Compilation and Interpretation Modules: An Organizational Framework for Interpreter Architecture

Joaquin Ignacio Romero[1], David Gonzalo Romero[1]

[1]Universidad Nacional de Salta, Salta, Argentina

joaquinromeroignacio@gmail.com, gromero@di.unsa.edu.ar

Abstract

Implementing a bytecode interpreter involves interrelated design decisions across two principal modules: a compilation module that transforms source syntax into an internal representation, and an interpretation module that executes that representation to produce program behavior. Canonical references — including Aho et al. (1986) and Nystrom (2021) — explain the algorithms of each module in detail, but leave implicit how decisions in one module impose structural obligations on the other. This paper introduces an organizational framework that makes these inter-module dependencies explicit. We characterize bytecode interpreter architecture through its internal representation and derive four organizational principles governing expressions, data types, variables, and control flow, illustrated using CGecko — a bytecode interpreter derived from CLox (Nystrom, 2021). The result is a conceptual framework that clarifies which design decisions are free and which are forced by prior choices in the other module.

Keywords: Interpreter Architecture, Compilation Module, Interpretation Module, Internal Representation, Bytecode Virtual Machine.

Módulos de Compilación e Interpretación: Un Marco Organizacional para la Arquitectura de Intérpretes

Resumen

Implementar un intérprete a bytecode implica decisiones de diseño interrelacionadas entre dos módulos principales: un módulo de compilación que transforma la sintaxis fuente en una representación interna, y un módulo de interpretación que ejecuta dicha representación para producir el comportamiento del programa. Las referencias canónicas — incluyendo Aho et al. (1986) y Nystrom (2021) — explican los algoritmos de cada módulo en detalle, pero dejan implícito cómo las decisiones en un módulo imponen obligaciones estructurales en el otro. Este trabajo introduce un framework organizacional que hace explícitas estas dependencias entre módulos a través de cuatro principios organizacionales que gobiernan expresiones, tipos de datos, variables y flujo de control, ilustrados con CGecko — un intérprete derivado de CLox (Nystrom, 2021).

Palabras clave: Arquitectura de Intérprete, Módulo de Compilación, Módulo de Interpretación, Representación Interna, Máquina Virtual a Bytecode.

1 Introduction

Interpreter implementation is well-studied at the algorithmic level. Aho et al. (1986) covers lexical analysis, parsing, and code generation; Nystrom (2021) walks through two complete implementations, including a bytecode compiler with virtual machine. Despite this breadth, a structural gap persists: these works explain the algorithms of each module in relative isolation but do not articulate how decisions in one module constrain the other. When the compilation module resolves local variables statically, the virtual machine must maintain an indexed execution stack with predictable offsets. When the compilation module stores all values as uniform tagged-unions, the execution stack must represent every slot in the same format. These inter-module dependencies are not apparent from reading either module in isolation — the implementer discovers them through implementation errors. This paper introduces an organizational framework that makes them explicit. Following Stallings (2016), *organization* refers to operational attributes not visible to the programmer but central to the implementer. We present four organizational principles governing expressions, data types, variables, and control flow, illustrated using CGecko — a bytecode interpreter for the Gecko language derived from CLox (Nystrom, 2021) (`https://github.com/Joacoromero06/CGecko`). Our contributions are: (1) a characterization of bytecode interpreter architecture through its IR and the structural obligations it generates; (2) four organizational principles making inter-module dependencies explicit; and (3) a worked analysis of how these principles manifest in a concrete implementation.

2 Interpreter Architecture

A bytecode interpreter consists of three components: a **Compilation Module**, an **Internal Representation (IR)**, and an **Interpretation Module** (Figure 1). The compilation module's parser assembles tokens into grammatical constructs; its compiler transforms those constructs into bytecode and makes decisions that extend beyond its own module — how variables are classified, how literals are stored, how heap objects are managed — each generating a corresponding obligation in the interpretation module. The IR is the coupling interface between both modules: it consists of a *bytecode vector* of encoded instructions, a *constant table* of uniform value representations for all literals, a *globals map* associating global names with runtime values, and a *heap object list* of variable-size objects. The interpretation module receives these compiled artifacts, dispatches each instruction via a protocol execution, and maintains the execution stack and instruction pointer. Both modules grow together: each new language feature requires new semantic encodings on the compilation side and new execution protocols on the interpretation side.

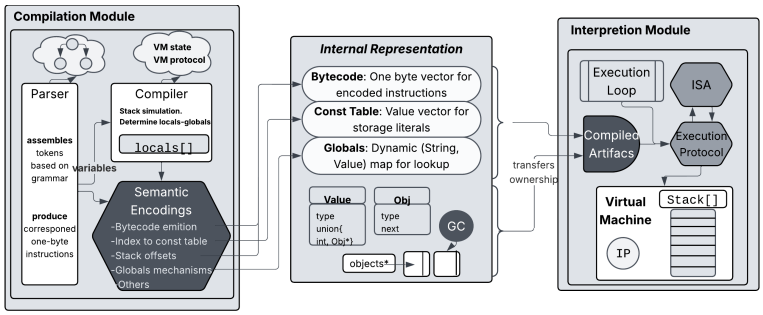

Figure 1: Architecture of a bytecode interpreter (CGecko). The IR mediates both modules and encodes inter-module structural obligations. *Semantic Encodings* denotes bytecode sequences, constant table entries, and stack layout decisions; *Protocol Execution* denotes the corresponding VM handler for each ISA instruction.

3 Organizational Principles

The four principles that follow identify specific inter-module dependencies arising from fundamental language constructs and articulate the design obligations they generate. This coupling pattern — a shared representation that constrains adjacent components mutually — recurs throughout systems design: system call interfaces couple kernel and user space; packet formats couple network layers. Recognizing these coupling points early separates systematic design from trial-and-error implementation.

3.1 Expressions: The Postorder–Stack Duality

The compilation module does not materialize an abstract syntax tree. Instead, as it parses an expression using a precedence-aware algorithm (Pratt, 1973), it emits bytecode in postorder — the order in which the AST would be traversed during evaluation (Aho et al., 1986), with literals stored in the constant table by index. This encoding generates a precise obligation: the interpretation module must maintain a LIFO execution stack. Postorder emission places operand instructions before operator instructions; the stack accumulates operand values, and each operator pops its arguments, computes, and pushes the result. Evaluating bytecode this way is structurally equivalent to a recursive postorder AST traversal, an instance of the classical correspondence between recursion and stack-based computation (Aho et al., 1983). Postorder emission and the LIFO stack are not independent design choices: they are two sides of a single organizational decision spanning both modules, which together avoid runtime tree traversal. As an example, $2 * 3 + 4$ compiles to `CONST 2, CONST 3, *, CONST 4, +`, yielding stack states $[2]$, $[3, 2]$, $[6]$, $[4, 6]$, $[10]$ — semantically equivalent to a postorder AST traversal without materializing the tree.

3.2 Data Types: Uniform Value Representation and Memory Management

The execution stack demands uniformity — every slot must occupy the same fixed size for constant-time push and pop — yet a language must operate over values of heterogeneous size. The interpretation module resolves this with a *tagged-union*: a struct pairing an explicit type tag with a union field (Strömberg Skott, 2022). Fixed-size primitives are stored inline; variable-size entities are heap-allocated, with only a pointer stored in the field. This `Value` type is also the shared representation through which values cross the module boundary. The compilation module stores every literal in the constant table as a `Value`; the bytecode carries only an index, and at runtime the interpretation module retrieves and pushes it with a single uniform operation regardless of type. For operations, the compilation module emits type-agnostic instructions (a single `OP_ADD` covers both numeric addition and string concatenation), delegating type dispatch to the interpretation module. For heap objects, the interpretation module assumes exclusive ownership via an intrusive list traversed by the garbage collector, making double-free errors structurally impossible and freeing the compilation module from any memory management responsibility. Without this abstraction, a distinct instruction per type-and-operation combination would be required, growing the ISA proportionally; the tagged-union keeps the ISA compact and bounds execution-loop complexity to a single dispatch per instruction.

3.3 Variables: Binding-Time Bifurcation

Implementing variable access requires resolving, for each identifier, where its value is stored and when that location becomes known. This binding-time decision is not uniform, forcing the compilation module to adopt two structurally distinct strategies with correspondingly distinct runtime mechanisms.

Global variables may be referenced before their declaration is encountered in a sequential pass. Late binding is required: each global access is compiled as an instruction carrying the variable's name as a constant-table index, deferring resolution to runtime. This obliges the interpretation module to maintain a dynamic hash table (globals map) and perform a lookup on every global access.

Local variables benefit from lexical scope: their storage is bounded to the enclosing block, and statements have net-zero stack effect while declarations persistently leave a value in a dedicated slot. Because this behavior is fully predictable from source structure, the compilation module can simulate the runtime stack statically via `locals[]` — an internal array whose entries mirror the VM's execution stack slots exactly. This compile-time knowledge transforms all three variable operations into constant-time, index-based instructions: declaration emits no opcode (the slot is the variable); reference and assignment each emit a single-byte instruction with the slot index; the variable name never appears in the bytecode. Global access is thus a dynamic hash-table lookup by name; local access is a direct stack-slot read by numeric index. This bifurcation is encoded in the instruction set as separate opcodes with different operands and

execution paths (Figure 2).

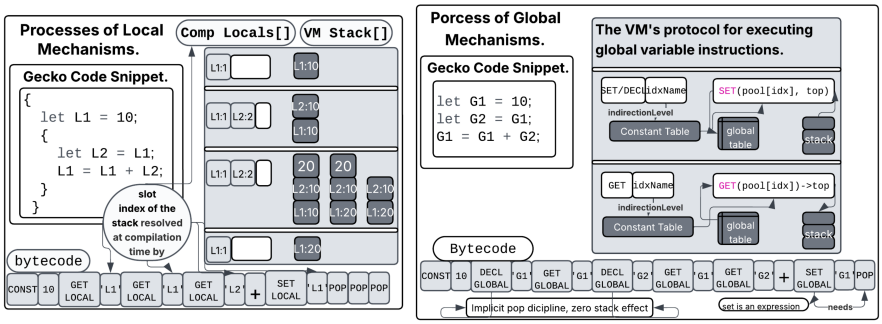

Figure 2: Local (left) and global (right) variable mechanisms. Left: the compile-time `locals[]` simulation and its correspondence with the runtime stack. Right: the two global-variable protocols (SET/DECL, GET) mediated by the constant table and globals hash table.

3.4 Control Flow: Non-Local Stack Obligations

The preceding principles generate *local* obligations: at the point of evaluation, storage, or resolution. Control flow introduces a qualitatively different kind: a jump at one bytecode position generates a stack obligation at another position, arbitrarily far away. The interpretation module's contribution is deliberately minimal — three primitives (`OP_JUMP`, `OP_JUMP_IF_FALSE`, `OP_LOOP`) that adjust only the instruction pointer. The VM has no notion of statement boundaries, does not balance the stack after a jump, and cannot validate jump targets semantically. All semantic responsibility rests with the compilation module, which must enforce the invariant that *every statement has net-zero stack effect*. A condition expression leaves its value on the stack; `OP_POP` instructions must appear on every execution path before the next statement begins. A missing pop leaves an unreferenced value below current operands; subsequent instructions silently consume the wrong operand with no VM-detectable error.

A second structural constraint compounds this. The single-pass compiler must emit `OP_JUMP_IF_FALSE` before compiling the then-branch body, without knowing its final size. Backpatching (Cooper et al., 2024) resolves this: emit the jump with a placeholder offset, record the position, compile the body, then overwrite the placeholder with the correct value. Backpatching is required for every forward jump and is a direct consequence of the linear IR from Principle 1: the same property that avoids tree allocation now requires retroactive jump-target repair. The denotational semantics of each construct (Sebesta, 2019) determines precisely where each pop and jump must fall; the resulting bytecode must reproduce the construct's semantics on every execution path using only the three VM primitives (Figure 3). With conditional branching and unbounded iteration in place, the interpreter reaches Turing completeness (Hopcroft et al.,

2001), making control flow the principle that transforms an expression evaluator into a general-purpose machine.

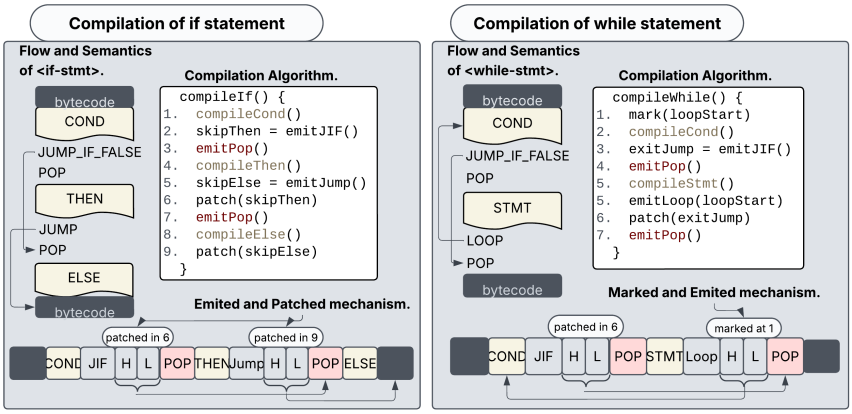

Figure 3: Compilation algorithms and bytecode for if-else and while statements. Each panel shows the semantic structure, the single-pass compilation algorithm, and the emitted bytecode with backpatched offsets. Steps in red correspond to `OP_POP` emissions that maintain the net-zero stack invariant on each execution path.

4 Conclusion

The four principles collectively reveal that the internal representation is not a passive data format but an *architectural contract*: postorder bytecode forces a LIFO execution stack; local slot allocation forces a stack layout the VM must maintain at exact offsets; uniform `Value` entries force uniform stack slots; placeholder jump offsets force a backpatching protocol. In each case, a compiler decision is crystallized into a VM obligation. These decisions propagate and are not independent: choosing a stack-based VM (a tradeoff still actively revisited under modern JIT compilation (Shi et al., 2008; Šimek et al., 2025)) implies postorder emission; postorder emission enables compile-time stack simulation; stack simulation enables slot-indexed local access. Existing literature (Aho et al., 1986; Appel, 2002; Ierusalimschy et al., 2005; Nystrom, 2021; Titzer, 2022) explains how individual components work; this framework explains why they must be built the way they are — making the propagation of compiler decisions through the IR the fundamental organizing mechanism of interpreter architecture. Future work includes applying the framework to other interpreters (CPython, Lua, Wren), extending it to closures and first-class functions (which introduce upvalue capture and heap-lifted variables not covered here), and formalizing the dependency structure as a machine-checkable architectural model.

Declaración sobre el uso de IA

Durante la preparación de este trabajo, el autor utilizó Claude (Anthropic) con el fin de mejorar el lenguaje y la legibilidad del manuscrito. Tras utilizar esta herramienta, el autor revisó y editó el contenido según fue necesario y asume plena responsabilidad por el contenido de la publicación.

References

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Addison-Wesley.

Appel, A. W. (2002). *Modern compiler implementation in Java*. Cambridge University Press.

Cooper, K. D. y Torczon, L. (2024). *Engineering a compiler* (3.ª ed.). Morgan Kaufmann.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation* (2nd ed.). Addison-Wesley.

Ierusalimschy, R., de Figueiredo, L. H., & Celes, W. (2005). The implementation of Lua 5.0. *Journal of Universal Computer Science*, *11*(7), 1159–1176.

Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.

Pratt, V. R. (1973). Top down operator precedence. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (pp. 41–51). ACM.

Sebesta, R. W. (2019). *Concepts of programming languages* (12th ed.). Pearson.

Shi, Y., Casey, K., Ertl, M. A., & Gregg, D. (2008). Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, *4*(4), Article 21.

Šimek, B., Fiala, D., & Dostal, M. (2025). Register-based and stack-based virtual machines: Which perform better in JIT compilation scenarios? *Software: Practice and Experience*, *55*(11), 1896–1910.

Stallings, W. (2016). *Computer organization and architecture* (10th ed.). Pearson.

Strömberg Skott, K. (2022). *VM instruction decoding using C unions in stack and register architectures* [Bachelor's thesis, Malmö University]. DiVA. https://www.diva-portal.org/smash/get/diva2:1664943/FULLTEXT02

Titzer, B. L. (2022). A fast in-place interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages*, *6*(OOPSLA2), 646–672.