner (Iterative Approach)

- **Mechanism:** The manual scanner iterates through arrays of Keywords and Operators for every potential token match (e.g., using `startsWith()` in a loop).
- **Complexity:** $O(T \times K)$, where T is the number of tokens and K is the number of keywords/rules. As the number of keywords grows, the scanner becomes slower.
- **Overhead:** High overhead due to repeated string comparisons and object creation during matching.

3.1.2 2. JFlex Scanner (DFA Approach)

- **Mechanism:** JFlex compiles the Regular Expressions into a **Deterministic Finite Automaton (DFA)**. This is a state-transition table.
- **Complexity:** $O(L)$, where L is the length of the source code (Linear Time).
- **Efficiency:** The time to recognize a token depends only on the length of the token, not on the number of keywords in the language. It is significantly faster and more scalable for large source files.

3.2 Conclusion

While both scanners produce correct outputs for valid code, the **JFlex Scanner** is superior in terms of:

1. **Speed:** Due to DFA optimization ($O(L)$ vs $O(T \times K)$).
2. **Error Reporting:** Due to the ability to define "catch" rules for common mistakes.
3. **Maintainability:** Adding a new keyword requires just one line in JFlex, compared to updating multiple arrays and logic blocks in Java.