IJVM Interpreter: Design and Implementation of a Subset of the IJVM Specification

Author: Antonio Masala
Co-Author: Marco La Civita

April 28, 2025

Abstract

This paper presents the design and implementation of an interpreter for a subset of the IJVM instruction set, developed as an educational tool to demonstrate virtual machine concepts. Built using Java and ANTLR, the system comprises four key components: Lexer, Parser, Semantic Analyzer, and Interpreter.

While not intended for formal verification, this system serves as a practical and flexible tool for learning and experimentation.

Project repository: `https://github.com/atom7xyz/antlr-impl`

1 Introduction

The IJVM (Integer Java Virtual Machine) is a simplified virtual machine model used in academic settings to illustrate fundamental low-level programming and computer architecture concepts. This project implements an interpreter for a chosen subset of the IJVM specification.

The interpreter reads and executes IJVM-like source code directly, without compiling it into bytecode or machine code. It implements a full processing pipeline—including lexical analysis, parsing, semantic analysis, and execution—using Java 11 along with the ANTLR 4 framework for robust grammar-based language recognition.

This implementation was developed as part of the Architecture of Computer Systems course under the supervision of Professor Diego Reforgiato Recupero at the University of Cagliari. Certain IJVM features, such as the `WIDE` instruction, have been deliberately excluded to maintain focus on core concepts and reduce complexity.

Although the design emphasizes logical and structural correctness, it does not provide formal guarantees of correctness or completeness. Users should view this interpreter as a valuable educational tool rather than a complete implementation.

2 Design and Architecture

The system is designed as a modular architecture, with clearly defined components corresponding to each phase of the interpretation process.

2.1 Lexer and Parser

The lexer and parser are generated using ANTLR 4 based on the grammar files `IJVMLexer.g4` and `IJVMParser.g4`. The lexer tokenizes the IJVM source code into meaningful tokens, which the parser then processes to construct an abstract syntax tree (AST), serving as the structural basis for subsequent semantic analysis and interpretation.

2.2 Semantic Analyzer

The semantic analyzer, implemented in `SemanticAnalyzer.java`, traverses the AST to enforce semantic rules. It checks for issues such as the use of undeclared or uninitialized variables, as well as invalid method invocations. The `SymbolTable` class manages scopes and symbols, ensuring proper resolution

of variables and methods, while the `SemanticWarning` and `SemanticError` classes communicate potential issues and critical errors, respectively.

2.3 Interpreter

The interpreter itself—implemented in `IJVMProgram.java`—executes the IJVM code directly from the AST. The `IJVMProgram` class orchestrates the overall execution flow, including the management of a call stack and constant pool. The `IJVMScope` class governs individual execution contexts, handling local variables and operand stacks, while the `IJVMInstruction` class encapsulates the behavior of individual instructions. The system's stack operates on signed 32-bit integers.

2.4 Snapshot and Tracer

The interpreter includes a tracing mechanism implemented through the `IJVMTracer` and `IJVMSnapshot` classes to aid in debugging and understanding program execution.

2.4.1 Tracer

The `IJVMTracer.java` class provides a detailed view of the interpreter's state during execution. It displays information about the current instruction, program counter, scope details (including stack and local variables), call stack, and constant pool. The tracer uses color-coded console output to organize information into logical categories such as static data, current execution context, and control flow.

2.4.2 Snapshot

The `IJVMSnapshot.java` class captures a complete memory dump at a specific point during the execution of the program. It records the state of all scopes, call stacks, constant pools, local variables, and operand stacks associated with each scope. By maintaining a snapshot of the interpreter's memory, it enables detailed analysis and comparison of program states between execution steps.

3 Implementation Details

The interpreter is implemented entirely in Java 11, with an emphasis on modularity and extensibility. Key implementation details for the major com-

ponents are described in the following subsections.

3.1 Lexer and Parser

The lexer's role is to convert raw IJVM source code into tokens. The design choices in `IJVMLexer.g4` include:

- **Case Insensitivity**: IJVM instructions such as `IADD`, `POP`, and `GOTO` accept both uppercase and lowercase representations, though the interpreter normalizes them to uppercase.
- **Flexible Number Formats**: The lexer supports decimal, hexadecimal (prefixed with `0x` or `0X`), and octal (prefixed with `o` or `O`) formats.
- **Identifier Rules**: Identifiers for variables, constants, methods, and labels begin with a letter or underscore, followed by any combination of alphanumeric characters, underscores, or hyphens.
- **Whitespace and Comment Handling**: Whitespace and comments (using both `//` and `;` styles) are ignored, simplifying tokenization. Newlines are explicitly tokenized (as `NEWLINE`) to aid in block delimitation and error reporting.
- **Block Keyword Recognition**: Specific tokens such as `.constant`, `.end-constant`, `.main`, `.end-main`, `.method`, `.end-method`, `.var`, and `.end-var` structure the IJVM program into well-defined sections.

The parser (`IJVMParser.g4`) uses these tokens to construct an AST, following a clear hierarchical structure that organizes IJVM programs into blocks (constant, main, method, and var). Decisions made include:

- **Hierarchical Structure**: The grammar defines a program as a sequence of blocks, ensuring clear separation between the main execution code and method definitions.
- **Instruction Classification**: Instructions are grouped by their expected argument patterns (e.g., zero-argument, byte-argument, variable-argument), simplifying subsequent semantic analysis.
- **Label and Method Declarations**: Labels (e.g., `loop:`) and methods (with optional parameters) are parsed to ensure clear error reporting and unambiguous interpretation.

- **Robust Error Recovery**: The grammar includes rules (such as requiring `NEWLINE` tokens after block terminations) that facilitate graceful error recovery.

3.2 Semantic Analysis

The semantic analyzer validates the AST through a multi-pass approach:

3.2.1 Declaration Registration

In the first pass, the analyzer registers all declarations for constants, variables, and methods. Constants are stored in a global scope, while variables and method parameters are scoped to their respective blocks. This registration ensures that symbols are defined before use.

3.2.2 Label Scanning

The second pass catalogues all labels (targets for jump instructions such as `GOTO` and `IFEQ`) within each scope. Labels are confined to their respective method or main block to avoid cross-scope conflicts.

3.2.3 Instruction and Expression Validation

During the final pass, the analyzer verifies:

- That all variables are properly initialized before use.
- That method invocations target declared methods with the correct number of parameters.
- That jump targets (labels) are defined within the same scope.

3.2.4 Symbol Table Management

The `SymbolTable` class maintains a hierarchical record of symbols with details such as initialization status and parameter counts. This detailed bookkeeping allows the analyzer to produce precise error messages and warnings.

3.2.5 Error and Warning Reporting

Errors (e.g., undeclared variables, misuse of jump targets) are reported with specific source code locations to aid debugging. Non-critical issues, such as unused symbols, are flagged as warnings to help users refine their code.

3.3 Interpreter

The interpreter executes the IJVM code by traversing the validated AST. It supports a specific subset of instructions, including:

- **Arithmetic Operations**: `IADD`, `ISUB`, `IAND`, `IOR`.
- **Stack Manipulations**: `POP`, `SWAP`, `DUP`.
- **Control Flow**: `GOTO`, `IFEQ`, `IFLT`, `IF_ICMPEQ`.
- **Method Invocation**: `INVOKEVIRTUAL` (with limited support).

Table 1 details the instruction specifications and operands. The interpreter maintains a stack-based execution model, where each execution scope has its own operand stack and local variables. The stack operates on signed 32-bit integers.

3.3.1 Supported Instructions

Table 1: Supported IJVM Instructions

Instruction	Description	Operands
HALT	Terminates program execution	None
NOP	No operation	None
IADD	Adds the top two integers on the stack	None
ISUB	Subtracts the top integer from the second	None
IAND	Performs bitwise AND on the top two integers	None
IOR	Performs bitwise OR on the top two integers	None
POP	Removes the top element from the stack	None
SWAP	Swaps the top two elements on the stack	None
DUP	Duplicates the top element on the stack	None
ERR	Simulates an error	None
IN	Simulates input reading	None
OUT	Outputs the top stack value	None
IRETURN	Returns from a method with the top stack value	None
BIPUSH	Pushes a byte value onto the stack	Integer
IINC	Increments a local variable by a value	Variable ID, Integer
ILOAD	Loads a variable value onto the stack	Variable ID
ISTORE	Stores the top stack value into a variable	Variable ID
INVOKEVIRTUAL	Invokes a method	Method ID
LDC_W	Loads a constant value onto the stack	Constant ID
IFLT	Jumps if the top stack value is less than 0	Label ID
IFEQ	Jumps if the top stack value equals 0	Label ID
IF_ICMPEQ	Jumps if the top two stack values are equal	Label ID
GOTO	Unconditionally jumps to a label	Label ID

4 Usage

The IJVM interpreter is packaged as a command-line tool. The main application is invoked as follows:

```
Usage: java -jar antlr-impl.jar [options]
Options:
  -parse, -interpret       Specify the mode
  -ijvm                    Specify the language
  -file <path>             Path to the source file
  -d, -debug               Enable debug mode
  -tracer                  Enable tracer for step-by-step execution view
```

- `-parse`: Parse the specified IJVM file and report syntax and semantic errors.
- `-interpret`: Execute the specified IJVM file, provided no errors are found during parsing or semantic analysis.
- `-ijvm`: Specify the IJVM language.
- `-file <path>`: Path to the IJVM source file.
- `-d`, `-debug`: Enable debug mode for verbose output.
- `-tracer`: Enable the tracer to view detailed step-by-step execution states, including memory dumps and stack contents.

These options support both code validation and execution, providing detailed error and warning reports for debugging and learning.

5 Testing

Comprehensive unit tests have been developed to verify the correct operation of the parser, semantic analyzer, and interpreter. These tests are contained in the following classes:

- `IJVMProgramTest.java`: Evaluates the interpreter's handling of IJVM instructions, including arithmetic operations, stack manipulation, control flow, and method invocation. Test cases address scenarios such as stack underflow, variable initialization, and looping constructs.

- `SemanticAnalyzerTest.java`: Checks that the semantic analyzer detects issues like the use of uninitialized variables, undeclared method calls, and improper references.
- `ParserErrorTest.java`: Validates that the parser correctly identifies syntax errors, such as malformed instructions or missing block terminators.

These tests enhance confidence in the interpreter's functionality, although the implementation remains primarily an educational tool.

6 Summary

In this paper, we have presented the design and implementation of an interpreter for a selected subset of the IJVM specification. The system is structured as a modular pipeline, featuring a lexer and parser built with ANTLR, a semantic analyzer for robust validation, and a direct execution engine implemented in Java. While the interpreter does not aim for full compliance with the entire IJVM standard, it provides a practical and accessible platform for learning virtual machine architecture, language processing, and execution models. Through its clear structure and comprehensive error reporting, the interpreter serves as a valuable educational resource and a foundation for further exploration or extension.

References

- ANTLR: `https://www.antlr.org/`
- Java: `https://www.java.com/en/`