Javdin Parser

Course: [F25] Compiler Construction
Date: October 16, 2025
Team: 806

Team Information

Team 806

- **Timofey Ivlev**
- **George Selivanov**

Project: Javdin

Java **d**ynamic **in**terpreter - A dynamic language interpreter with a Bison-based parser

Technology Stack

Core Technologies

- **Source Language**: Project D (academic dynamic language)
- **Implementation Language**: Java 17
- **Parser Development Tool**: **CUP (Construction of Useful Parsers)**
  - Bison-based parser generator for Java
  - Generates LR parsers from grammar specifications
  - Website: https://www2.cs.tum.edu/projects/cup/

How it works:

1. Write grammar in `.cup` file
2. CUP generates Java parser code
3. Generated parser uses LR parsing algorithm
4. Parser creates AST nodes during parsing (but! we used custom ast nodes)

Our Grammar File: `src/main/resources/parser.cup` (~800 lines)

Build & Testing Tools

- **Build System**: Maven 3.6+
- **Testing Framework**: JUnit 5
- **Assertion Library**: AssertJ
- **Code Coverage**: JaCoCo
- **Lexer Generator**: JFlex (companion to CUP)

Target Platform

- **JVM** (Java Virtual Machine)
- Runs on any platform with Java 17+

Example Programs

Example 1: Recursive Factorial

```
// Factorial function with recursion
var factorial := func(n) is
    if n <= 1 then
        return 1
    else
        return n * factorial(n - 1)
    end
end

var result := factorial(5)
print result
```

Features demonstrated:

- Function literals (long form: `func(...) is ... end`)
- Recursion
- Control flow (`if-then-else`)
- Return statements
- Binary operators (`<=`, `-`, `*`)
- Function calls

```
./visualize-ast.sh presentation-example-1.d
```

Example 2: Array Processing

```
// Array and loop operations
var numbers := [1, 2, 3, 4, 5]
var sum := 0
var i := 0

for i in numbers loop
    sum := sum + i
end

print "Sum:", sum

// Calculate average
var average := sum / 5
print "Average:", average
```

Features demonstrated:

- Array literals (`[1, 2, 3, 4, 5]`)
- Variable declarations with initialization
- For-in loops
- String literals
- Multiple print arguments
- Arithmetic operations

```
./visualize-ast.sh presentation-example-2.d
```

Parser Implementation Details

Architecture Overview

1. Lexical Analysis (Tokenization)

- Input: Source code in .d files
- Tool: JFlex-generated lexer
- Output: Stream of tokens with position information
- Tokens include: keywords (IF, WHILE, VAR), identifiers, literals, operators, delimiters

2. Token Adaptation

- Component: LexerAdapter
- Purpose: Bridge between JFlex lexer and CUP parser
- Converts Token objects to CUP Symbol objects
- Maps token types to CUP terminal symbols

3. Syntactic Analysis (Parsing)

- Tool: CUP-generated LR parser
- Input: Token stream from LexerAdapter
- Grammar: 417 lines in parser.cup
- Algorithm: LALR(1) left-to-right parsing
- Output: Abstract Syntax Tree (AST)

4. AST Construction

- Strategy: Custom strongly-typed AST nodes (without XMLElement)
- Node hierarchy: 23 specialized classes extending StatementNode or ExpressionNode
- Each production rule creates specific AST node type
- All nodes are immutable with final fields
- Position tracking: Every node stores source line and column

Custom AST vs Alternative Approaches

We chose custom AST classes over CUP's auto-generated approach for several advantages:

Aspect	Custom AST	CUP XML (-xmlactions)
Type Safety	Compile-time type checking	Runtime string-based access
Performance	Direct field access, ~48 bytes/node	XML parsing overhead, ~120 bytes/node
Extensibility	Visitor pattern support	Requires XSLT/XQuery
Error Messages	compile-time errors	Generic runtime errors

Example Node Construction

```
RESULT = new IfNode(ileft, iright, condition, thenBlock, elseBlock);
```

AST Node Hierarchy

```
AstNode (interface)
│
├── ProgramNode (root, implements AstNode directly)
│
├── StatementNode (abstract)
│   │
│   ├── DeclarationNode
│   ├── AssignmentNode
│   ├── IfNode
│   ├── WhileNode
│   ├── ForNode
│   ├── ReturnNode
│   ├── BreakNode
│   ├── ContinueNode
│   ├── PrintNode
│   ├── BlockNode
│   └── ExpressionStatementNode
│
└── ExpressionNode (abstract)
    │
    ├── LiteralNode (integer, real, string, boolean, none)
    ├── ArrayLiteralNode
    ├── TupleLiteralNode
    ├── FunctionLiteralNode
    ├── BinaryOpNode
    ├── UnaryOpNode
    ├── ReferenceNode
    ├── ArrayAccessNode
    ├── TupleMemberAccessNode
    ├── FunctionCallNode
    └── TypeCheckNode (for 'is' operator)
```

Total: 23 specialized AST node types plus ProgramNode root

Node Categories:

- Statements (11 types): Control program flow and state changes
- Expressions (12 types): Evaluate to values
- Root (1 type): ProgramNode containing statement list

AST Node Representation

Node Structure

All AST nodes share common characteristics:

```
public abstract class StatementNode implements AstNode {
    private final int line;       // Source location
    private final int column;     // Source location

    protected StatementNode(int line, int column) {
        this.line = line;
        this.column = column;
    }

    // Visitor pattern support
    public abstract <T> T accept(AstVisitor<T> visitor);
}
```

Example: BinaryOpNode

```
public class BinaryOpNode extends ExpressionNode {
    private final ExpressionNode left;     // Left operand
    private final String operator;         // Operator symbol
    private final ExpressionNode right;    // Right operand

    public BinaryOpNode(int line, int column,
                        ExpressionNode left,
                        String operator,
                        ExpressionNode right) {
        super(line, column);
        this.left = left;
        this.operator = operator;
        this.right = right;
    }

    // Getters
    public ExpressionNode getLeft() { return left; }
    public String getOperator() { return operator; }
    public ExpressionNode getRight() { return right; }

    // Visitor pattern
    @Override
    public <T> T accept(AstVisitor<T> visitor) {
        return visitor.visitBinaryOp(this);
    }
}
```

Key Features:

- **Immutable**: All fields are `final`
- **Source location**: Every node knows its position
- **Type-safe**: Strong typing for operands
- **Visitor support**: For AST traversal

Core Parsing Logic

1. Grammar Specification

Expression precedence (lowest to highest):

```
precedence left OR;                  // or
precedence left XOR;                 // xor
precedence left AND;                 // and
precedence left EQUAL, NOT_EQUAL;    // =, !=, /=
precedence left LESS_THAN, ...;      // <, <=, >, >=
precedence left PLUS, MINUS;         // +, -
precedence left MULTIPLY, DIVIDE;    // *, /
precedence right NOT;                // not
precedence left DOT, LEFT_BRACKET;   // ., [, (
```

2. Production Rules

Example: If statement

```
if_statement ::=
    IF:i expression:cond THEN statement_list:thenBody END
    {: RESULT = new IfNode(ileft, iright, cond,
                           new BlockNode(ileft, iright, thenBody), null);
:}
  | IF:i expression:cond THEN statement_list:thenBody
       ELSE statement_list:elseBody END
    {: RESULT = new IfNode(ileft, iright, cond,
                           new BlockNode(ileft, iright, thenBody),
                           new BlockNode(ileft, iright, elseBody)); :}
  | IF:i expression:cond SHORT_IF statement:body
    {: List<StatementNode> bodyList = new ArrayList<>();
       bodyList.add(body);
       RESULT = new IfNode(ileft, iright, cond,
                           new BlockNode(ileft, iright, bodyList), null);
:}
  ;
```

What happens during parsing:

1. CUP matches input tokens to grammar rule pattern
2. Binds matched tokens to variables (e.g., `cond`, `thenBody`)
3. Executes semantic action code in `{: ... :}` block
4. Creates custom strongly-typed AST node (IfNode, BlockNode)
5. Passes source position using special variables (`ileft`, `iright`)
6. Assigns constructed node to `RESULT` for parser stack
7. Returns node to parent production rule

3. Token Mapping

LexerAdapter bridges our lexer to CUP:

```
public class LexerAdapter implements Scanner {
    private final Lexer lexer;

    @Override
    public Symbol next_token() throws Exception {
        Token token = lexer.nextToken();
        int symbolId = mapTokenTypeToSymbol(token.type());
        Object value = extractTokenValue(token);
        return new Symbol(symbolId,
                         token.line(),
                         token.column(),
                         value);
    }

    private int mapTokenTypeToSymbol(TokenType type) {
        return switch (type) {
            case IF -> Symbols.IF;
            case THEN -> Symbols.THEN;
            case INTEGER -> Symbols.INTEGER;
            // ... 50+ token types mapped
        };
    }
}
```

Statement Separators

Project D allows flexible separation:

```
separator ::=
    SEMICOLON      // ;
  | NEWLINE        // \n
  ;

separator_list ::=
    separator_list separator
  | separator
  ;

statement_list ::=
    separator_opt statement_list_core separator_opt
  ;
```

Examples:

```
var x := 1; var y := 2    // Semicolons
var x := 1
var y := 2                // Newlines
var x := 1;
var y := 2                // Mixed
```

Error Handling

How errors are detected:

1. **Lexer errors**: Invalid characters, malformed literals
2. **Parser errors**: Syntax errors (CUP's built-in detection)

Example error message:

```
Parse error at line 3, column 15: Syntax error
Expected one of: SEMICOLON, NEWLINE, END
Found: IDENTIFIER
```

Current strategy: Fail-fast

- Stop at first error
- Clear error message
- Provides error location

Implementation Statistics

Code Metrics

Metric	Value
Total Tests	193 tests
Test Coverage	78% overall, 81.5% parser
Grammar Lines	417 lines (parser.cup)
AST Node Types	23 types (plus ProgramNode root)
Supported Operators	20+ operators
Keywords	15 keywords

Test Organization

```
Parser Tests (136 tests):
├── ControlFlowTest.java        (23 tests) - if/while/for
├── ReturnPrintTest.java        (31 tests) - return/print
├── SeparatorTest.java          (22 tests) - semicolons/newlines
├── ErrorHandlingTest.java      (26 tests) - error detection
├── FunctionLiteralTest.java    (10 tests) - functions
├── OperatorPrecedenceTest.java (10 tests) - precedence
├── AssignmentTest.java         (13 tests) - declarations
└── ParserTest.java             (11 tests) - basic features

Integration Tests (4 tests):
└── EndToEndTest.java           (4 tests)  - full pipeline

Lexer Tests (43 tests):
├── LexerTest.java              (22 tests)
└── LexerEnhancedTest.java      (21 tests)
```

Key Features Implemented

Expressions

- Binary operators: `+`, `-`, `*`, `/`, `=`, `!=`, `<`, `>`, `<=`, `>=`, `and`, `or`, `xor`
- Unary operators: `+`, `-`, `not`
- Type checking: `expr is type`

Statements

- Variable declarations: `var x := 10, y := 20`
- Assignments: `x := value`
- If statements: `if...then...end`, `if...then...else...end`

- Short if: `if condition => action`
- While loops: `while...loop...end`
- For loops: `for x in iterable loop...end`
- Return and print statements

Literals

- All basic types: int, real, string, bool, none
- Collections: arrays `[...]`, tuples `{...}`
- Functions: `func(x) is...end`, `func(x) => expr`

References

- Variables, array access, tuple members, function calls
- Chaining: `obj.field[5].method()`
