A Stack-Based Bytecode Virtual Machine for the Lattice Programming Language

Alex Jokela
alex.c.jokela@gmail.com

February 2026

Abstract

The Lattice language's original tree-walking interpreter incurred per-node dispatch overhead and required AST retention for concurrency primitives. We present a bytecode virtual machine that achieves full feature parity—including the phase system, concurrent `scope`/`spawn`/`select`, exception handling, and modules—with significantly lower dispatch overhead. Key contributions include: a 100-opcode instruction set with wide constant variants and specialized integer fast paths, a single-pass compiler with upvalue-based lexical closures, pre-compiled sub-chunks for concurrency primitives that eliminate all runtime AST dependency, an ephemeral bump arena for short-lived string temporaries, a binary serialization format (`.latc`) for ahead-of-time distribution, and a self-hosted compiler written in Lattice itself. The implementation is validated by 815 tests passing under AddressSanitizer with zero memory errors.

1 Introduction

The Lattice programming language [1] introduces a *phase system* in which every runtime value carries a phase tag—either fluid (mutable) or crystal (deeply immutable)—with explicit transitions mediated by **freeze**, **thaw**, **clone**, and **forge** operations. The original implementation described in [1] is a tree-walking interpreter: the evaluator traverses AST nodes directly, dispatching on each node's tag. While straightforward, this approach has three significant limitations.

Interpreter overhead. Each AST node visit incurs a switch dispatch, pointer chasing through the tree structure, and recursive function call overhead. For tight loops and deeply nested expressions, these costs dominate execution time.

AST retention for concurrency. Lattice's structured concurrency primitives—`scope`, `spawn`, and `select`—require evaluating sub-expressions in spawned threads. In the tree-walker, this means the AST must remain live for the duration of concurrent execution, preventing the program representation from being discarded after compilation.

Inability to serialize. A tree-walking interpreter ties execution to the AST, making it impossible to compile a program once and distribute the compiled artifact. Users must ship source code and re-parse on every invocation.

This paper presents a bytecode virtual machine that addresses all three limitations while maintaining full feature parity with the tree-walker. The VM has been the default execution mode since Lattice v0.3.5; the tree-walker remains available via the `--tree-walk` flag.

Contributions.

1. A 100-opcode instruction set with variable-length encoding, wide constant variants for programs exceeding 256 constants, and specialized integer operations (§2).
2. A single-pass bytecode compiler with upvalue-based closures, constant folding, and peephole optimizations (§3, §4).
3. Pre-compiled sub-chunks for concurrency primitives, eliminating all runtime AST dependency (§6).
4. A binary serialization format (`.latc`) enabling ahead-of-time compilation and distribution (§7).
5. A self-hosted compiler written in Lattice (§8).
6. Empirical validation via 815 tests passing under both normal and AddressSanitizer builds (§9).

2 Instruction Set Architecture

2.1 Design Overview

The Lattice VM uses a stack-based architecture with variable-length instruction encoding. Each instruction begins with a single-byte opcode (values 0–99), optionally followed by one or more operand bytes. The instruction set comprises 100 opcodes organized into 14 functional categories (Table 1).

We chose a stack-based design over a register-based design (as used by Lua 5 [3]) for several reasons: (1) simpler code generation—the compiler never needs to perform register allocation; (2) more compact bytecode—stack instructions are typically 1–3 bytes versus 4 bytes for register triples; and (3) natural fit for expression evaluation in a dynamically typed language where operand types are not known at compile time.

2.2 Opcode Categories

Table 1: Instruction set overview (100 opcodes, indices 0–99).

Category	**Representative opcodes**	**Count**
Stack manipulation	`OP_CONSTANT`, `OP_NIL`, `OP_POP`, `OP_DUP`, `OP_SWAP`	8
Arithmetic/logical	`OP_ADD`, `OP_SUB`, `OP_MUL`, `OP_DIV`, `OP_NEG`, `OP_NOT`	7
Bitwise	`OP_BIT_AND`, `OP_BIT_OR`, `OP_LSHIFT`, `OP_RSHIFT`	6
Comparison	`OP_EQ`, `OP_NEQ`, `OP_LT`, `OP_GT`, `OP_LTEQ`, `OP_GTEQ`	6
String	`OP_CONCAT`	1
Variables	`OP_GET_LOCAL`, `OP_SET_LOCAL`, `OP_GET/SET_GLOBAL`, `OP_GET/SET_UPVALUE`	8
Control flow	`OP_JUMP`, `OP_JUMP_IF_FALSE`, `OP_LOOP`	5
Functions	`OP_CALL`, `OP_CLOSURE`, `OP_RETURN`	3
Iterators	`OP_ITER_INIT`, `OP_ITER_NEXT`	2
Data structures	`OP_BUILD_ARRAY`, `OP_INDEX`, `OP_INVOKE`, `OP_GET_FIELD`	15
Exceptions/defer	`OP_PUSH_EXCEPTION_HANDLER`, `OP_THROW`, `OP_DEFER_PUSH`	6
Phase system	`OP_FREEZE`, `OP_THAW`, `OP_REACT`, `OP_BOND`, `OP_SEED`	14
Builtins/modules	`OP_PRINT`, `OP_IMPORT`	2
Concurrency	`OP_SCOPE`, `OP_SELECT`	2
Integer fast paths	`OP_INC_LOCAL`, `OP_ADD_INT`, `OP_LOAD_INT8`	8
Wide variants	`OP_CONSTANT_16`, `OP_CLOSURE_16`	5
Special	`OP_RESET_EPHEMERAL`, `OP_HALT`	2
Total		**100**

2.3 Encoding Formats

Instructions use four encoding formats, shown in Figure 1:

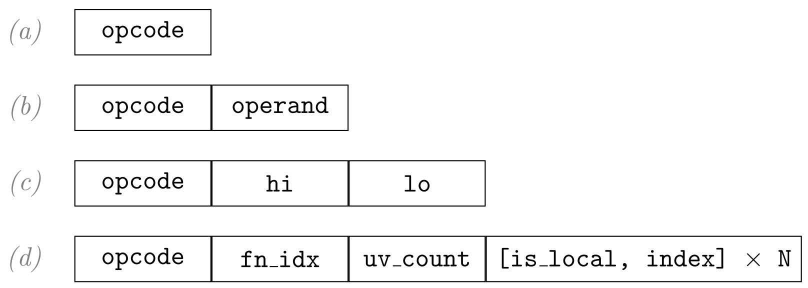

Figure 1: Instruction encoding formats: (a) zero-operand, (b) single-operand, (c) 16-bit big-endian operand, (d) variable-length (`OP_CLOSURE`).

Most instructions use format (a) or (b). Jump instructions use format (c) with a 16-bit signed offset. The wide variants (`OP_CONSTANT_16`, `OP_GET_GLOBAL_16`, etc.) also use format (c), addressing up to 65,536 constants per chunk. The `OP_CLOSURE` and `OP_SCOPE`/`OP_SELECT` instructions use format (d) with variable-length trailing data.

2.4 The Chunk Data Structure

Each compiled function (or top-level script) is represented by a `Chunk`:

```c
typedef struct {
    uint8_t  *code;          // bytecode stream
    size_t    code_len, code_cap;
    LatValue *constants;     // constant pool
    size_t    const_len, const_cap;
    int      *lines;         // source line per byte
    size_t    lines_len, lines_cap;
    char    **local_names;  // slot → variable name
    size_t    local_name_cap;
    char     *name;          // function name (debug)
} Chunk;
```

Listing 1: The Chunk structure.

The `constants` pool stores literal values (integers, floats, strings) and, crucially, nested function chunks. A compiled function is stored as a `VAL_CLOSURE` with `body = NULL` and `native_fn = Chunk*`, a convention we detail in §4. The `local_names` array provides debug information mapping stack slots to variable names, used both for error reporting and for exporting locals to the environment during concurrent execution (§6).

2.5 Detailed Opcode Listing

Table 2 provides a complete listing of all 100 opcodes organized by category, with their numeric values and operand formats.

Table 2: Complete opcode listing with numeric values (0–99).

Range	Category	Opcodes	Format
0–7	Stack	`CONSTANT`, `NIL`, `TRUE`, `FALSE`, `UNIT`, `POP`, `DUP`, `SWAP`	(a)/(b)
8–14	Arith	`ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `NEG`, `NOT`	(a)
15–20	Bitwise	`BIT_AND/OR/XOR/NOT`, `LSHIFT`, `RSHIFT`	(a)
21–26	Compare	`EQ`, `NEQ`, `LT`, `GT`, `LTEQ`, `GTEQ`	(a)
27	String	`CONCAT`	(a)
28–35	Vars	`GET/SET_LOCAL`, `GET/SET/DEFINE_GLOBAL`, `GET/SET_UPVALUE`, `CLOSE_UPVALUE`	(b)
36–40	Jumps	`JUMP`, `JUMP_IF_FALSE/TRUE/NOT_NIL`, `LOOP`	(c)
41–43	Funcs	`CALL`, `CLOSURE`, `RETURN`	(b)/(d)
44–45	Iter	`ITER_INIT`, `ITER_NEXT`	(a)/(c)
46–60	Data	`BUILD_*`, `INDEX`, `SET_INDEX`, `GET/SET_FIELD`, `INVOKE*`	(b)/multi
61–64	Except	`PUSH/POP_EXCEPTION_HANDLER`, `THROW`, `TRY_UNWRAP`	(c)/(a)
65–66	Defer	`DEFER_PUSH`, `DEFER_RUN`	(c)/(a)
67–80	Phase	`FREEZE`, `THAW`, `CLONE`, `MARK_FLUID`, `REACT`, etc.	(a)/multi
81–82	Misc	`PRINT`, `IMPORT`	(b)
83–84	Concur	`SCOPE`, `SELECT`	(d)
85–92	IntOpt	`INC/DEC_LOCAL`, `ADD/SUB/MUL/LT/LTEQ_INT`, `LOAD_INT8`	(b)/(a)
93–97	Wide	`CONSTANT_16`, `GET/SET/DEFINE_GLOBAL_16`, `CLOSURE_16`	(c)/(d)
98–99	Special	`RESET_EPHEMERAL`, `HALT`	(a)

Several design choices merit discussion. The integer fast-path opcodes (85–92) bypass the type-checking overhead of their generic counterparts: `OP_ADD_INT` assumes both operands are `VAL_INT` and produces an `int64_t` result directly, while the generic `OP_ADD` must check for string concatenation, float promotion, and type errors. The OP_LOAD_INT8 instruction encodes a signed byte directly in the instruction stream, avoiding constant pool allocation for the most common small integer values.

The phase system opcodes (67–80) include both simple operations (OP_FREEZE, OP_THAW, OP_CLONE) that operate on the top of stack, and variable-targeted operations (OP_FREEZE_VAR, OP_THAW_VAR, OP_SUBLIMATE_VAR) that take a variable name index and location descriptor. The latter enable the compiler to emit OP_DUP + OP_SET write-back sequences when `freeze(x)` or `thaw(x)` is called with an identifier argument, ensuring the variable itself is updated rather than just the stack copy.

3 The Bytecode Compiler

3.1 Architecture

The compiler performs a single-pass walk over the AST produced by the parser. It maintains a chain of `Compiler` structs linked via `enclosing` pointers, one per lexical scope that introduces a new function:

```c
typedef struct Compiler {
    struct Compiler *enclosing;
    Chunk       *chunk;
    FunctionType type;     // SCRIPT, FUNCTION, CLOSURE
    char        *func_name;
    int          arity;
    Local       *locals;
    size_t       local_count, local_cap;
    CompilerUpvalue *upvalues;
    size_t       upvalue_count;
    int          scope_depth;
    // break/continue tracking, contracts ...
} Compiler;
```

Listing 2: The Compiler structure (simplified).

The top-level script uses FUNC_SCRIPT; named functions and closures use FUNC_FUNCTION and FUNC_CLOSURE, respectively.

3.2 Scope and Local Variables

Local variables are tracked by the `Local` struct, which records the variable name, scope depth, and whether it has been captured as an upvalue:

```c
typedef struct {
    char *name;
    int   depth;        // -1 = uninitialized
    bool  is_captured;
} Local;
```

Slot 0 is reserved for the function reference itself (or `"self"` in methods). When a scope ends, the compiler emits OP_POP for uncaptured locals and OP_CLOSE_UPVALUE for captured ones, ensuring upvalues are properly closed before the stack slot is reclaimed.

3.3 Compilation Modes

The compiler supports three entry points for different use cases:

`compile()` Standard file compilation. After compiling all declarations, emits an implicit call to `main()` if one is defined.

`compile_module()` Module compilation for `import`. Identical to `compile()` but omits the auto-call to `main()`.

`compile_repl()` REPL compilation. Preserves the last expression on the stack as the iteration's return value (displayed with `=>` prefix). Does not free the known-enum table between iterations, allowing enum declarations to persist across REPL lines.

3.4 Compile-Time Optimizations

The compiler implements several optimizations during code generation:

Constant folding. Binary operations on literal operands are evaluated at compile time. For instance, `3 + 4` emits a single `OP_LOAD_INT8 7` rather than two loads and an `OP_ADD`. The compiler folds addition, subtraction, multiplication, division, modulo, and all comparison operators for both integer and floating-point literal pairs.

Small-integer encoding. Integer literals in the range $[-128, 127]$ are emitted as `OP_LOAD_INT8` with a signed byte operand, avoiding a constant pool entry entirely. This two-byte instruction replaces what would otherwise be a three-byte `OP_CONSTANT` plus an eight-byte constant pool entry.

Increment peephole. The pattern `x += 1` (desugared by the parser to an assignment with a binary addition) is detected and emitted as the single-byte `OP_INC_LOCAL`, which modifies the stack slot in place without pushing or popping. Similarly, `x -= 1` becomes `OP_DEC_LOCAL`.

Statement-boundary arena reset. Each statement is compiled through a `compile_stmt_reset()` wrapper that appends `OP_RESET_EPHEMERAL` after the statement's bytecode. This instruction resets the ephemeral bump arena (described in [2]), reclaiming short-lived string temporaries at well-defined points.

4 Closures and Upvalues

4.1 Compilation

When the compiler encounters a function definition inside another function, it creates a nested `Compiler` linked to the enclosing one. Variable references in the inner function are resolved through a three-tier lookup:

1. **Local:** Scan the current compiler's `locals` array.
2. **Upvalue:** Recursively invoke `resolve_upvalue()` on the enclosing compiler. If the variable is a local in the immediate enclosing function, mark it as captured (`is_captured = true`) and call `add_upvalue()` with `is_local = true`. If it is itself an upvalue of the enclosing function, call `add_upvalue()` with `is_local = false`.

3. **Global:** Fall through to `OP_GET_GLOBAL`.

The `add_upvalue()` function deduplicates: if an upvalue with the same index and locality flag already exists, its index is reused.

4.2 OP_CLOSURE Encoding

The `OP_CLOSURE` instruction uses variable-length encoding:

42	fn_idx	uv_cnt	is_loc	index	···

The `fn_idx` operand indexes into the chunk's constant pool, where the function's compiled `Chunk` is stored as a `VAL_CLOSURE` constant. The `uv_cnt` byte gives the number of upvalue descriptors that follow, each a pair `[is_local, index]`. The wide variant `OP_CLOSURE_16` uses a two-byte big-endian function index.

4.3 Runtime Representation

Open upvalues are represented by the `ObjUpvalue` struct:

```c
typedef struct ObjUpvalue {
    LatValue *location;       // → stack slot (open)
    LatValue  closed;         //     or &closed (closed)
    struct ObjUpvalue *next; // linked list
} ObjUpvalue;
```

Listing 3: The ObjUpvalue structure.

While a variable is still on the stack, `location` points directly into the stack slot. The VM maintains a singly linked list of all open upvalues, sorted by stack address in descending order. This ordering enables efficient insertion and lookup during capture.

4.4 Capture and Closure

When executing `OP_CLOSURE`, the VM processes each upvalue descriptor:

- If `is_local = 1`: call `capture_upvalue()`, which walks the open upvalue list to find an existing upvalue pointing at the same stack slot. If found, it is reused (ensuring multiple closures share the same upvalue). Otherwise, a new `ObjUpvalue` is created and inserted into the sorted list.
- If `is_local = 0`: copy the upvalue pointer from the enclosing frame's upvalue array at the given index.

When a local variable goes out of scope (`OP_CLOSE_UPVALUE`), the VM calls `close_upvalues()`: each open upvalue whose `location` is at or above the target stack address is *closed* by deep-cloning the pointed-to value into `closed` and redirecting `location` to `&closed`. This approach, inspired by Lua's upvalue design [3] and refined by Nystrom [4], ensures that closed-over variables survive beyond their lexical scope without garbage collector integration.

4.5 The Storage Hack

The bytecode VM repurposes existing fields of the `LatValue` closure representation to avoid adding VM-specific fields:

- `closure.body` is `NULL` for compiled functions (the tree-walker uses this field for the AST body).
- `closure.native_fn` stores a `Chunk*` pointer to the compiled function body.
- `closure.captured_env` is cast to `ObjUpvalue**`, storing the array of captured upvalue pointers.
- `region_id` stores the upvalue count (since compiled closures do not participate in region tracking).

To distinguish VM-native C functions (registered via `vm_register_native()`) from compiled closures, a sentinel value is used:

```
#define VM_NATIVE_MARKER ((struct Expr **)(uintptr_t)0x1)
```

A closure with `body == NULL`, `native_fn != NULL`, and `default_values == VM_NATIVE_MARKER` is a C-native function; one with `body == NULL` and `native_fn != NULL` but without the marker is a compiled bytecode closure.

4.6 Walkthrough: Compilation and Execution

To illustrate the complete pipeline, consider the following Lattice program that demonstrates closure capture:

```
fn make_counter() {
    flux count = 0
    return fn() {
        count += 1
        return count
    }
}
let c = make_counter()
print(c())  // 1
print(c())  // 2
```

Listing 4: A closure capturing a local variable.

Compilation. The compiler processes `make_counter` by creating a nested `Compiler`. The inner closure references `count`, which is a local in the enclosing function. The compiler:

1. Resolves `count` via `resolve_upvalue()`, which finds it as local slot 1 in `make_counter` and marks it as captured.
2. Calls `add_upvalue()` with `index=1, is_local=true`.
3. Compiles `count += 1` as: `OP_GET_UPVALUE` 0, `OP_LOAD_INT8` 1, `OP_ADD`, `OP_SET_UPVALUE` 0, `OP_POP`.

4. Stores the inner function's chunk as a constant in `make_counter`'s chunk.
5. Emits `OP_CLOSURE [fn_idx, 1, 1, 1]` — one upvalue, `is_local=1`, `index=1`.

When `make_counter` returns, the compiler emits `OP_CLOSE_UPVALUE` for the captured local `count` (rather than a plain `OP_POP`).

Execution. At runtime:

1. `OP_CLOSURE` executes: `capture_upvalue()` creates an `ObjUpvalue` pointing at `frame->slots[1]` (where `count` lives), and inserts it into the VM's open upvalue list.
2. When `make_counter` returns, `OP_CLOSE_UPVALUE` calls `close_upvalues()`: the upvalue's `closed` field receives a deep clone of the stack value, and `location` is redirected to `&closed`.
3. Each call to `c()` accesses `count` via `OP_GET_UPVALUE` 0, which dereferences `location` — now pointing to the heap-allocated `closed` field.
4. Successive calls correctly see the incrementing value because `OP_SET_UPVALUE` 0 writes through the same `location` pointer.

5 VM Execution Engine

5.1 VM State

The VM maintains the following state:

```
typedef struct {
    CallFrame  frames[256];     // call frame stack
    size_t     frame_count;
    LatValue   stack[4096];     // value stack
    LatValue  *stack_top;
    Env       *env;             // global variables
    ObjUpvalue *open_upvalues; // linked list
    ExceptionHandler handlers[64];
    size_t     handler_count;
    VMDeferEntry defers[256];
    size_t     defer_count;
    LatValue   fast_args[16];  // pre-allocated
    BumpArena *ephemeral;      // arena for temps
    LatMap     module_cache;
    // phase system: tracked vars, pressures,
    //   reactions, bonds, seeds ...
} VM;
```

Listing 5: VM structure (abridged).

Each call frame records the chunk being executed, the instruction pointer, a base pointer into the value stack, and the frame's upvalue array:

```
typedef struct {
    Chunk       *chunk;
    uint8_t     *ip;
    LatValue    *slots;  // frame base on stack
    ObjUpvalue **upvalues;
    size_t       upvalue_count;
} CallFrame;
```

5.2 Dispatch Loop

The VM's inner loop reads one opcode per iteration and dispatches to the corresponding handler. On compilers supporting the GCC/Clang labels-as-values extension, we use *computed goto* via a dispatch table:

```
#define READ_BYTE() (*frame->ip++)
#define READ_U16()  (frame->ip += 2, \
    (uint16_t)((frame->ip[-2] << 8) | frame->ip[-1]))

#ifdef VM_USE_COMPUTED_GOTO
  static void *dispatch_table[] = {
    [OP_CONSTANT] = &&lbl_OP_CONSTANT,
    [OP_NIL]      = &&lbl_OP_NIL,
    // ... all 100 entries
  };
#endif

for (;;) {
    uint8_t op = READ_BYTE();
#ifdef VM_USE_COMPUTED_GOTO
    goto *dispatch_table[op];
#endif
    switch (op) {
#ifdef VM_USE_COMPUTED_GOTO
        lbl_OP_CONSTANT:
#endif
        case OP_CONSTANT: { /* ... */ }
        // ...
    }
}
```

Listing 6: Dispatch loop (simplified).

Computed goto eliminates the overhead of the switch statement's bounds check and branch table lookup, replacing it with a single indirect jump per instruction. On platforms without the extension, the compiler falls back to a standard switch.

5.3 Call Protocol

The `OP_CALL` instruction discriminates three callee types:

Native functions. If the callee is marked with `VM_NATIVE_MARKER`, the VM pops arguments into a pre-allocated `fast_args[16]` buffer (falling back to `malloc` for calls with more than 16 arguments), invokes the C function pointer, and pushes the return value. This fast path avoids allocating a new call frame entirely.

Compiled closures. The VM first promotes all ephemeral values in the current frame to prevent the callee's `OP_RESET_EPHEMERAL` from invalidating the caller's temporaries. It then pushes a new `CallFrame` pointing to the callee's chunk, with the instruction pointer at byte 0 and the stack base set to the callee's position on the stack (so `slots[0]` is the closure itself, followed by arguments in `slots[1..arity]`).

Callable structs. If the callee is a struct value, the VM looks for a field named with the struct's type (constructor convention) and dispatches accordingly.

5.4 Exception Handling

Exception handlers are registered by `OP_PUSH_EXCEPTION_HANDLER`, which records the current instruction pointer, chunk, call frame index, and stack top. When `OP_THROW` executes:

1. The nearest handler is popped from the handler stack.
2. The call frame stack is unwound to the handler's frame index.
3. The value stack is restored to the handler's recorded stack top.
4. The error value is pushed onto the stack.
5. Execution resumes at the handler's saved IP (the catch block).

Deferred blocks interact correctly: `OP_DEFER_RUN` executes all defer entries registered at or above the current frame before the frame is popped by `OP_RETURN`.

5.5 Iterators

The `OP_ITER_INIT` instruction converts a range or array value into an internal iterator representation occupying two stack slots (the collection and a cursor index). The `OP_ITER_NEXT` instruction advances the cursor, pushes the next element, or jumps to the specified offset when exhausted. This design avoids allocating a closure object for `for` loops, unlike the tree-walker which used closure-based iterators.

5.6 Reference-Counted Shared Values: `Ref<T>`

Lattice's deep-clone-on-read semantics eliminate aliasing: every variable read produces an independent copy, and mutations never propagate across scope boundaries. While this provides strong safety guarantees, it makes shared mutable state impossible by default. The `Ref<T>` type is an explicit opt-in to reference semantics.

A `Ref` wraps a value in a reference-counted, heap-allocated container:

```
struct LatRef {
    LatValue value;       // wrapped inner value
    size_t   refcount;    // reference count
};
```

Listing 7: The LatRef structure.

When a `VAL_REF` is cloned—which occurs on every variable read, like all other types—the VM calls `ref_retain()` to increment the reference count and copies the pointer. It does *not* deep-clone the inner value. Multiple copies of a `Ref` therefore share the same underlying `LatRef`, and mutations through any copy are visible through all others.

The VM provides transparent proxying for `Ref` values. The `OP_INDEX`, `OP_SET_INDEX`, and `OP_INVOKE` handlers check for `VAL_REF` and delegate to the inner value: indexing into a `Ref<Array>` indexes the inner array, calling `.push()` on a `Ref<Array>` mutates the inner array directly. This transparency extends to the builtin dispatch in `vm_invoke_builtin()`, which proxies map operations (`get`, `set`, `keys`, `values`), array operations (`push`, `pop`), and other type-specific methods through the `Ref` wrapper.

`Ref`-specific methods include `get()`/`deref()` (deep-clone the inner value out), `set(v)` (replace the inner value), and `inner_type()` (return the type name of the wrapped value). A `Ref` is created via `Ref::new(value)`, registered as a native function in the VM.

The phase system applies to `Ref` values: freezing a `Ref` sets its phase tag to `crystal`, and all mutation methods (`set()`, `push()`, index assignment) check the phase and produce a runtime error if the `Ref` is frozen. Frozen `Ref` values are safe to share across concurrent boundaries, as they provide immutable handles to immutable data.

`Ref<T>` introduces a third memory management strategy alongside the dual-heap architecture (mark-and-sweep for fluid values, arena-based regions for crystal values) and the ephemeral bump arena. Reference counting is simple and deterministic: `ref_retain()` on clone, `ref_release()` on free, with the inner value deallocated when the count reaches zero. Since `Ref` is the uncommon case—most Lattice code relies on value semantics—cycle collection has not been necessary.

5.7 Architecture Overview

Figure 2 illustrates the VM's runtime data structures and their relationships.

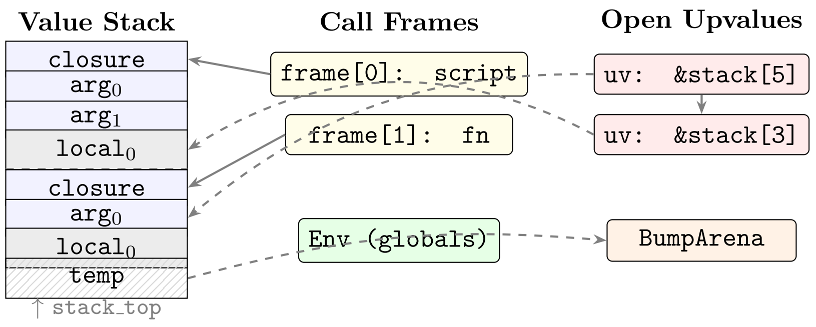

Figure 2: VM runtime architecture: value stack with frame boundaries, call frame stack, global environment, open upvalue linked list, and ephemeral bump arena.

6 Compiling Concurrency Primitives

6.1 The Problem

Lattice provides structured concurrency through three primitives: `scope` (defines a concurrent region), `spawn` (launches a lightweight task within a scope), and `select` (multiplexes over channels). In the tree-walking interpreter, these primitives evaluate sub-expressions by passing AST node pointers to spawned threads. This creates a hard dependency: the AST must remain live for the duration of concurrent execution, and the program cannot be serialized to bytecode alone.

6.2 Pre-Compiled Sub-Chunks

The solution is to compile each concurrent body into a standalone `Chunk` at compile time. The compiler provides two helper functions:

- `compile_sub_body(stmts, count, line)`: Saves the current compiler, creates a fresh `Compiler` with type `FUNC_SCRIPT`, compiles the statement list, emits `OP_HALT`, restores the outer compiler, and returns the resulting `Chunk`.
- `compile_sub_expr(expr, line)`: Similar, but compiles a single expression and emits `OP_HALT` after it.

The resulting chunk is stored in the parent chunk's constant pool as a `VAL_CLOSURE` constant (using the same storage hack: `body = NULL`, `native_fn = Chunk*`).

6.3 OP_SCOPE Encoding and Execution

The `OP_SCOPE` instruction uses variable-length encoding:

83	spawn_cnt	sync_idx	spawn_idx$_0$ ...spawn_idx$_{N-1}$

Each index refers to a constant pool entry containing a pre-compiled sub-chunk. At runtime, `OP_SCOPE` executes as follows:

1. **Export locals.** Current frame locals are exported to the global environment using the `local_names` debug table, so sub-chunks (which run as independent scripts) can access them via `OP_GET_GLOBAL`.
2. **Run sync body.** If `sync_idx` is valid (not `0xFF`), the corresponding chunk is executed via a recursive `vm_run()` call on the same VM.
3. **Spawn threads.** For each spawn index, a child VM is created via `vm_clone_for_thread()` (which clones the environment but shares struct metadata), and the spawn body is executed on a new `pthread`.
4. **Join.** All threads are joined; the first error (if any) is propagated.

6.4 OP_SELECT Encoding and Execution

The `OP_SELECT` instruction encodes one record per arm:

84	`arm_cnt`	`flags`	`chan_idx`	`body_idx`	`bind_idx`	$\times N$

Each arm specifies: flags (send vs. receive, default arm), the channel expression's sub-chunk index, the body sub-chunk index, and a binding name constant index. At runtime, the VM evaluates channel expressions, polls for readiness, executes the winning arm's body with the received value bound to the specified name, or falls through to a default arm.

6.5 Local Variable Export

Because sub-chunks are compiled as `FUNC_SCRIPT` without lexical access to the parent's locals, the VM must explicitly export the parent frame's live locals into the global environment before running any sub-chunk. This is accomplished by iterating over all active frames' slots, using the `chunk->local_names` debug table to recover variable names, and calling `env_define()` for each. A scope is pushed before export and popped after all sub-chunks complete, ensuring the exported bindings do not leak.

7 Bytecode Serialization

7.1 File Format

The `.latc` binary format enables ahead-of-time compilation: a program can be compiled once and the resulting bytecode distributed without source. The format begins with an 8-byte header:

`'L'`	`'A'`	`'T'`	`'C'`	`version (u16)`	`reserved (u16)`

The magic bytes `"LATC"` identify the format; the 16-bit format version (currently 1) enables future format evolution; the reserved field is zero. All multi-byte integers use little-endian encoding.

7.2 Chunk Encoding

Following the header, the top-level chunk is serialized recursively:

1. **Code section:** `code_len` (u32) followed by raw bytecode bytes.
2. **Line section:** `lines_len` (u32) followed by `lines_len` line numbers (u32 each), providing source mapping.
3. **Constants section:** `const_len` (u32) followed by typed constants.
4. **Local names section:** `name_count` (u32) followed by presence flags and strings.

7.3 Constant Type Tags

Each constant is prefixed by a one-byte type tag:

Tag	Type and encoding
0	`Int`: 8-byte signed little-endian
1	`Float`: 8-byte IEEE 754 double
2	`Bool`: 1 byte (0 or 1)
3	`String`: length-prefixed (u32 + bytes)
4	`Nil`: no payload
5	`Unit`: no payload
6	`Closure`: param_count (u32) + has_variadic (u8) + recursive chunk

The `Closure` tag enables recursive serialization: a function constant contains its parameter metadata followed by a complete serialized sub-chunk. This naturally handles arbitrary nesting depth.

7.4 Loading and Validation

The loader (`chunk_load()`/`chunk_deserialize()`) performs bounds-checked reading through a `ByteReader` struct that tracks position and length. Validation includes magic byte verification, format version checking, and graceful error reporting for truncated or malformed inputs.

7.5 Example: Byte Layout

Figure 3 shows the byte layout of a minimal `.latc` file compiled from `print(42)`.

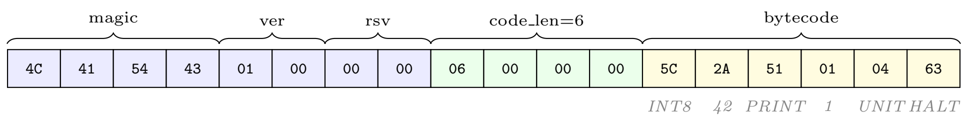

Figure 3: Byte layout of a `.latc` file for `print(42)`. Header bytes shown in blue, code length in green, bytecode in yellow. The `LOAD_INT8` 42 / `PRINT` 1 / `UNIT` / `HALT` sequence is followed by line number and constant sections (not shown).

7.6 CLI Integration

The serialization integrates with the command-line interface:

- `clat compile input.lat [-o out.latc]` compiles source to bytecode.
- `clat file.latc` auto-detects the `.latc` suffix and loads pre-compiled bytecode directly, skipping parsing and compilation.
- `clat file.lat` compiles and executes in a single step (the default).

8 The Self-Hosted Compiler

8.1 Overview

The file `compiler/latc.lat` is a self-hosted bytecode compiler written entirely in Lattice. At approximately 2,060 lines, it reads `.lat` source, produces bytecode, and writes `.latc` files using the same binary format as the C implementation. Usage:

```
$ clat compiler/latc.lat input.lat output.latc
$ clat output.latc   # run compiled bytecode
```

8.2 Architecture

The self-hosted compiler is organized into 12 sections:

1. **Opcode constants** defined as global integers.
2. **Compiler state** using parallel global arrays: `code`, `constants`, `c_lines`, and `local_names` (since structs and maps are pass-by-value in Lattice, a struct-based compiler state would not propagate mutations through function calls).
3. **Lexing** via the built-in `tokenize()` function, which returns an array of token objects.
4. **Recursive-descent parser** with functions for each grammar production (`parse_expression`, `parse_statement_or_expression`, etc.).
5. **Code emission** helpers (`emit_byte`, `emit_bytes`, `emit_jump`, `patch_jump`, `emit_loop`).
6. **Scope management** (`begin_scope`, `end_scope`, `add_local`, `resolve_local`, `resolve_upvalue`).
7. **Function compilation** with a compiler stack using `save_compiler`/`restore_compiler` to handle nested functions.
8. **Control flow** (if/else, while, loop, for, break, continue, match).
9. **Declarations** (struct, enum, try/catch, defer, import).
10. **Top-level entry** (`compile_program`).
11. **Serializer** writing the `.latc` binary format to a global `Buffer`.
12. **Main** entry point.

8.3 Language Constraint Workarounds

Writing a compiler in Lattice required several workarounds for language semantics that differ from C:

Parallel arrays for state. Because structs and maps are pass-by-value, the compiler uses parallel global arrays (`local_names[]`, `local_depths[]`, `local_captured[]`) rather than an array of structs. Mutations to global arrays propagate correctly.

Global serialization buffer. The `Buffer` type is also pass-by-value; to accumulate serialized bytes across function calls, a global `ser_buf` variable is used.

Compiler stack. Nested function compilation saves and restores compiler state via explicit `save_compiler()/restore_compiler()` functions that copy all global arrays to local temporaries and back.

No `else if`. Lattice requires `else { if ...}` for chained conditionals; the `match` expression is used for dispatch where possible.

Mandatory type annotations. Function parameters require type annotations (`fn foo(a: any)`), which the self-hosted compiler uses throughout.

8.4 Coverage

The self-hosted compiler currently handles: arithmetic and string expressions, variable declarations and assignments, functions and closures (with upvalue capture), control flow (if/else, while, loop, for, break, continue, match), structs, enums, exception handling (try/catch/throw), defer, string interpolation, and imports. Not yet implemented: concurrency primitives (`scope`/`spawn`/`select`) and advanced phase operations (`react`, `bond`, `seed`).

8.5 Bootstrapping Path

The current bootstrapping chain is:

$$\begin{tikzcd} \boxed{\texttt{latc.lat}} \arrow[r, "\text{interprets}"] & \boxed{\text{C VM}} \arrow[r, "\text{produces}"] & \boxed{\texttt{output.latc}} \arrow[r, "\text{executes}"] & \boxed{\text{C VM}} \end{tikzcd}$$

Full self-hosting (where `latc.lat` compiles itself) requires adding concurrency support and resolving the remaining feature gaps.

9 Evaluation

9.1 Test Suite

The implementation is validated by a comprehensive test suite of **815 tests** covering all VM features: arithmetic, closures, upvalues, phase transitions, exception handling, defer, iterators, data structures, concurrency, modules, bytecode serialization, and the self-hosted compiler. All 815 tests pass under both normal compilation and AddressSanitizer (ASan) builds, confirming the absence of memory errors including use-after-free, buffer overflows, and memory leaks in the test-exercised paths.

9.2 Correctness

The bytecode VM maintains full parity with the tree-walking interpreter. Both execution modes share the same test suite and produce identical results:

```
$ make test          # bytecode VM (default): 815 passed
$ make test TREE_WALK=1  # tree-walker: 815 passed
```

This dual-execution model serves as a cross-validation mechanism: any divergence between the two modes immediately surfaces as a test failure.

9.3 Feature Completeness

Table 3 summarizes feature coverage across both execution modes.

Table 3: Feature parity between tree-walker and bytecode VM.

Feature	**Tree-walker**	**Bytecode VM**
Phase system (freeze/thaw/clone/forge)	✓	✓
Closures with upvalues	✓	✓
Exception handling (try/catch/throw)	✓	✓
Defer blocks	✓	✓
Pattern matching	✓	✓
Structs with methods	✓	✓
Enums with payloads	✓	✓
Arrays, maps, tuples, sets, buffers	✓	✓
Iterators (for-in, ranges)	✓	✓
Module imports	✓	✓
Concurrency (scope/spawn/select)	✓	✓
Channels	✓	✓
Phase reactions/bonds/seeds	✓	✓
String interpolation	✓	✓
Contracts (require/ensure)	✓	✓
Variable tracking (history)	✓	✓
Bytecode serialization (.latc)	—	✓
Computed goto dispatch	—	✓
Ephemeral bump arena	—	✓
Specialized integer ops	—	✓

9.4 Test Distribution

Table 4 shows the distribution of tests across feature categories, illustrating the breadth of coverage.

Table 4: Approximate test distribution by feature category.

Category	**Approx. tests**
Core expressions and variables	85
Functions and closures	75
Control flow (if/while/for/match)	70
Data structures (array/map/struct)	95
Phase system	80
Exception handling and defer	45
String operations and interpolation	50
Concurrency (scope/spawn/select)	40
Module imports	30
Enums and pattern matching	55
Iterators and ranges	35
Bytecode serialization (.latc)	25
Integer optimizations	20
Contracts (require/ensure)	15
Miscellaneous (REPL, tracking, etc.)	95
Total	**815**

9.5 AddressSanitizer Validation

Running the full test suite under Clang's AddressSanitizer (`make asan`) enables dynamic detection of:

- **Heap buffer overflows:** Out-of-bounds reads or writes to heap-allocated arrays (constant pools, value arrays, etc.).
- **Use-after-free:** Accessing values or chunks that have been deallocated—particularly relevant for the upvalue system, where closed-over values must outlive their stack slots.
- **Stack buffer overflows:** Overrunning the fixed-size VM stacks (4096 value slots, 256 call frames).
- **Memory leaks:** Values or chunks that are allocated but never freed (detected via LeakSanitizer, enabled by default with ASan).

All 815 tests pass under ASan with zero reported errors, providing confidence in the correctness of memory management throughout the VM, compiler, and serialization subsystems.

9.6 Qualitative Assessment

The bytecode VM provides several structural advantages over the tree-walker:

- **Eliminated AST retention.** Concurrent execution no longer requires keeping AST nodes alive, thanks to pre-compiled sub-chunks. This enables bytecode serialization and reduces memory pressure for long-running concurrent programs.

- **Reduced dispatch overhead.** The computed goto dispatch loop replaces recursive AST node visits with a flat indirect-jump table, eliminating per-node function call overhead and improving branch prediction behavior.
- **Serialization.** The `.latc` format enables compile-once-run-anywhere workflows and eliminates parse/compile time on subsequent runs.
- **Foundation for optimization.** The bytecode representation provides a natural target for future optimizations including type specialization, register allocation, and JIT compilation.

10 Related Work

Lua 5. Ierusalimschy et al. [3] describe Lua's register-based bytecode VM, which was a direct inspiration for Lattice's upvalue design. Lua's `UpVal` structure (with open/closed states and a linked list of open upvalues sorted by stack level) is closely mirrored by Lattice's `ObjUpvalue`. The key difference is that Lattice uses a stack-based VM rather than a register-based one, accepting slightly larger bytecode in exchange for simpler code generation and natural expression evaluation.

CPython. CPython [5] uses a stack-based bytecode interpreter with `LOAD_FAST/STORE_FAST` for locals, a constant pool approach similar to Lattice's, and a `.pyc` serialization format. Lattice adds the phase system and first-class concurrency primitives as bytecode-level concepts, which CPython handles at the library level.

Crafting Interpreters. Nystrom's *Crafting Interpreters* [4] provides the pedagogical foundation for the upvalue closure mechanism (open-to-closed transition via linked list) that Lattice adopts. Lattice extends this with phase-aware value semantics: closing an upvalue performs a deep clone rather than a pointer copy, preserving phase tag integrity.

Ruby YARV. The YARV (Yet Another Ruby VM) project [6] replaced Ruby's tree-walking interpreter with a stack-based bytecode VM, motivated by the same performance concerns that drove Lattice's transition. YARV uses catch tables for exception handling; Lattice uses an explicit handler stack, which simplifies interaction with the defer mechanism.

V8 Ignition. The V8 JavaScript engine's Ignition interpreter [7] compiles JavaScript to a register-based bytecode that feeds into the TurboFan optimizing compiler. Lattice's bytecode similarly provides a foundation for future JIT compilation, though the current implementation is interpreter-only.

Erlang BEAM. The BEAM virtual machine [8] compiles Erlang to bytecode with first-class support for lightweight processes and message passing. Lattice's `OP_SCOPE` and `OP_SELECT` instructions serve an analogous role for structured concurrency.

WebAssembly. WebAssembly [9] uses a stack-based bytecode with a typed section-based binary format. Lattice's `.latc` format shares the constant-pool-plus-typed-sections approach, though at a higher level of abstraction (dynamically typed values rather than low-level types).

11 Conclusion

We have presented a bytecode virtual machine for the Lattice programming language that achieves full feature parity with the original tree-walking interpreter across all language features, including the phase system, structured concurrency, exception handling, and modules.

The key insight enabling the transition is that concurrency primitives can be *pre-compiled* into standalone sub-chunks at compile time, stored in the constant pool, and executed at runtime without any AST dependency. This eliminates the primary obstacle to bytecode compilation in a language with first-class structured concurrency.

The 100-opcode instruction set balances generality with targeted optimizations: wide constant variants handle large programs, specialized integer operations accelerate common loop patterns, and the ephemeral bump arena reclaims string temporaries at statement boundaries. The `.latc` serialization format enables ahead-of-time compilation, and the self-hosted compiler demonstrates that Lattice is expressive enough to implement its own toolchain.

Future work includes register allocation to reduce stack traffic, type-specialized dispatch paths guided by runtime profiling, tail call optimization for recursive patterns, constant pool deduplication across compilation units, and ultimately JIT compilation targeting the bytecode as an intermediate representation.

References

[1] A. Jokela, "The Lattice phase system: First-class immutability with dual-heap memory management," Technical report, 2026.

[2] A. Jokela, "Ephemeral bump arena for a bytecode virtual machine: Design and memory safety proof," Technical report, 2026.

[3] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The implementation of Lua 5.0," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, 2005.

[4] R. Nystrom, *Crafting Interpreters.* Genever Benning, 2021.

[5] Python Software Foundation, "CPython: The reference implementation of Python," `https://github.com/python/cpython`.

[6] K. Sasada, "YARV: Yet another RubyVM," In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 158–159, 2005.

[7] R. Hindley, "Firing up the Ignition interpreter," V8 Blog, 2016. `https://v8.dev/blog/ignition-interpreter`.

[8] J. Armstrong, "A history of Erlang," In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pp. 6-1–6-26, 2007.

[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200, 2017.

[10] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The evolution of Lua," In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pp. 2-1–2-26, 2007.

[11] A. Jokela, "Formal semantics of the Lattice phase system," Technical report, 2026.