Week 1: 4th September 2025

INNOPOLIS UNIVERSITY

[F25] Compiler Construction

Team 806

- Timofey Ivlev
- George Selivanov

We glad to present you our project:

Javdin

Java **d**ynamic **in**terpreter with a Bison-based parser, for Dynamic academic language

Project's technology stack

- Source Language **D**
- Implementation language **Java**
- Parser development tool **Bison-based CUP (Construction of Useful Parsers)**
- Target platform **JVM**
- Other tools and versions:
  - Java 17
  - Maven 3.6
  - JUnit 5 for testing
  - AssertJ for assertions
  - JaCoCo for code coverage
  - CUP and JFlex for parser generation

Tests:

Here is how we declared Tokens:

```
// record = immutable class
public record Token(
    TokenType type,
    String value,
    int line,
    int column
) {

    public Token(TokenType type, int line, int column) {
        this(type, "", line, column);
    }

    @Override
    public String toString() {
        if (value.isEmpty()) {
            return String.format("%s at %d:%d", type, line, column);
        }
        return String.format("%s('%s') at %d:%d", type, value, line,
column);
    }
}

public enum TokenType {
    // Literals
    INTEGER,
    REAL,
    BOOL,
    STRING,

    // Identifiers
    IDENTIFIER,

    // Keywords
    VAR,
    IF,
    ELSE,
    WHILE,
    FOR,
    FUNCTION,
    RETURN,
    PRINT,
    INPUT,
    TRUE,
    FALSE,
    LAMBDA,
    BREAK,
    CONTINUE,

    // Operators
    PLUS,           // +
    MINUS,          // -
```

```
    MULTIPLY,       // *
    DIVIDE,         // /
    MODULO,         // %
    ASSIGN,         // =
    EQUAL,          // ==
    NOT_EQUAL,      // !=
    LESS_THAN,      // <
    LESS_EQUAL,     // <=
    GREATER_THAN,   // >
    GREATER_EQUAL,  // >=
    AND,            // and
    OR,             // or
    NOT,            // not

    // Delimiters
    LEFT_PAREN,     // (
    RIGHT_PAREN,    // )
    LEFT_BRACE,     // {
    RIGHT_BRACE,    // }
    LEFT_BRACKET,   // [
    RIGHT_BRACKET,  // ]
    SEMICOLON,      // ;
    COMMA,          // ,
    DOT,            // .
    COLON,          // :
    ARROW,          // ->

    // Special
    NEWLINE,
    EOF,
    UNKNOWN
}
```

Here is template class for lexer testing:

```
class LexerTest {

    private Lexer lexer;

    @Test
    void testBasicTokens() {
        lexer = new Lexer("var x = 42;");

        Token token1 = lexer.nextToken();
        assertThat(token1.type()).isEqualTo(TokenType.VAR);

        Token token2 = lexer.nextToken();
        assertThat(token2.type()).isEqualTo(TokenType.IDENTIFIER);
        assertThat(token2.value()).isEqualTo("x");

        Token token3 = lexer.nextToken();
        assertThat(token3.type()).isEqualTo(TokenType.ASSIGN);
```

```
        Token token4 = lexer.nextToken();
        assertThat(token4.type()).isEqualTo(TokenType.INTEGER);
        assertThat(token4.value()).isEqualTo("42");

        Token token5 = lexer.nextToken();
        assertThat(token5.type()).isEqualTo(TokenType.SEMICOLON);

        Token token6 = lexer.nextToken();
        assertThat(token6.type()).isEqualTo(TokenType.EOF);
    }

    @ParameterizedTest
    @CsvSource({
        "123, INTEGER, 123",
        "3.14, REAL, 3.14",
        "true, TRUE, true",
        "false, FALSE, false",
        "\"hello\", STRING, hello",
        "identifier, IDENTIFIER, identifier"
    })
    void testLiterals(String input, String expectedType, String
expectedValue) {
        lexer = new Lexer(input);
        Token token = lexer.nextToken();

        assertThat(token.type().name()).isEqualTo(expectedType);
        assertThat(token.value()).isEqualTo(expectedValue);
    }

    @Test
    void testKeywords() {
        lexer = new Lexer("if else while for function return print");

        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.IF);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.ELSE);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.WHILE);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.FOR);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.FUNCTION);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.RETURN);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.PRINT);
    }

    @Test
    void testOperators() {
        lexer = new Lexer("+ - * / == != <= >= = < >");

        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.PLUS);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.MINUS);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.MULTIPLY);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.DIVIDE);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.EQUAL);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.NOT_EQUAL);
```

```
assertThat(lexer.nextToken().type()).isEqualTo(TokenType.LESS_EQUAL);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.GREATER_EQUAL);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.ASSIGN);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.LESS_THAN);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.GREATER_THAN);
    }

    @Test
    void testStringLiterals() {
        lexer = new Lexer("\"Hello, World!\" \"\" \"Line\\nBreak\"");

        Token token1 = lexer.nextToken();
        assertThat(token1.type()).isEqualTo(TokenType.STRING);
        assertThat(token1.value()).isEqualTo("Hello, World!");

        Token token2 = lexer.nextToken();
        assertThat(token2.type()).isEqualTo(TokenType.STRING);
        assertThat(token2.value()).isEqualTo("");

        Token token3 = lexer.nextToken();
        assertThat(token3.type()).isEqualTo(TokenType.STRING);
        assertThat(token3.value()).isEqualTo("Line\nBreak");
    }

    @Test
    void testComments() {
        lexer = new Lexer("var x; // This is a comment\nvar y;");

        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.VAR);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.IDENTIFIER);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.SEMICOLON);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.NEWLINE);
        assertThat(lexer.nextToken().type()).isEqualTo(TokenType.VAR);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.IDENTIFIER);

assertThat(lexer.nextToken().type()).isEqualTo(TokenType.SEMICOLON);
    }

    @Test
    void testLineAndColumnTracking() {
        lexer = new Lexer("var\nx");

        Token token1 = lexer.nextToken();
        assertThat(token1.line()).isEqualTo(1);
        assertThat(token1.column()).isEqualTo(1);

        Token token2 = lexer.nextToken();
```

```
        assertThat(token2.line()).isEqualTo(1);
        assertThat(token2.column()).isEqualTo(4);

        Token token3 = lexer.nextToken();
        assertThat(token3.type()).isEqualTo(TokenType.NEWLINE);
        assertThat(token3.line()).isEqualTo(1);

        Token token4 = lexer.nextToken();
        assertThat(token4.line()).isEqualTo(2);
        assertThat(token4.column()).isEqualTo(1);
    }

    @Test
    void testLexicalException() {
        lexer = new Lexer("@");

        assertThatThrownBy(() -> lexer.nextToken())
            .isInstanceOf(LexicalException.class)
            .hasMessageContaining("Unexpected character: @");
    }
}
```