INNOPOLIS UNIVERSITY

Javdin Parser

Course: [F25] Compiler Construction
Date: November 20, 2025
Team: 806

Team Information

Team 806

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

Project: Javdin

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/

Build & Testing Tools

- **Build System**: Maven 3.6+
- **Testing Framework**: JUnit 5
- **Assertion Library**: AssertJ
- **Code Coverage**: JaCoCo

Target Platform

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

Interpreter Implementation

The Javdin interpreter executes the optimized AST using the visitor pattern. It implements a tree-walking interpreter with dynamic typing, supporting eight value types: integer, real, boolean, string, array, tuple, function, and void. The interpreter uses a stack-based approach to handle lexical scoping in blocks and functions.

The core interpreter class implements the AstVisitor interface:

```
public class Interpreter implements AstVisitor<Value> {
    private final ErrorHandler errorHandler;
    private final Environment environment;

    public void interpret(ProgramNode program) {
        try {
            program.accept(this);
        } catch (RuntimeError error) {
            errorHandler.addError("Runtime error: " + error.getMessage(),
                                  error.getLine(), error.getColumn());
        } catch (ReturnSignal signal) {
            errorHandler.addError("Return statement outside function",
                                  program.getLine(), program.getColumn());
        }
    }
}
```

Dynamic typing is implemented through a Value wrapper class with type enumeration:

```
public class Value {
    public static final Value VOID = new Value(null);
    private final Object value;
    private final ValueType type;

    public enum ValueType {
        INTEGER, REAL, BOOLEAN, STRING, ARRAY, TUPLE, FUNCTION, VOID
    }

    public static Value integer(int value) { return new Value(value); }
    public static Value real(double value) { return new Value(value); }
    public static Value array(ArrayValue value) { return new Value(value);
}
}
```

Function values capture their lexical environment to support closures:

```
@Override
public Value visitFunctionLiteral(FunctionLiteralNode node) {
    FunctionValue functionValue = new FunctionValue(node,
                                    environment.captureCurrentScope());
    return Value.function(functionValue);
}
```

Binary operations handle type checking and arithmetic with automatic promotion:

```
@Override
public Value visitBinaryOp(BinaryOpNode node) {
    Value left = evaluate(node.getLeft());
    Value right = evaluate(node.getRight());
    return switch (node.getOperator()) {
        case "+" -> add(left, right, node);
        case "-" -> subtract(left, right, node);
        case "*" -> multiply(left, right, node);
        case "/" -> divide(left, right, node);
        case "=", "==" -> Value.bool(equals(left, right));
        case "and" -> Value.bool(requireBoolean(left, node) &&
                                 requireBoolean(right, node));
        default -> throw runtimeError("Unsupported operator", node);
    };
}
```

Execution Flow Example

Consider this simple program:

```
var x := 5
var y := x + 3
print y
```

The interpreter executes this program as follows:

1. visitProgram(ProgramNode) - Start execution of the program
2. visitDeclaration(DeclarationNode) - Process first declaration
    - visitLiteral(5) - Evaluate right side, returns Value.integer(5)
    - environment.define("x", Value.integer(5)) - Store variable x
3. visitDeclaration(DeclarationNode) - Process second declaration
    - visitBinaryOp(+) - Evaluate right side
        - visitReference("x") - Look up x, returns Value.integer(5)
        - visitLiteral(3) - Evaluate 3, returns Value.integer(3)
        - add(5, 3) - Perform addition, returns Value.integer(8)
    - environment.define("y", Value.integer(8)) - Store variable y
4. visitPrint(PrintNode) - Execute print statement

- visitReference("y") - Look up y, returns Value.integer(8)
- System.out.println("8") - Output to console

Interpreter Tests

Test 1: Factorial with Recursion and Loop Control

```
var factorial := func(n) is
    if n <= 1 then
        return 1
    else
        return n * factorial(n - 1)
    end
end

for i in 1..5 loop
    print factorial(i)
    if i = 3 => exit
end
```

Run with:

```
java -jar target/javdin-1.0.0.jar test-resources/factorial.d
```

Expected output:

```
1
2
6
```

Test 2: Complex Program with Functions and For-In Loop

```
var factorial := func(n) is
    if n <= 1 then
        return 1
    else
        return n * factorial(n - 1)
    end
end

var numbers := [1, 2, 3, 4, 5]
var sum := 0

for value in numbers loop
    sum := sum + value
    print "Adding element", value, "sum is now", sum
```

```
end

var result := factorial(5)
print "Factorial of 5 is:", result

var square := func(x) => x * x
print "Square of 7 is:", square(7)
```

Run with:

```
java -jar target/javdin-1.0.0.jar test-resources/complex.d
```

Expected output:

```
Adding element 1 sum is now 1
Adding element 2 sum is now 3
Adding element 3 sum is now 6
Adding element 4 sum is now 10
Adding element 5 sum is now 15
Factorial of 5 is: 120
Square of 7 is: 49
```

Test 3: Array Mixed Types with First-Class Functions

```
var arr := []
var fn := func(x) => x * 2
arr[1] := fn
arr[2] := 42
arr[3] := "hello"
arr[4] := [1, 2, 3]
arr[5] := {a := 1}
print arr[1](5)
print arr[2]
print arr[3]
print arr[4][2]
print arr[5].a
```

Run with:

```
java -jar target/javdin-1.0.0.jar test-resources/test-array-mixed-types.d
```

Expected output:

```
10
42
hello
2
1
```

Test 4: Nested Arrays

```
var nested := [[1, 2], [3, 4], [5, 6]]
print nested[1][1]
print nested[1][2]
print nested[2][1]
print nested[3][2]
```

Run with:

```
java -jar target/javdin-1.0.0.jar test-resources/test-array-nested.d
```

Expected output:

```
1
2
3
6
```

Test 5: Complex Tuples with Nested Access

```
var fn := func(x) => x + 10
var tup := {
    name := "test",
    value := 42,
    calc := fn,
    nested := {x := 1, y := 2}
}
print tup.name
print tup.value
print tup.calc(5)
print tup.nested.x
print tup.nested.y
```

Run with:

```
java -jar target/javdin-1.0.0.jar test-resources/test-tuple-complex.d
```

Expected output:

```
test
42
15
1
2
```