CS4031 - Compiler Construction

Assignment 01

Lexical Analyzer Implementation

Spring 2026

Important Information

Total Marks: 100 + 10 Bonus — **Deadline:** As per GCR
Submission: RollNumber1-RollNumber2-Section.zip (e.g., 22i1234-22i5678-A.zip)

1 Submission Instructions

1. Groups of **2 students only** NOTE:This group will remain the same for all future assignments and project.)
2. Submit **only .ZIP files** on Google Classroom (Classroom Tab, not Lab)
3. File naming: `RollNo1-RollNo2-Section.zip`
4. Test your program on multiple machines before submission
5. Submit **3 hours before deadline** to avoid last-minute issues
6. Only **Java** programming language allowed
7. No built-in compiler libraries (except JFlex for Part 2)
8. Syntax errors = zero marks for that component
9. Poor viva performance can result in up to 50% deduction
10. **Plagiarism = ZERO marks**

2 Introduction

In this assignment, you will implement a **Lexical Analyzer (Scanner)** for a custom programming language. The lexical analyzer is the first phase of a compiler that reads source code and breaks it into meaningful units called **tokens**. You will implement the scanner in **two ways**:

1. **Manual Implementation:** Using Regular Expressions, NFA, and DFA
2. **JFlex Implementation:** Using JFlex tool to validate your manual implementation

3 Regular Expression Specifications

Token Patterns - Must Implement All

Your scanner must recognize the following token types using exact regex patterns:

3.1 Keywords (Case-Sensitive, Exact Match)

Regex: `(start|finish|loop|condition|declare|output|input|function|return|break|continue|else)`
Required Keywords: start, finish, loop, condition, declare, output, input, function, return, break, continue, else

3.2 Identifiers

Regex: `[A-Z][a-z0-9_]{0,30}` **Rules:** Must start with uppercase letter (A-Z), followed by lowercase letters, digits, or underscores. Maximum 31 characters total. **Valid:** Count, Variable_name, X, Total_sum_2024
Invalid: count, _Variable, 2Count, myVariable

3.3 Integer Literals

Regex: `[+-]?[0-9]+` **Valid:** 42, +100, -567, 0
Invalid: 12.34, 1,000

3.4 Floating-Point Literals

Regex: `[+-]?[0-9]+\.[0-9]{1,6}([eE][+-]?[0-9]+)?`
Valid: 3.14, +2.5, -0.123456, 1.5e10, 2.0E-3
Invalid: 3., .14, 1.2345678 (¿6 decimals)

3.5 String Literals

Regex: `"([^"\\\n]|\\["\\ntr])*"`
Escape Sequences: `\"`, `\\`, `\n`, `\t`, `\r`

3.6 Character Literals

Regex: `'([^'\\\n]|\\['\\ntr])'`

3.7 Boolean Literals

Regex: `(true|false)` (case-sensitive)

3.8 Operators

Arithmetic:	`(\*\*|[+\-*/%])`
	+, -, *, /, %, ** (exponentiation)
Relational:	`(==|!=|<=|>=|<|>)`
Logical:	`(&&||||!)`
Assignment:	`(\+=|-=|\*=|/=|=)`
Inc/Dec:	`(\+\+|--)`

3.9 Punctuators

Regex: `[(){}[\][,;:]]`
Characters: ( ) { } [ ] , ; :

3.10 Comments

Single-line: `##[^\n]*`
Multi-line: `#\*([^*]|\*+[^*#])*\*+#`

3.11 Whitespace

Regex: `[ \t\r\n]+` (skip but track line numbers)

3.12 Pattern Matching Priority

CRITICAL: Check patterns in this order to avoid ambiguity:

1. Multi-line comments
2. Single-line comments
3. Multi-character operators (**, ==, !=, ¡=, ¿=, &&, ——, ++, –, +=, -=, *=, /=)
4. Keywords
5. Boolean literals
6. Identifiers
7. Floating-point literals
8. Integer literals
9. String/character literals
10. Single-character operators
11. Punctuators
12. Whitespace

4 Part 1: Manual Scanner Implementation (60 Marks)

4.1 Task 1.1: Automata Design (15 Marks)

Create `Automata_Design.pdf` containing:

- Regular expressions for each token category (3 marks)
- NFA diagrams for the following token types (6 marks):
  - Integer literal (mandatory)
  - Floating-point literal (mandatory)
  - Identifier (mandatory)
  - Single-line comment (mandatory)
  - Any 3 additional token types of your choice
- Minimized DFA conversion with transition state tables for the same 7 token types (6 marks)
- Show NFA states, transitions, start/accepting states clearly
- No need to show intermediate conversion steps - just final NFA, minimized DFA, and transition tables
- You may draw by hand or use tools (JFLAP, draw.io, etc.)

4.2 Task 1.2: Scanner Implementation (45 Marks)

Implement `ManualScanner.java` that:

A. Token Recognition (25 marks)

- Recognizes all token types from Section 3
- Uses DFA-based matching
- Applies longest match principle

- Handles operator precedence correctly

B. Pre-processing (5 marks)

- Removes unnecessary whitespace
- Preserves whitespace in string literals
- Tracks line and column numbers accurately

C. Token Output (5 marks)
Create `Token.java` class with: TokenType, lexeme, line number, column number
Output format: `<KEYWORD, "start", Line:  1, Col:  1>`
D. Statistics (5 marks)
Display: Total tokens, count per token type, lines processed, comments removed
E. Symbol Table (5 marks)
Create `SymbolTable.java` that stores: identifier name, type, first occurrence, frequency

5 Part 2: JFlex Implementation (30 Marks)

5.1 Task 2.1: JFlex Specification (20 Marks)

Create `Scanner.flex` with:

- User code section (imports, helper methods)
- Macro definitions for patterns (DIGIT, LETTER, etc.)
- Lexical rules matching all patterns from Section 3
- Same pattern matching priority as manual implementation
- Proper handling of comments, whitespace, and errors

5.2 Task 2.2: Token Class (5 Marks)

Implement `Token.java` compatible with both scanners

5.3 Task 2.3: Comparison (5 Marks)

Create `Comparison.pdf` showing:

- Side-by-side outputs on same test files
- Explanation of any differences
- Performance comparison

6 Part 3: Error Handling (10 Marks)

Implement `ErrorHandler.java` that detects and reports:
Error Types (5 marks):

- Invalid characters (@, $, etc.)
- Malformed literals (multiple decimals, unterminated strings)
- Invalid identifiers (wrong starting character, exceeding length)
- Unclosed multi-line comments

Error Reporting (3 marks):
Format: Error type, line, column, lexeme, reason
Error Recovery (2 marks):
Skip to next valid token, continue scanning, report all errors

7 Deliverables

7.1 Folder Structure

```
22i1234-22i5678-A/
 src/
    ManualScanner.java, Token.java, TokenType.java
    SymbolTable.java, ErrorHandler.java
    Scanner.flex, Yylex.java
 docs/
    Automata_Design.pdf
    README.md
    LanguageGrammar.txt
    Comparison.pdf
 tests/
    test1.lang (all valid tokens)
    test2.lang (complex expressions)
    test3.lang (string/char with escapes)
    test4.lang (lexical errors)
    test5.lang (comments)
    TestResults.txt
 README.md
```

7.2 README.md Must Include

- Language name and file extension
- Complete keyword list with meanings
- Identifier rules and examples
- Literal formats with examples
- Operator list with precedence
- Comment syntax
- At least 3 sample programs
- Compilation and execution instructions
- Team members with roll numbers

8 Grading Rubric

Component	Marks
Part 1: Manual Implementation	**60**
RE, NFA, DFA Design	15
Token Recognition (all types)	25
Pre-processing & Whitespace	5
Token Output Format	5
Statistics Display	5
Symbol Table	5
Part 2: JFlex Implementation	**30**
JFlex Specification File	20
Token Class	5
Output Comparison	5
Part 3: Error Handling	**10**
Error Detection	5
Error Reporting	3
Error Recovery	2
Deductions	
Inadequate README	-10
Poor code quality/comments	-5
Total	**100**

9 Bonus Tasks (10 Marks)

1. **GitHub Repository (3 marks):** Upload project and maintain commit history
2. **Nested Multi-line Comments (3 marks):** Support properly nested comments
3. **Advanced String Features (2 marks):** Multi-line strings, Unicode escapes
4. **DFA Minimization (2 marks):** Implement and document minimization algorithm

10 Demo and Viva

Warning: Poor viva can result in 50% deduction or zero marks

Demo Requirements:

- Run scanner on provided test cases
- Live code modification (add keyword, modify rules, fix bugs)
- Explain automata designs and transitions

Viva Topics:

- NFA vs DFA differences
- Regex to NFA conversion
- DFA minimization process
- Longest match principle
- Operator precedence handling
- JFlex working mechanism
- Your implementation decisions

11 Important Notes

- Start early - assignment requires 1-1.5 weeks
- Define unique file extension for your language
- Test on multiple machines before submission
- Both scanners must produce identical output
- Be prepared to modify code during demo
- Understand every line of code you submit

Resources:

- `https://www.geeksforgeeks.org/introduction-of-compiler-design/`
- `https://jflex.de/manual.html`
- Course textbook: Dragon Book (Aho et al.)

Best of luck with your assignment!