Lab 6: Full-Stack Integration of a Managed Runtime Environment — Shell, Compiler, and Virtual Machine with Garbage Collection

Jithendra Gannavarapu (2025MCS2743)
Manmadha Rao Kuttum (2025MCS2101)

February 3, 2026

Abstract

This report details the successful integration of a multi-stage software toolchain. The system unifies a POSIX-compliant shell (Lab 1), a Flex/Bison front-end (Lab 2), a Bytecode IR Generator (Lab 3), and a managed Virtual Machine with an internal Mark-Sweep Garbage Collector (Labs 4 & 5). We demonstrate how metadata generated during parsing is utilized for instruction-level debugging and memory liveness analysis. The final system achieves full binary compatibility between independent modules through a strictly defined instruction set and value-tagging system.

1 Introduction

The objective of Lab 6 was to move beyond standalone implementations and create a unified "Integrated System." In this architecture, the shell is not just a command-line interface but a process coordinator. The compiler is not just a syntax checker but a bytecode emitter. The Virtual Machine (VM) is not just a mathematical evaluator but a state machine that supports introspection.

2 System Architecture and Codebase Organization

The system is organized into a modular pipeline where each directory represents a distinct phase of the software lifecycle.

- **01_Shell**: The primary user interface. It manages program IDs (PIDs) and coordinates between the compiler and VM via the `ProgramManager`.
- **02_Parser**: Contains the Lexer (`lexer.l`) and Parser (`parser.y`) developed in Lab 2 to transform source code into an Abstract Syntax Tree (AST).
- **03_Compiler**: The IR Generator. It performs post-order traversal on the AST to emit a linear vector of bytecode opcodes.

- **04_VM_Execution**: The runtime engine (`VirtualMachine.cpp`) that executes bytecode using a Fetch-Decode-Execute cycle and supports software breakpoints.
- **05_Memory_GC**: The Mark-Sweep Garbage Collector developed in Lab 5 for dynamic memory reclamation and liveness tracking.

3 Formal Technical Specification

3.1 Instruction Set Architecture (ISA)

The system utilizes a custom 8-bit opcode format with 32-bit operands. Below are the primary instructions that unify the compiler and the VM:

Opcode	Mnemonic	Description
0x01	PUSH val	Pushes a 32-bit integer onto the operand stack.
0x10	ADD	Pops two values, adds them, and pushes the result.
0x21	JZ addr	Pops a value; if zero, sets PC to the target address.
0x30	STORE idx	Pops a value and stores it in the VM memory at index.
0x31	LOAD idx	Loads a value from memory index to the stack.
0x42	PRINT	Pops a value and displays it to the standard output.
0xFF	HALT	Terminates execution and returns control to the shell.

Table 1: Core Instruction Set for the Integrated System

3.2 Value Tagging and Object Representation

To support the Garbage Collector (Lab 5), every stack and memory element uses a `Value` tagged union.

$$Value = \{\text{Type: } (INT|OBJ), \text{Data: } (int32\_t|Object*)\}$$

This allows the VM to distinguish between primitive data and heap-allocated objects during the **Mark Phase**.

4 Advanced Integration Features

4.1 Control Flow and Label Back-patching

One of the most complex integration points was the implementation of `WHILE` loops. Because jump targets (e.g., the end of a loop) are unknown when the jump instruction is first emitted, the `IRGenerator` uses a **Fixup List**.

1. The generator emits a jump instruction with a placeholder (0).
2. It records the bytecode offset of this placeholder in a fixup table.
3. Once the target label is "placed," it iterates through the fixup table and overwrites the placeholders with the actual memory address.

4.2 The Debugger State Machine

The Lab 6 debugger replaces the Lab 2 `ptrace` logic with internal state accessors. By modifying the VM to use an `executeNext()` method, the shell can execute precisely one instruction and then pause to display the "registers" (PC and Stack).

5 Garbage Collection & Memory Safety

The system utilizes the Mark-Sweep algorithm to handle transitive reachability. Unlike standalone versions, this GC is "Execution-Aware."

- **Mark Phase**: Recursively traces live objects starting from the current VM stack and global memory variables.
- **Sweep Phase**: Traverses the `objects` linked-list and unlinks nodes whose `marked` flag is false.

6 Integrated System Validation

6.1 Integration Case Study: Control Flow and Variable Bindings

We verified the system using a script containing an `if-condition` and a `while-loop`. The logs below demonstrate the data flow from AST to execution.

```
(debug) ast
WHILE
  CONDITION: OPERATOR: < (ID: i, VALUE: 3)
  BODY: ASSIGNMENT: i (OPERATOR: + (ID: i, VALUE: 1))

(debug) bytecode
  25: 0x31 (LOAD ) 2
  27: 0x01 (PUSH ) 3
  29: 0x14 (CMP  )
  30: 0x21 (JZ   ) 41

(debug) regs
--- VM Register State ---
PC    : 2
Stack : [ 10 ]
```

Listing 1: Actual System Logs during Debugging

6.2 Analysis of Results

The trace above confirms that:

1. The **Parser** correctly interpreted the `WHILE` loop hierarchy.
2. The **Compiler** generated a `JZ` (Jump if Zero) at offset 30 pointing to the escape address 41.
3. The **VM** successfully stepped through instructions, showing the value 10 on the stack after the first assignment.

7 Conclusion

The Lab 6 Integrated System achieves the goal of a "Unified Execution Model." By centralizing control in a managed `ProgramManager` and implementing a step-enabled Virtual Machine, we have created a toolchain that is both executable and observable. This work demonstrates the critical necessity of metadata flow—where the front-end's syntax analysis directly informs the back-end's debugging and memory management capabilities.