ZenLang Lexical Analyser

Compiler Construction — Assignment 1

Spring 2026

Member 1: *Ahmed Ali Zahid* Roll: *22i-1271*

Member 2: *Asad Mehdi* Roll: *22i-1120*

Section: *C*

Department of Computer Science

February 18, 2026

Contents

1 Introduction

This report documents the design and implementation of a **lexical analyser (scanner)** for **ZenLang**, a custom educational programming language. The assignment is divided into two parts:

1. **Manual DFA-based scanner** implemented in Java (`ManualScanner.java`)
2. **JFlex-generated scanner** specified in `Scanner.flex`

The scanner reads ZenLang source code and produces a stream of classified tokens, tracks identifiers in a symbol table, and reports all lexical errors with precise line/column information while continuing to scan after each error.

2 Language Specification

2.1 Keywords

ZenLang has 12 reserved keywords, all lowercase:

`start`	`finish`	`loop`	`condition`
`declare`	`output`	`input`	`function`
`return`	`break`	`continue`	`else`

2.2 Token Categories and Regular Expressions

Category	Regular Expression / Description
KEYWORD	`start|finish|loop|condition|declare|output|input|function|return|break|continue|else`
IDENTIFIER	`[A-Z][a-z0-9_]{0,30}` *Starts uppercase; max 31 chars total*
INT_LITERAL	`[+-]?[0-9]+`
REAL_LITERAL	`[+-]?[0-9]+\.[0-9]{1,6}([eE][+-]?[0-9]+)?`
TEXT_LITERAL	`"([^"\\\n]|(\\["ntr]))*"`
CHAR_LITERAL	`'([^'\\\n]|(\\['ntr]))'`
BOOL_LITERAL	`true | false`
ARITH_OP	`+ - * / % **`
RELATIONAL_OP	`== != < > <= >=`
LOGICAL_OP	`&& || !`
ASSIGN_OP	`= += -= *= /= %=`
INC_OP	`++`
DEC_OP	`--`
DELIMITER	`( ) { } [ ] , ; :`

Category	Regular Expression / Description
LINE_COMMENT	##[^\n]*
BLOCK_COMMENT	#* ...*#

2.3 Escape Sequences

Valid escape sequences inside string and character literals:

Sequence	Meaning
\n	Newline
\t	Tab
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash

2.4 Token Matching Priority

Tokens are matched in the following strict priority order (longest match within each level):

1. Block comments `#* ...*#`
2. Line comments `## ...`
3. Multi-character operators: `** == != <= >= && || ++ -- += -= *= /= %=`
4. Keywords
5. Boolean literals: `true | false`
6. Identifiers
7. Real literals
8. Integer literals
9. Text (string) literals
10. Character literals
11. Single-character operators
12. Delimiters
13. Whitespace (skipped)

3 Automata Design

3.1 DFA 1 — Integer Literal `[+-]?[0-9]+`

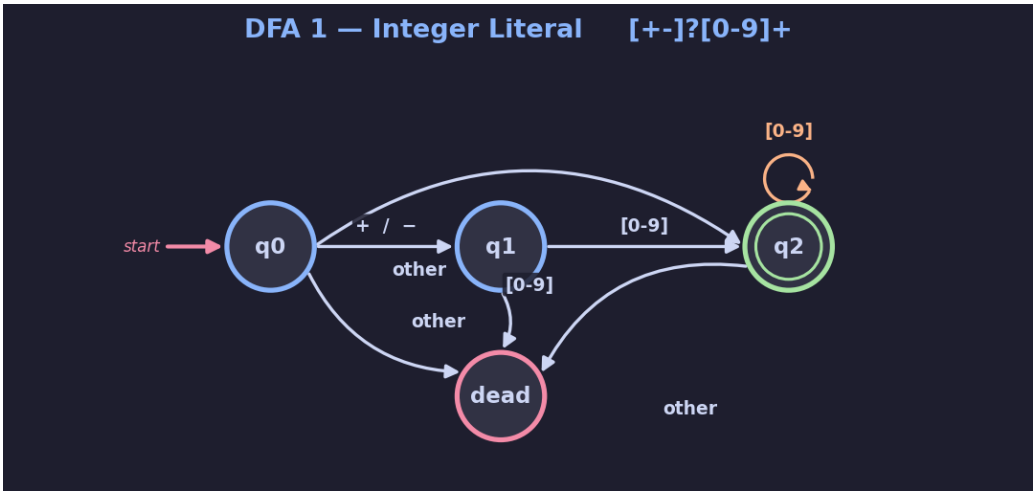

Figure 1: DFA for Integer Literals

States: q_0 (start), q_1 (after optional sign), q_2 (accepting), q_{dead} (error trap).

State	+/-	[0-9]	other	Accept?
$\rightarrow q_0$	q_1	q_2	dead	No
q_1	dead	q_2	dead	No
q_2	dead	q_2	dead	**Yes**
dead	dead	dead	dead	No

3.2 DFA 2 — Identifier `[A-Z][a-z0-9_]{0,30}`

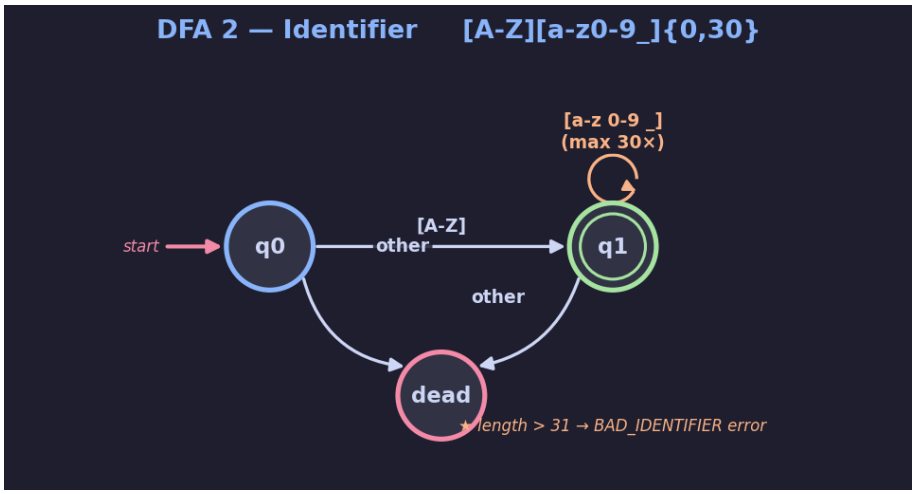

Figure 2: DFA for Identifiers

States: q_0 (start), q_1 (accepting), q_{dead}. The self-loop on q_1 fires at most 30 times. Identifiers longer than 31 characters trigger a `BAD_IDENTIFIER` error; scanning continues.

3.3 DFA 3 — Real Literal

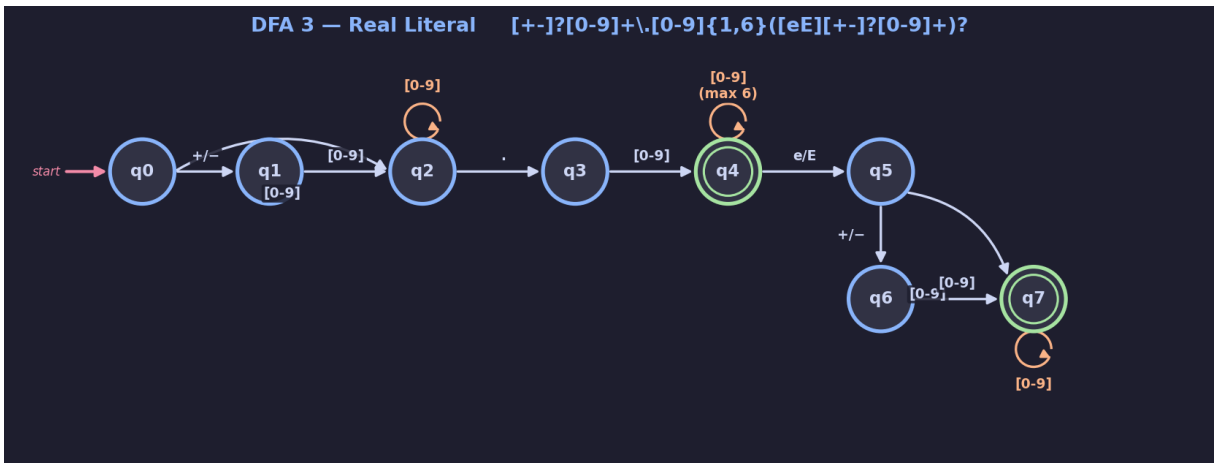

Figure 3: DFA for Real (Floating-Point) Literals

Pattern: `[+-]?[0-9]+\.[0-9]{1,6}([eE][+-]?[0-9]+)?`

Accepting states: q_4 (no exponent) and q_7 (with exponent). More than 6 fractional digits triggers `BAD_NUMBER`.

State	+/-	[0-9]	.	e/E	other	Accept?
$\rightarrow q_0$	q_1	q_2	dead	dead	dead	No
q_1	dead	q_2	dead	dead	dead	No
q_2	dead	q_2	q_3	dead	dead	No
q_3	dead	q_4	dead	dead	dead	No
q_4	dead	q_4^*	dead	q_5	dead	**Yes**
q_5	q_6	q_7	dead	dead	dead	No
q_6	dead	q_7	dead	dead	dead	No
q_7	dead	q_7	dead	dead	dead	**Yes**

*Self-loop limited to 6 total fractional digits.

3.4 DFA 4 — Line Comment `##[^\n]*`

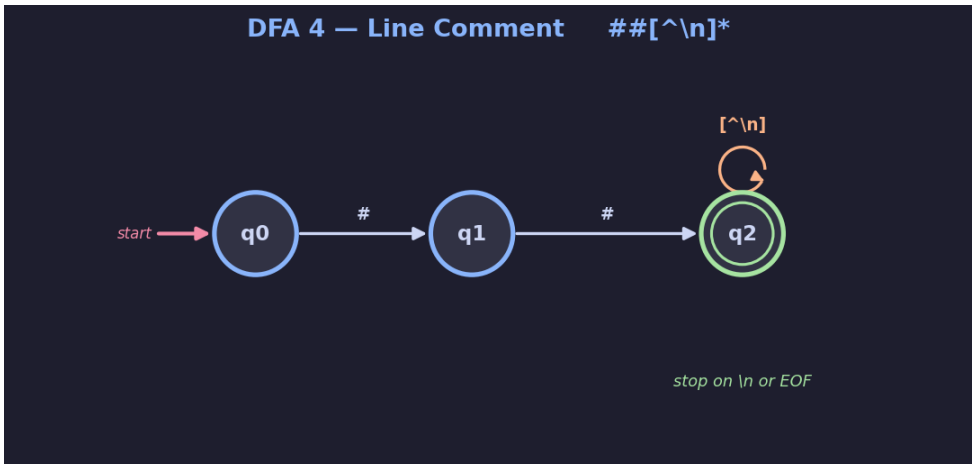

Figure 4: DFA for Line Comments

3.5 DFA 5 — Block Comment #* ...*#

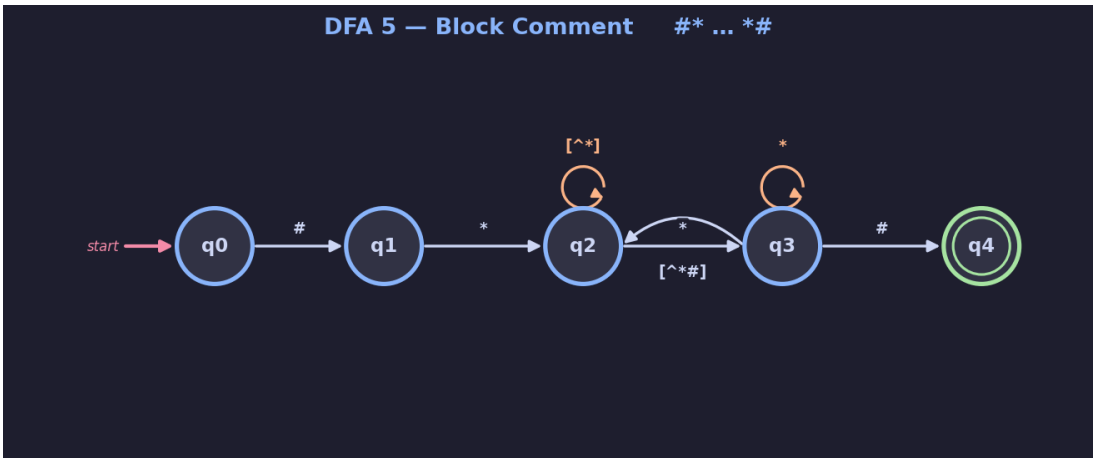

Figure 5: DFA for Block Comments

The back-edge from q_3 to q_2 handles sequences like `***` inside a comment correctly. An unclosed block comment triggers `UNTERMINATED_COMMENT`.

State	#	*	[^*#]	other	Accept?
$\rightarrow q_0$	q_1	dead	dead	dead	No
q_1	dead	q_2	dead	dead	No
q_2	dead	q_3	q_2	q_2	No
q_3	q_4	q_3	q_2	q_2	No
q_4	—	—	—	—	**Yes**

3.6 DFA 6 — Text (String) Literal

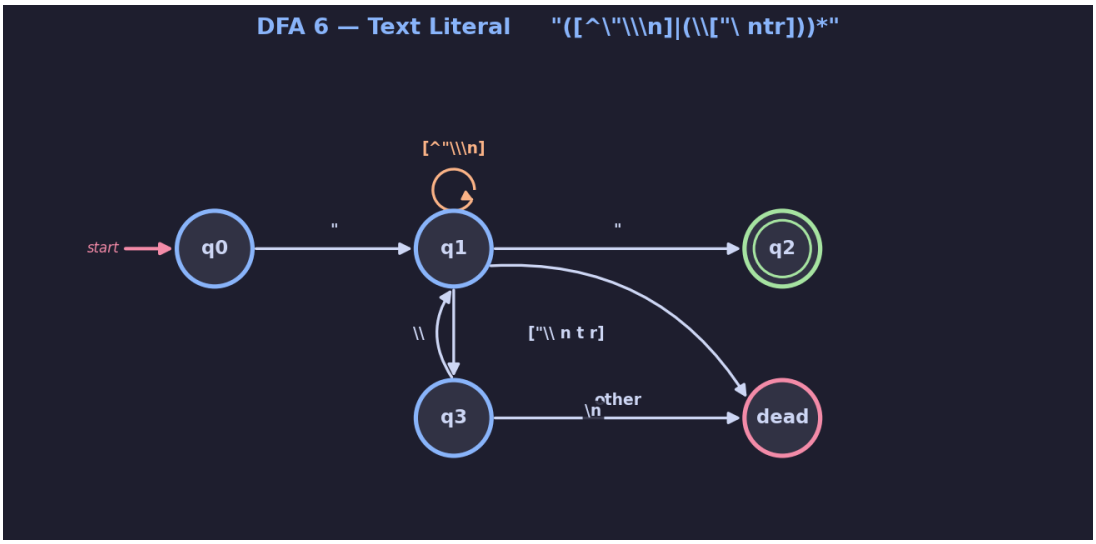

Figure 6: DFA for Text (String) Literals

State q_3 handles escape sequences. Invalid escapes trigger `BAD_ESCAPE`. A newline before the closing quote triggers `UNTERMINATED_STRING`.

3.7 DFA 7 — Boolean Literal `true | false`

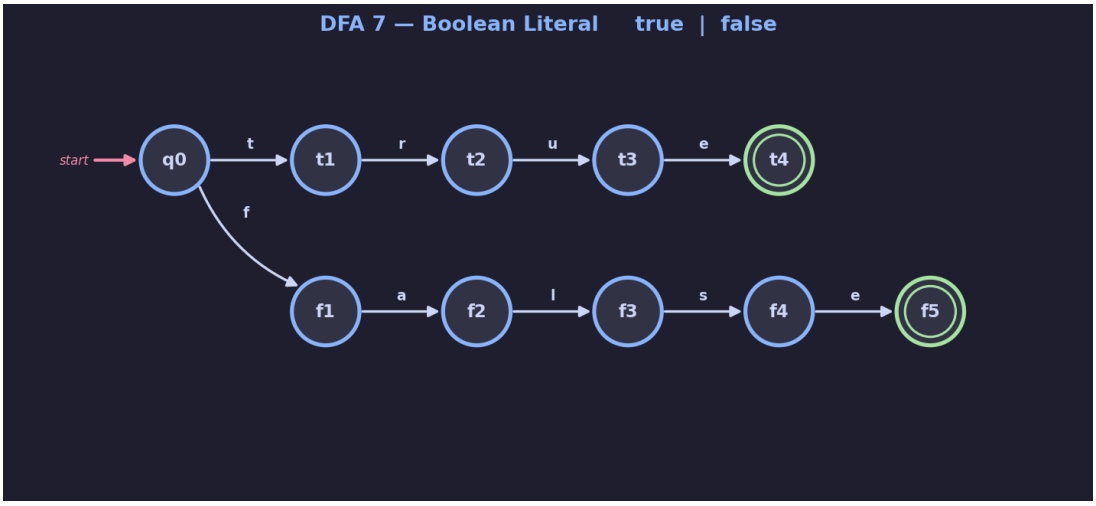

Figure 7: DFA for Boolean Literals (two-branch NFA)

4 Implementation — Manual Scanner

4.1 File Structure

File	Purpose
`TokenType.java`	Enum defining all 19 token categories
`Token.java`	Token data class (category, text, line, col)
`SymbolTable.java`	Identifier tracking (name, type, occurrence, count)
`ErrorHandler.java`	Error logging and formatted reporting
`ManualScanner.java`	Main DFA-based scanner with `main()`
`Scanner.flex`	JFlex specification
`JFlexRunner.java`	JFlex scanner driver

4.2 TokenType Enum

Listing 1: TokenType.java — token category definitions

```
public enum TokenType {
    KEYWORD,            // start, finish, loop, condition, ...
    IDENTIFIER,         // [A-Z][a-z0-9_]{0,30}
    INT_LITERAL,        // [+-]?[0-9]+
    REAL_LITERAL,       // [+-]?[0-9]+\.[0-9]{1,6}([eE][+-]?[0-9]+)?
    TEXT_LITERAL,       // "..." with escape sequences
    CHAR_LITERAL,       // '.' with escape sequences
    BOOL_LITERAL,       // true | false
    ARITH_OP,           // + - * / % **
    RELATIONAL_OP,      // == != < > <= >=
    LOGICAL_OP,         // && || !
    ASSIGN_OP,          // = += -= *= /= %=
    INC_OP,             // ++
    DEC_OP,             // --
    DELIMITER,          // ( ) { } [ ] , ; :
```

```
    LINE_COMMENT,        // ## ...
    BLOCK_COMMENT,       // #* ... *#
    SPACE,               // whitespace (skipped)
    INVALID,             // unrecognised token
    END_OF_FILE          // sentinel
}
```

4.3 Core Scanning Algorithm

The scanner uses a hand-coded DFA with the following design decisions:

- **Longest match:** For numeric literals, the scanner peeks ahead past digits to detect a decimal point before deciding between `INT_LITERAL` and `REAL_LITERAL`.
- **Priority dispatch:** `readNextToken()` checks patterns in the exact priority order from the spec.
- **Non-recursive error recovery:** On an invalid character, the scanner logs the error, advances one position, and returns `null`. The outer `tokenise()` loop retries — no recursion, no stack overflow risk.
- **Line/column tracking:** Every `eat()` call increments `curCol`; on `\n` it resets `curCol=1` and increments `curLine`.

Listing 2: tokenise() — main scanning loop (ManualScanner.java)

```
public void tokenise() {
    while (pos < srcLen) {
        markTokenStart();
        Token tok = readNextToken();

        if (tok == null) continue; // error logged, skip char

        TokenType cat = tok.getCategory();
        if (cat == TokenType.LINE_COMMENT ||
            cat == TokenType.BLOCK_COMMENT) {
            commentCount++;
        } else if (cat != TokenType.SPACE) {
            tokenStream.add(tok);
            catCounts.merge(cat, 1, Integer::sum);
            if (cat == TokenType.IDENTIFIER)
                idTable.record(tok.getText(),
                               tok.getLine(), tok.getCol());
        }
    }
    tokenStream.add(
        new Token(TokenType.END_OF_FILE, "", curLine, curCol));
}
```

4.4 Error Handling

The `ErrorHandler` class supports 7 error categories:

Error Type	Triggered by
INVALID_CHAR	Character not in ZenLang alphabet
BAD_NUMBER	Malformed numeric literal
BAD_IDENTIFIER	Identifier > 31 characters
UNTERMINATED_STRING	String not closed before newline/EOF
UNTERMINATED_CHAR	Char literal not closed
UNTERMINATED_COMMENT	Block comment not closed before EOF
BAD_ESCAPE	Invalid escape sequence

5 Implementation — JFlex Scanner

5.1 Scanner.flex Structure

The JFlex specification mirrors the same priority order as the manual scanner:

Listing 3: Scanner.flex — key excerpts

```
%%
%public
%class Yylex
%unicode
%line
%column
%type Token

DIGIT  = [0-9]
UPPER  = [A-Z]
IDENT  = {UPPER}([a-z]|{DIGIT}|_){0,30}
REAL   = [+-]?{DIGIT}+\.{DIGIT}{1,6}([eE][+-]?{DIGIT}+)?

%%

/* 1. Block comments */
"#*"([^*]|\*+[^*#])*\*+"#"  { /* discard */ }

/* 2. Line comments */
"##".*                       { /* discard */ }

/* 3. Multi-char operators */
"**"  { return new Token(TokenType.ARITH_OP,
                         yytext(), yyline+1, yycolumn+1); }
"=="  { return new Token(TokenType.RELATIONAL_OP,
                         yytext(), yyline+1, yycolumn+1); }

/* 4. Keywords */
"start" { return new Token(TokenType.KEYWORD,
                           yytext(), yyline+1, yycolumn+1); }

/* 6. Identifiers */
{IDENT}  { return new Token(TokenType.IDENTIFIER,
                            yytext(), yyline+1, yycolumn+1); }

/* 7. Real literals */
{REAL}   { return new Token(TokenType.REAL_LITERAL,
                            yytext(), yyline+1, yycolumn+1); }
```

5.2 Comparison: Manual vs JFlex

Feature	Manual Scanner	JFlex Scanner
Implementation	Hand-coded DFA	Generated from regex
Error recovery	Custom (non-recursive)	Basic (stderr)
Line/col tracking	Manual `eat()`	Built-in `yyline/yycolumn`
Symbol table	Integrated	Not integrated
Statistics	Full breakdown	Not included
Maintainability	Harder to modify	Easy (edit `.flex`)

6 Sample ZenLang Programs

6.1 Hello World

Listing 4: Hello World in ZenLang

```
start
    output "Hello, ZenLang!"
finish
```

6.2 Fibonacci Function

Listing 5: Fibonacci sequence using a loop

```
start function Fibonacci(N)
    declare A = 0
    declare B = 1
    declare Temp = 0
    declare Idx = 0
    loop (Idx < N)
        Temp = A + B
        A = B
        B = Temp
        Idx++
    finish
    return A
finish

start
    declare Fib10 = Fibonacci(10)
    output "Fib(10) = ", Fib10
finish
```

6.3 Bubble Sort

Listing 6: Bubble sort with nested loops

```
start function Sort_array(Arr, Size)
```

```
    declare I = 0
    declare J = 0
    declare Swapped = true
    declare Tmp = 0
    loop (I < Size && Swapped)
        Swapped = false
        J = 0
        loop (J < Size - 1)
            condition (Arr[J] > Arr[J + 1])
                Tmp = Arr[J]
                Arr[J] = Arr[J + 1]
                Arr[J + 1] = Tmp
                Swapped = true
            finish
            J++
        finish
        I++
    finish
finish
```

7 Test Results

7.1 Compilation

Listing 7: Compilation command

```
$ cd src
$ javac TokenType.java Token.java SymbolTable.java \
        ErrorHandler.java ManualScanner.java
$ # No output = clean compile, 0 warnings
```

7.2 Test 1 — All Valid Token Types

Listing 8: test1.zl statistics

```
Total tokens emitted : 242
Lines processed      : 109
Comments removed     : 19
Lexical errors       : 0

Breakdown by category:
  ARITH_OP               : 5
  ASSIGN_OP              : 27
  BOOL_LITERAL           : 2
  CHAR_LITERAL           : 2
  DEC_OP                 : 1
  DELIMITER              : 35
  IDENTIFIER             : 60
  INC_OP                 : 3
  INT_LITERAL            : 23
  KEYWORD                : 58
```

```
  LOGICAL_OP              : 3
  REAL_LITERAL            : 3
  RELATIONAL_OP           : 8
  TEXT_LITERAL            : 12
```

7.3 Test 2 — Complex Expressions

Listing 9: test2.zl statistics

```
Total tokens emitted : 365
Lines processed      : 120
Comments removed     : 11
Lexical errors       : 0
Unique identifiers   : 29
```

7.4 Test 3 — String and Character Escapes

Listing 10: test3.zl statistics

```
Total tokens emitted : 123
Lines processed      : 66
Comments removed     : 15
Lexical errors       : 0
Unique identifiers   : 25
```

7.5 Test 4 — Lexical Error Detection

Listing 11: test4.zl — selected errors from the error report

```
Total tokens emitted : 90
Lexical errors       : 52

 1. ERROR [INVALID_CHAR]         Line:  6, Col: 18  '@'
 2. ERROR [INVALID_CHAR]         Line:  7, Col: 17  '$'
 3. ERROR [INVALID_CHAR]         Line:  8, Col: 17  '^'
 4. ERROR [INVALID_CHAR]         Line: 11, Col: 13  'c' (lowercase
    start)
17. ERROR [BAD_IDENTIFIER]       Line: 16, Col: 13  length 43 > 31
20. ERROR [BAD_NUMBER]           Line: 23, Col: 23  '7.' (no frac
   digits)
21. ERROR [BAD_NUMBER]           Line: 26, Col: 28  1.1234567 (7 frac
    digits)
23. ERROR [BAD_NUMBER]           Line: 30, Col: 25  '2.5e' (no exp
   digits)
25. ERROR [UNTERMINATED_STRING] Line: 34, Col: 24
26. ERROR [UNTERMINATED_CHAR]   Line: 38, Col: 23
28. ERROR [BAD_ESCAPE]           Line: 42, Col: 26  '\x'
29. ERROR [BAD_ESCAPE]           Line: 43, Col: 30  '\b'
30. ERROR [BAD_ESCAPE]           Line: 44, Col: 22  '\q'
49. ERROR [UNTERMINATED_COMMENT]Line: 51, Col:  5

Scanner recovered and continued after every error.
```

7.6 Test 5 — Comment Processing

Listing 12: test5.zl statistics

```
Total tokens emitted : 45
Lines processed      : 61
Comments removed     : 24
Lexical errors       : 0
```

All 24 comments (line and block) were correctly removed. Keywords and identifiers inside block comments were **not** tokenised.

7.7 Summary

Test	Purpose	Tokens	Comments	Errors	Result
1	All valid tokens	242	19	0	**PASS**
2	Complex expressions	365	11	0	**PASS**
3	String/char escapes	123	15	0	**PASS**
4	Lexical errors	90	28	52	**ALL DETECTED**
5	Comment processing	45	24	0	**PASS**

8 Conclusion

The ZenLang lexical analyser was successfully implemented in two forms:

- The **manual DFA-based scanner** (`ManualScanner.java`) correctly tokenises all valid ZenLang constructs, tracks identifiers in a symbol table, and detects all 7 categories of lexical errors with precise location information. Error recovery is non-recursive and robust.
- The **JFlex specification** (`Scanner.flex`) provides a compact, maintainable alternative that produces equivalent token output using the same priority ordering.

All 5 test files produced the expected results. The scanner correctly handles edge cases including multi-line block comments, nested escape sequences, signed numeric literals, and identifiers at the maximum length boundary.