antlr-impl: Design and Implementation of an Interpreter for a subset of the IJVM specification

Author: Antonio Masala
Co-Author: Marco La Civita

June 19, 2025

Abstract

This paper provides a high-level overview of the design and implementation of the tool `antlr-impl`, an interpreter for a subset of the IJVM instruction set, which was created as an educational tool to show low-level execution flow and memory management. The system, which was built with Java and ANTLR, consists of four main components: a Lexer, a Parser, a Semantic Analyzer, and an Interpreter.

While not intended for formal verification, this approach is an effective tool for learning and experimenting.

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

1 Introduction

The IJVM (Integer Java Virtual Machine) is a simple virtual machine model that is used in academic contexts to illustrate key low-level programming and computer architecture concepts.

This tool was developed for the Architecture of Computer Systems course taught by Professor Diego Reforgiato Recupero at the University of Cagliari.

It makes use of Java 11 and the ANTLR 4 framework for robust grammar-based language recognition to put together an exhaustive processing pipeline, involving lexical analysis, parsing, semantic analysis, and execution.

To maintain focus on core concepts and reduce complexity the tool does not implement the IJVM specification fully: some instructions such as the `WIDE` instruction have been deliberately excluded. Table 1 provides the list of supported instructions.

Although the design of the tool aims to achieve logical and structural correctness, it does not provide formal guarantees of correctness or completeness. Users should view it as a valuable educational tool rather than an oracle of thruth.

2 Design and Architecture

The system features a modular design with separate components for each interpretation step. Unlike traditional approaches, the interpreter executes IJVM source code directly without intermediate compilation into bytecode.

2.1 Lexer and Parser

The lexer and parser are generated using ANTLR 4 and it's grammar files `IJVMLexer.g4` and `IJVMParser.g4`. The lexer converts the IJVM source code into meaningful tokens, which the parser then uses to build an abstract syntax tree (AST), which serves as the structural foundation for semantic analysis and interpretation.

2.2 Semantic Analyzer

The semantic analyzer implemented in `SemanticAnalyzer.java` enforces semantic rules through AST traversal. It detects problems such as undeclared or uninitialized variables, as well as incorrect function calls. The `SymbolTable` class handles scopes and symbols to ensure proper resolution of

variables and methods. The `SemanticWarning` and `SemanticError` classes are used to signal potential problems and fatal errors, respectively.

2.3 Interpreter

The interpreter in `IJVMProgram.java` executes the IJVM code directly from the AST without intermediate compilation. That same class manages the execution flow, including the call stack and constant pool. The `IJVMScope` class handles execution contexts, including local variables and operand stacks, whereas the `IJVMInstruction` class encapsulates the behavior of individual instructions. The system's stack utilizes signed 32-bit integers.

2.4 Snapshot and Tracer

The interpreter includes a tracing system built around the `IJVMTracer` and `IJVMSnapshot` classes to aid in debugging and in understanding program execution steps.

2.4.1 Tracer

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

2.4.2 Snapshot

The `IJVMSnapshot.java` class records a complete memory dump at a given point during the program's execution. It keeps track of the status of all scopes, call stacks, constant pools, local variables, and operand stacks for each scope. Maintaining a snapshot of the interpreter's memory allows for extensive analysis and comparison of program states between execution steps.

3 Implementation Details

The interpreter has been completely implemented in Java 11, with a focus on modularity and extensibility. The following subsections cover key implementation details for the primary components.

3.1 Lexers and Parsers

The lexer transforms raw IJVM source code into tokens. The design features in `IJVMLexer.g4` include:

- **Case Insensitivity**: All instructions can have both uppercase and lowercase representations, still, the interpreter normalizes them to uppercase.
- **Flexible Number Formats**: The lexer accepts decimal, hexadecimal (prefixed with `0x` or `0X`), and octal (prefixed with `0`) byte rapresentations.
- **Identifier Rules**: Identifiers for variables, constants, methods, and labels start with a letter or underscore, followed by any combination of alphanumeric characters, underscores, or hyphens.
- **Whitespace and Comment Handling**: Tokenization is simplified by ignoring whitespace and comments (using `//` and `;` styles). Newlines are explicitly tokenized (as `NEWLINE`) to help with block delimitation and error reporting.
- **Block Keyword Recognition**: Tokens like `.constant`, `.end-constant`, `.main`, `.end-main`, `.method`, `.end-method`, `.var`, and `.end-var` (which are part of the IJVM specification) aid in organizing the IJVM program into well-defined sections.

The parser uses these tokens to create the AST, which has a clear hierarchical structure that divides IJVM programs into blocks (constant, main, method, and var). The decisions made include:

- **Hierarchical Structure**: The language defines a program as a sequence of blocks, clearly separating the main execution code and method definitions.
- **Instruction Classification**: Instructions are classified based on expected argument patterns (e.g., zero-argument, byte-argument, variable-argument), allowing for successive semantic analysis.
- **Label and Method Declarations**: Labels (e.g., `loop:`) and methods (with optional parameters) are parsed to provide clear error reporting and unambiguous interpretation.

- **Robust Error Identification**: The grammar includes rules (e.g., requiring `NEWLINE` tokens after block terminations) that aid in the pin-pointing of mistakes throught the IJVM source code.

3.2 Semantic Analysis

The semantic analyzer evaluates the AST using a multi-pass technique.

3.2.1 Declaration Registration

During the first pass, the analyzer registers all declarations of constants, variables, and methods. Constants are maintained in a global scope, whereas variables and method parameters are scoped to specific blocks. The registration process guarantees that symbols are defined upfront.

3.2.2 Label Scanning

The second pass catalogs all labels (targets of jump instructions like `GOTO` and `IFEQ`) within each scope. Labels are limited to their correct method or main block to avoid cross-scope issues.

3.2.3 Instruction and Expression Validation

During the final pass, the analyzer ensures that:

- All variables are properly initialized before use.
- Method invocations target declared methods with the correct amount of parameters.
- Jump targets (labels) are defined inside the same scope.

3.2.4 Symbol Table Management

The `SymbolTable` class keeps a hierarchical record of symbols, including details like initialization status and parameter counts. This extensive record-keeping enables the analyzer to generate precise error messages and warnings.

3.2.5 Error and Warning Reporting

Errors (e.g., undeclared variables, misuse of jump targets) are reported with precise source code locations for debugging. Non-critical errors, such as unnecessary symbols, are displayed as warnings to assist users in refining their code.

3.3 Interpreter

The interpreter executes the IJVM code by traversing the validated AST. It supports a specific set of instructions, which include:

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

Table 1 details the instruction specifications and operands supported by this tool. 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.

5 Testing

Comprehensive unit tests have been written to ensure that the parser, semantic analyzer, and interpreter are functioning properly. These tests are found in the following classes:

- `IJVMProgramTest.java`: Evaluates the interpreter's computation of IJVM instructions, including arithmetic operations, stack management, control flow, and method calls. Test cases address scenarios such as stack underflow, variable initialization, and looping constructs.
- `SemanticAnalyzerTest.java`: Ensures that the semantic analyzer finds errors such as uninitialized variables, undeclared method calls, and invalid references.
- `ParserErrorTest.java`: Ensures the parser finds syntax issues, such as invalid instructions or missing block terminations.

These tests increase confidence in the interpreter's functionality, but the implementation remains primarily an educational and experimental tool.

6 Summary

In this paper, we describe the design and implementation of an interpreter for a subset of the IJVM specification. The system is designed as a modular pipeline, with an ANTLR-based lexer and parser, a semantic analyzer for source code validation, and a Java 11 direct execution engine. While the interpreter does not seek to be in full compliance with the IJVM specification, it does provide a useful and accessible platform for learning virtual machine architecture, language processing, and execution models through its clear structure and comprehensive error/warning reporting capabilities.

References

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