lumbda.

A Lisp/Scheme-derived, just-in-time lambda language. Four implementation tiers with MOAD defect isolation. Workloads migrate across basic UNIX systems.

Feedback is all you need: continuations within a process, portals across processes, S-expressions across implementations, sockets across machines.

russell@unturf, TimeHexOn, foxhop, Zoë Trout

lumbda.com · uncloseai.com · permacomputer.com

Source: git.unturf.com/engineering/unturf/lumbda

April 2026

Abstract

License: AGPL-3.0-only · This implementation, its bytecode VM, & all associated code carry the GNU Affero General Public License v3.0 (only). You may use, modify, & distribute under those terms. No proprietary relicensing exists.

Lumbda is a Lisp/Scheme-derived language designed around one primitive and one discipline: **feedback** (a function that receives its own continuation composes every control-flow pattern) and **MOAD defect isolation** (every implementation audited against the five canonical Mother-of-all-Defects patterns, with every defect confined to its own implementation tier rather than propagating through shared infrastructure — see §12).

The rest of the design follows: one primitive becomes universal when a function receives its own continuation. A continuation lets a program loop, branch, yield, checkpoint, resume, & migrate. Every control flow pattern reduces to a continuation captured & invoked. Extend feedback across time (portals) & across implementations (source-as-wire-format) and you recover the full scope of computation without new primitives.

Lumbda ships in **four implementation tiers** — each independently built, each MOAD-isolated, each able to run every test in the shared functional suite byte-identically:

- **Python bytecode VM** — 3,743 lines, full first-class continuations, JSON portal, reference implementation
- **C interpreter** — tree-walker + bytecode VM, 9,164 lines of runtime C, JSON portal
- **C + x86_64 JIT** — pattern-matched native code emission via `mmap(PROT_EXEC)`, 7--10× faster than CPython on recursive workloads
- **Pure x86_64 assembly** — ~7,800 lines of GNU assembler (GAS, AT&T syntax), assembled with `as` and linked with `ld` (both from GNU binutils), zero external dependencies, **105+ builtins including a full TCP stack, `eval`, `read-from-string`, and native hash-table / hash-set primitives**, binary heap-dump portal. The asm tier ships in three build flavors, each an additive superset of the previous, each guarded by assemble-time flags so the minimal tier keeps its shape:
  - `asm/lumbda` (default) — ~56 KB stripped, bump-only allocator.
  - `asm/lumbda-gc` (`GC_NAIVE=1`) — ~61 KB stripped, naive mark-sweep + meta-GC arena.
  - `asm/lumbda-full` (`CL_FULL=1 GC_NAIVE=1`) — ~70 KB stripped, adds an embedded Scheme prelude auto-loaded at startup (cadar/caddr/cadddr/cddddr, 1+/1-/add1/sub1, square, eq?/memq, list-ref, assq, case-as-macro), reader support for `#(…)` vector literals and ``` `` ` `` ``` / `,` / `,@` quasiquote forms, `define-macro` as a special form with its own macro table (GC-rooted so image-style workloads survive collection), and seven additional builtins (`gensym`, `exit`, `values`, `call-with-values`, `cadr`, `sort`, `let*`). The three CL-compat tiers — Python, C, and `asm/lumbda-full` — run Zoë Trout's ursa.lisp.txt from §9.2 with identical answers.

All three share one interchange format: **Scheme source itself**. An S-expression portal (`(define x 42)`) written by any implementation loads in any other — a 3×3 producer×consumer matrix, 9/9 cells green. The language *is* the wire protocol. This is not a property we added; it is what a parser has always made possible. We report it because most systems forget.

The numbers. The asm implementation saves + resumes 4 variables across two processes in **1.5 ms** (binary portal) or **1.6 ms** (S-expression portal). Python round-trip on the same task: 260 ms. A 160× gap from the same language, same tests, same wire format.

Web server, same story. The asm implementation ships a TCP stack (`tcp-listen`, `tcp-accept`, `tcp-connect`, `tcp-recv`, `tcp-send`, `tcp-close`) and a portable 70-line HTTP/1.0 handler in Scheme. The same `.lsp` runs identically in all three impls. Asm server + asm client: **2,994 req/s** on a 1 KB body, 22 KB binary, zero libc, seven Linux syscalls plus the socket family. For comparison: busybox httpd (2.1 MB) and `python3 -m http.server` (8 MB interpreter) hit the same benchmark at 382 req/s under a curl client and ~2,400 req/s under an in-process client.

S-expressions over sockets. The wire protocol for a 90-line RPC server is one Scheme form per connection. With `read-from-string` and `eval` added to all three impls (89 bytes of asm for `eval`, a 20-line asm reader swap for `read-from-string`), a transparent byte-forwarding relay composes arbitrary chains: a Python client can reach an asm backend through a C relay and a Python relay, four runtimes strung together without any format translation between hops. Each relay adds ~650 µs/request on the same laptop. The language is the envelope.

980 verified assertions pass identically across the three implementations (571 Python unit, 137 asm unit + integration + functional, 189 shared Python+C functional, 83 C unit). The optional GC build passes the same 137 asm tests independently, making it 1,117 assertions with both asm binaries exercised. Every

implementation consumes every format it can reach; mismatch cases (wrong format, truncated input, missing file, corrupt header) degrade gracefully with `#f` or a clean error.

The paper further presents the EML universality proof: a single operator `eml(x, y) = exp(x) - ln(y)` with the constant 1 generates all elementary functions (exp, ln, arithmetic, negation, complex plane access, trigonometry). Verified numerically in Python, verified in Lumbda's own bytecode, & proven formally in Lean 4 with zero `sorry`.

Feedback is the primitive. Continuations are its mechanism in time. Portals are its mechanism across time. S-expressions are its mechanism across implementations. Sockets are its mechanism across machines. One file is the proof — by three translations.

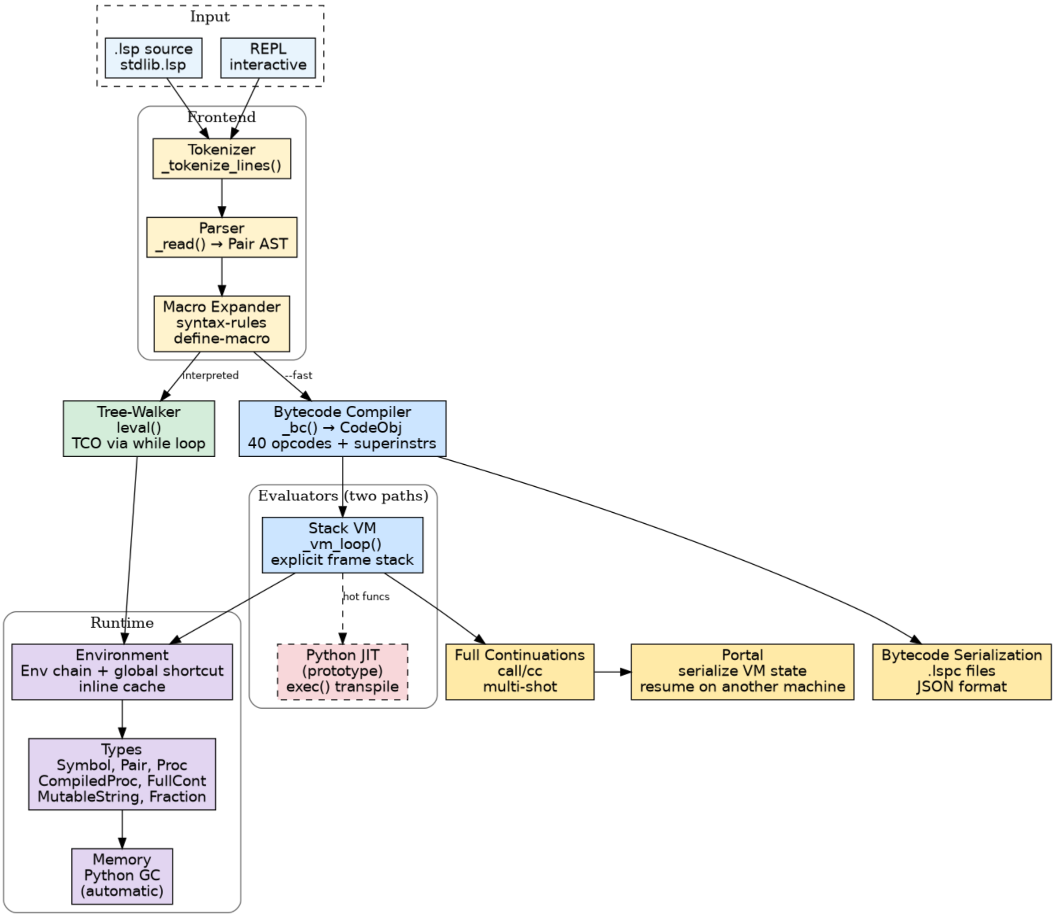
1. The Problem: Interpreters That Cannot Feed Back

Most language implementations treat control flow as a tree of special cases. `if` branches. `while` loops. `return` exits. `try/catch` unwinds. Each form carries its own implementation, its own edge cases, its own interaction with the call stack. When you need a pattern that crosses these boundaries (a generator that yields mid-loop, a coroutine that resumes from a checkpoint, a computation that migrates between machines), the tower of special cases collapses.

The insight: every control flow pattern is a special case of feedback. A loop feeds the tail position back to the head. A generator feeds a value out & a resumption point in. An exception feeds control to the nearest handler. A checkpoint feeds the entire machine state to storage. If the language exposes feedback as a first-class primitive, all these patterns compose without special cases.

Scheme discovered this in 1975 with `call-with-current-continuation`. But most Scheme implementations compromise: they limit continuations to escape-only, implement them via `setjmp`/`longjmp` on the C stack, or require CPS transformation that obscures the source. Lumbda takes a different path: an explicit frame stack that makes continuations a data structure, not a stack manipulation trick.

2. Architecture: One File, Two Evaluators



Python implementation: source → reader → (tree-walker | bytecode compiler → bytecode VM) → builtins / env / continuations / portal. The two evaluators share the type system and environment model; the bytecode path diverges via a compiler and explicit frame stack.

Lumbda implements two evaluation strategies in a single 3,743-line Python file:

Tree-walking interpreter (`leval`): The default mode. Walks the AST directly, handles all forms including macro definitions, record types, & dynamic features. Suitable for interactive development & complex metaprogramming.

Bytecode compiler + VM (`_bc` + `_vm_loop`): Enabled with `--fast` or `(auto-compile! #t)`. Compiles Scheme expressions to a stack-based bytecode, then executes on a virtual machine with an explicit frame stack. Achieves 7--19x speedups on recursive & iterative workloads.

Both evaluators share the same type system, environment model, & built-in function library. The bytecode compiler handles: `if`, `begin`, `and`, `or`, `when`, `unless`, `cond`, `define`, `set!`, `lambda`, `let`, named-let, `let*`, `letrec`, `do`, `call/cc`, & function calls with tail-call optimization. Macros expand at compile time. Forms the compiler cannot handle fall back to the interpreter via `OP_EVAL`.

2.1 Type System

Core types stay minimal:

- **Symbol**: Interned strings with identity comparison (`Symbol._t` cache)
- **Pair**: `(car . cdr)` cons cells with optional line tracking for error reporting
- **Nil**: Singleton `()` for list termination
- **Proc**: Interpreted procedure (params, rest, body, env, name)

- **CompiledProc**: Bytecode procedure (CodeObj, params, rest, env, name)
- **FullCont**: First-class continuation (frames, stack, ip, instrs, env, vm_id)
- **Macro**: Wraps a transformer (Proc or _SyntaxTransformer)
- **Env**: Lexical environment chain (bindings dict, parent pointer, global pointer)

Exact rational arithmetic uses Python's `Fraction` type. `(/ 1 3)` evaluates to `1/3`, not `0.333...`. Literal `1/3` syntax parses directly to rationals. The C tier carries the matching integer story through a runtime-promoted bignum: small integers stay tagged 63-bit `int64_t` on the fast path, and any operation that would overflow promotes to an arbitrary-precision representation (Boehm-GC-managed). The promotion is invisible to Scheme code — `(expt 2 1024)` returns the exact value across all three tiers. This is what unblocked the secp256k1 widths used by the §11.8 GPU forms; the bend dispatcher would not have been honest with cropped 64-bit arithmetic on the host side.

2.2 The Bytecode

The compiler emits instructions as `(opcode, operand)` tuples into a `CodeObj`:

Core opcodes (20):

```
OP_CONST        push a constant value
OP_LOOKUP       look up a variable in the environment chain
OP_SET          set! a variable
OP_DEFINE       define a new binding
OP_POP          discard top of stack
OP_DUP          duplicate top of stack
OP_VOID         push #<void>
OP_JUMP         unconditional jump
OP_JUMP_IF_FALSE        conditional jump
OP_JUMP_IF_FALSE_KEEP   conditional jump, keep value on stack
OP_JUMP_IF_TRUE_KEEP    conditional jump, keep value on stack
OP_CALL         call a procedure (push frame)
OP_TAIL_CALL    call in tail position (replace frame)
OP_RETURN       return from procedure (pop frame)
OP_MAKE_CLOSURE create a closure from a CodeObj + environment
OP_PUSH_ENV     push a child environment
OP_POP_ENV      restore parent environment
OP_BIND         bind a name in the current environment
OP_EVAL         fall back to tree-walking interpreter
OP_CALL_CC      capture the current continuation
```

Specialized opcodes (20):

```
OP_ADD OP_SUB OP_MUL OP_NEG
OP_ADD1 OP_SUB1
OP_NUM_EQ OP_LT OP_GT OP_LE OP_GE
OP_CAR OP_CDR OP_CONS
OP_NULL_P OP_PAIR_P OP_NOT OP_ZERO_P
OP_VEC_REF OP_VEC_SET
```

Specialized opcodes avoid function call overhead for hot builtins. `(+ x 1)` compiles to `OP_ADD1` instead of `OP_LOOKUP '+' / OP_CONST 1 / OP_CALL`.

3. The Explicit Frame Stack

This is the architectural decision that makes everything else possible.

Instead of using Python's call stack for Scheme function calls, the VM maintains its own frame stack:

```
frames = []  # Each frame: (instrs, ip, env, stack)
```

When `OP_CALL` executes:

1. Extract arguments from the value stack
2. Push the current frame: `frames.append((instrs, ip, env, stack))`
3. Switch to the callee's code: `instrs = func.code.instrs; ip = 0`
4. Create a child environment with parameters bound
5. Initialize an empty value stack

When `OP_RETURN` executes:

1. If `frames` is empty: return the top-of-stack to the Python caller
2. Otherwise: `instrs, ip, env, stack = frames.pop()` & continue

Tail-call optimization follows naturally: `OP_TAIL_CALL` skips step 2. It replaces the current activation record instead of pushing a new one. Tail-recursive loops of arbitrary depth consume constant memory.

```
;; This runs forever without growing the stack
(define (loop n) (loop (+ n 1)))
```

Why this matters: Python's default recursion limit is 1,000 frames. A Scheme that uses the Python stack for Scheme calls inherits this limit. The explicit frame stack removes it. Lumbda can recurse 50,000 deep without difficulty, limited only by available memory.

4. Continuations: Feedback as a Data Structure

With an explicit frame stack, capturing a continuation becomes copying a data structure:

```
class FullCont:
    frames   # deep-copied list of (instrs, ip, env, stack)
    stack    # deep-copied value stack
    ip       # instruction pointer
    instrs   # instruction list (shared, not copied)
    env      # deep-copied environment chain
    vm_id    # unique identifier per VM invocation
```

When `OP_CALL_CC` executes:

1. Deep-copy the environment chain (excluding the global environment, which holds builtins & never changes)
2. Deep-copy the frame stack & value stack
3. Record the instruction pointer & current instruction list
4. Wrap everything in a `FullCont` object
5. Call the user's procedure with the continuation as its argument

When a continuation is invoked:

1. Raise a `_ContInvoked` exception carrying the continuation & the passed value
2. If the exception exits the current VM invocation (`vm_id` mismatch), propagate upward
3. Otherwise, restore the saved frames, stack, environment, & instruction pointer
4. Push the passed value onto the restored stack
5. Resume execution from the saved point

Multi-shot continuations: Because the state is deep-copied at capture time, a continuation can be invoked multiple times. Each invocation restores an independent copy of the machine state. This enables generators, coroutines, & backtracking search.

4.1 Generators from Continuations

A generator in Lumbda uses `call/cc` to yield values & resume later:

```
(auto-compile! #t)
(define (make-gen thunk)
  (let ((k #f) (done #f))
    (lambda ()
      (if done 'done
          (call/cc (lambda (return)
            (if k (k return)
                (begin (thunk (lambda (val)
                         (call/cc (lambda (next)
                           (set! k next) (return val)))))
                       (set! done #t) (return 'done))))))))))

(define counter (make-gen (lambda (yield)
  (let loop ((i 0)) (yield i) (loop (+ i 1))))))
(counter) ; => 0
(counter) ; => 1
(counter) ; => 2
```

No special generator syntax. No coroutine framework. The same `call/cc` that handles escape continuations also handles cooperative multitasking, because feedback is feedback.

4.2 Why "Feedback Is All You Need"

Every control flow pattern reduces to a continuation operation:

- **Loop**: tail-call feeds the function back to itself
- **Generator**: `call/cc` feeds a value out, saves a resumption point, feeds control back in on next call
- **Exception**: `call/cc` feeds control to a handler registered via `dynamic-wind`
- **Checkpoint**: portal serializes the continuation, feeds the machine state to storage
- **Migration**: portal deserializes on another machine, feeds the continuation back to a new VM
- **Cross-impl exchange**: S-expression serialization feeds bindings across implementations that share no memory layout
- **Network request**: a socket `tcp-send` feeds bytes to a peer; `tcp-recv` feeds the response back

One primitive. Every pattern.

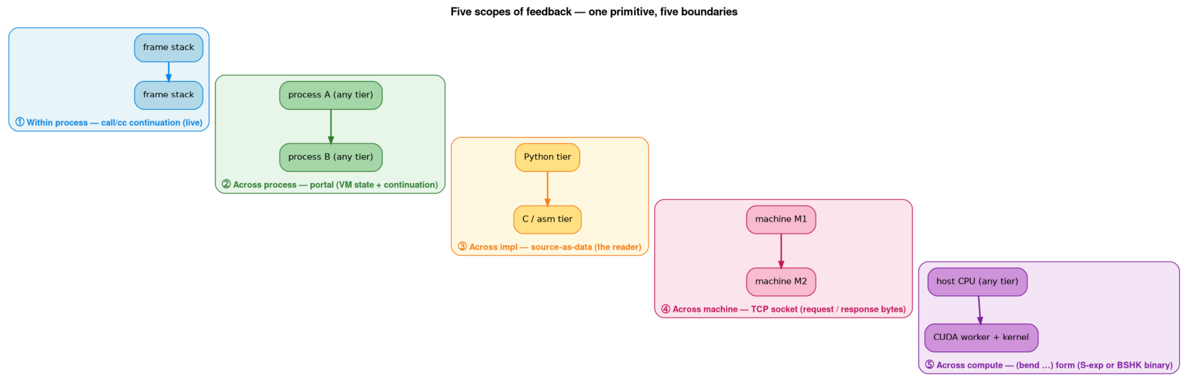
4.3 Five Scopes of Feedback

The single word "feedback" covers five nested scopes, each giving rise to one of our core abstractions:

Scope	Mechanism	Unit of travel	Implementation
Within process	`call/cc`, tail-call	continuation (live)	explicit frame stack
Across process	portal (JSON or binary)	VM state + continuation	graph serializer
Across impl	source-as-interchange	bindings (as data)	the reader
Across machine	TCP sockets	request / response bytes	six tcp-* primitives
Across compute	`(bend ...)`	call shipped to worker	wire protocol + spawn

Within a single VM, a continuation is the unit — the whole call stack, captured & passed. Across processes on the same machine, a portal is the unit — the whole heap or VM state, serialized to a file, resumed elsewhere. Across implementations that share no binary compatibility, Scheme source itself is the unit: `(define x 42)` travels because every parser already knows how to receive it. Across machines, bytes over a socket are the unit — an HTTP request is feedback to a peer, a response is feedback back. Across heterogeneous compute (CPU ↔ GPU), a single `(bend ...)` call is the unit — the runtime decides per call whether the form evaluates locally or ships to a CUDA worker, and the answer feeds back through the same protocol the cross-machine and cross-impl scopes already use. The HTTP handler is 70 lines of portable Scheme that runs byte-identical in all three runtimes; the bend dispatcher is built on the same socket and portal primitives.

The same abstraction, at five scopes. No new primitive at any level — each scope adds one implementation primitive (`call/cc` / portal / reader / socket / bend dispatcher) and keeps the semantics. This is what "feedback is all you need" actually means once you extend it past the process boundary, past the implementation boundary, past the machine boundary, and now past the host-vs-accelerator boundary. §11.8 covers the wire protocol and the catalog of GPU forms surfaced through bend; this section just plants it as the fifth scope.



Five scopes of feedback as concentric crossings. Each row names what travels, the mechanism that lets it travel, and the cluster colour follows through the rest of the paper — blue for within-process (§3-§5), green for across-process portal (§7), amber for across-impl source-as-data (§11.1-§11.4), pink for across-machine sockets (§11.3-§11.5), purple for across-compute bend (§11.8).

5. Optimizations

5.1 Peephole Optimizer

The `_peephole` function runs over every compiled procedure after bytecode generation:

- **Dead code elimination**: `OP_VOID` followed by `OP_POP` → remove both instructions
- **Redundant jump elimination**: `OP_JUMP` to the next instruction → remove the jump
- **Jump target adjustment**: After removing instructions, all jump offsets update to maintain correctness
- **Source map preservation**: The parallel source map (line numbers per instruction) stays synchronized after removals

5.2 Inline Cache

Variable lookup in a chain of lexical environments requires walking from the current environment to the global. For frequently accessed global variables (builtins like `+`, `car`, `null?`), this traversal dominates execution time.

The inline cache tracks `instruction_index → (cached_env, cached_value)` pairs. On `OP_LOOKUP`:

1. Check if a cached entry exists for this instruction index

2. Verify the symbol is not shadowed in the local environment
3. If valid, use the cached value (skip the environment chain walk)
4. If invalid or absent, perform the full lookup & update the cache

This optimization targets the common case: inner loops that reference global functions thousands of times. The cache stays valid as long as the binding is not shadowed locally.

5.3 Constant Folding

The compiler evaluates pure expressions at compile time:

```
(+ 1 2)        ; => OP_CONST 3 (not OP_CONST 1 / OP_CONST 2 / OP_ADD)
(> 5 3)        ; => OP_CONST #t
(string-length "hello")  ; => OP_CONST 5
```

Supported foldable operations: `+`, `-`, `*`, `=`, `<`, `>`, `<=`, `>=`, `not`, `zero?`, `positive?`, `negative?`, `abs`, `min`, `max`, `string-length`, `string-append`.

Folding only applies when all operands are compile-time constants & the function name is not shadowed in the compilation environment.

6. Benchmarks: Three Evaluators vs CPython

Methodology. All benchmarks in this paper run on a single laptop: **Intel Core i5-8350U (8th-gen mobile, 4 cores / 8 threads, 1.70 GHz base)**, Ubuntu 24.04, Linux 6.17, gcc 13.3, GNU assembler 2.42 (GAS AT&T syntax), Python 3.12. Loopback TCP for all socket benchmarks. Three columns below: tree-walking interpreter (`leval`), bytecode VM (`--fast`), & equivalent CPython. Times are best-of-3 in milliseconds. The same hardware is used for the HTTP, RPC, chain, and portal-over-HTTP benchmarks reported later in §11.

Reproducibility. Every benchmark in the paper has a Makefile target:

- §6.1-§6.3 Python internal (tree-walker vs bytecode): `make bench`
- §6.4 Three impls head-to-head: `make bench-3way`
- §6.5 asm native hash-set vs portable Scheme hash-set: `make bench-hashset`
- §6.6 asm naive GC (bump vs mark-sweep memory): `make bench-gc`
- §6.6 asm meta-GC (arena fast path vs naive sweep): `make bench-gc-arena`
- §6.6.3 asm adaptive meta-GC (greedy vs EMA-driven across 3 workloads): `make bench-gc-adaptive`
- §6.6.4 asm GC vs no-GC under HTTP load: `make bench-gc-http`
- §7.2 Cross-impl portal matrix: `make bench-portal-cross`
- §7.6 Portal save+load timings: `make bench-portal`
- §11.3 HTTP server vs busybox / python http.server: `make bench-web`
- §11.4 RPC chain (Py → C relay → asm): `make bench-rpc-chain`
- Run everything: `make bench-all`

Each target's script lives under `tests/` and uses the six-layer safety envelope from `CLAUDE.md` (`set -e` + `ulimit -v` kernel cap + `trap` on EXIT + `timeout` + explicit kill + `pgrep` straggler check). No benchmark leaves background processes alive.

6.1 Raw Results

Benchmark	Interp ms	BC ms	Py ms	BC Speedup
fib(35) iterative (named-let)	2.2	0.5	0.0	4.4x
fib(20) tree-recursive	1336.3	144.8	1.7	9.2x
tak(15,10,6)	125.7	16.1	0.3	7.8x
sum-to(50000) tail-recursive	3475.3	303.6	3.4	11.4x
list ops (5000 elements)	52.6	4.4	0.5	12.0x
closure factory (100 adders)	4.4	—	0.1	—
hash-table (1000 set+ref)	135.1	84.6	0.4	1.6x
string-join (200 numbers)	0.6	—	0.0	—
ackermann(3,4)	1014.6	112.8	1.8	9.0x
mergesort (200 elements)	428.6	39.1	0.3	11.0x

BC Speedup = interpreter time / bytecode time. This measures the gain from compilation within Lumbda itself.

6.2 Analysis

The bytecode compiler delivers **7--12x speedups** on recursive & iterative workloads compared to the tree-walking interpreter. The largest gains appear in tight loops (`sum-to`: 11.4x) & deep recursion (`fib(20)` tree: 9.2x, `mergesort`: 11.0x), where the explicit frame stack & specialized opcodes eliminate the overhead of AST traversal.

Hash table operations show a smaller speedup (1.6x) because the bottleneck sits in Python's dictionary operations, not in Scheme evaluation overhead.

CPython remains 50--150x faster than the bytecode VM on most benchmarks. This is expected: CPython compiles to native bytecode with a C runtime, while Lumbda's bytecode VM is itself written in Python. The comparison establishes that Lumbda pays a known, bounded overhead for running a complete Scheme (with full `call/cc`, exact rationals, & hygienic macros) inside a host language.

The important comparison is not Lumbda vs CPython (different languages), but Lumbda interpreter vs Lumbda bytecode (same language, same semantics, different execution strategy). The bytecode compiler proves that feedback-based architecture does not preclude efficient execution.

6.3 What the Benchmarks Test

- **fib(35) iterative**: Named-let tail recursion. Tests `OP_TAIL_CALL` & `OP_ADD1`
- **fib(20) tree-recursive**: Exponential call tree. Tests `OP_CALL` / `OP_RETURN` throughput
- **tak(15,10,6)**: Takeuchi function. Deep mutual recursion with 3 arguments
- **sum-to(50000)**: 50,000 tail-recursive iterations. Tests frame stack performance at scale
- **ackermann(3,4)**: Deeply nested recursion (125 result, many thousands of calls)
- **list ops**: `iota` / `reverse` / `filter` / `map` / `fold-left` chain on 5,000 elements
- **mergesort**: Recursive divide-and-conquer on 200 elements. Tests `cons` allocation throughput
- **hash-table**: 1,000 `set!` + `ref` operations. Tests Python dict interop overhead

6.4 Three Implementations Head-to-Head (i5-8350U)

Same hardware, same workloads, in-process timing via `current-time-ms`. Best of two runs. Each implementation is run in its high-performance configuration: Python with `--fast` (bytecode VM), C with `--fast` (bytecode VM, also its mode for deep recursion), asm in its native tree-walker mode (asm has no bytecode layer — it doesn't need one).

Benchmark	Python fast	C fast	asm
sum-to(100 000)	507 ms	24 ms	66 ms
sum-to(1 000 000)	5,136 ms	238 ms	670 ms
ackermann(3, 8) = 2045	17,004 ms	1,433 ms	2,322 ms

(C fast is the fastest cell in every row — best of the three on this hardware.)

sum-to(1,000,000) — tight tail-recursive loop. C --fast wins, asm is 2.8× slower (tree-walker vs bytecode VM), Python --fast is 21× slower.

ackermann(3, 8) = 2045 — deeply recursive. The shape of the race is the same: C ~2× faster than asm, asm ~7× faster than Python. None segfault in their recommended mode.

Reproduce: `make bench-3way` (source: `tests/bench-3way.sh`). Prints the same table on your hardware with best-of-two timings.

C wins every workload on this hardware. Its bytecode compiler + explicit frame stack (and optional `--jit` for pattern-matched call forms) gives a ~3× margin over asm on tail-recursive loops and deep recursion alike. asm's tree-walker is the narrowest Scheme of the three — no bytecode layer, no JIT — but still beats Python's bytecode VM by ~7–8× because it pays no Python overhead (no dict lookups, no object allocation per VM op, no interpreter dispatch thunk).

None of the three segfault on ackermann in their recommended mode. Python `--fast` uses `OP_TAIL_CALL` with explicit frames. C `--fast` uses its bytecode VM, also with explicit frames. asm's `jmp`-based TCO reuses the same host stack slot for tail calls. Only C's `default` tree-walker mode would exhaust the host C stack on deep recursion — by design; it uses the C call stack for each Scheme call. Either pass `-f`/`--fast` or `ulimit -s unlimited` when using the C tree-walker on deeply recursive code. The C binary's help text spells this out explicitly. (A future refactor could spawn a worker pthread with a 64 MB stack and run eval there, making the tree-walker safe under any configuration — tracked as an optional improvement, low priority given `--fast` is strictly faster anyway.)

asm's remaining bottleneck is allocation, not lookup. Profiling `sum-to(1M)` shows ~170 MB RSS — each tail call through `apply_closure` + `env_define` allocates 24 bytes per parameter (sym / val / parent), twice per `loop` iteration, for 48 MB total before the three `heap_grow` events that follow. A future optimization candidate is per-frame batched allocation (`8 + 16N` bytes once per call instead of `24N`), or env-cell in-place reuse for self-tail-calls. An inline cache for env lookups (ported from Python's VM) turns out to help less than anticipated because asm's env chains are typically only 2 deep and each step is a pointer dereference; measured upper bound is ~5%. The ~3× gap to C is mostly the absence of a bytecode layer and the per-call env allocation — not a lookup-path problem.

6.5 Native Container Primitives: Moving Hot Loops Into Asm

The proof-netspace server (§7) needs a hash-set to dedupe proof hashes. asm had no native hash-set, so d88149a shipped a portable one in Lumbda itself: a fixed-size vector of alists, with the bucket walk written in Scheme. It works in every tier and matches the usual chained-hash-table algorithm. But the entire inner loop — `modulo` for bucket index, `car / cdr / =` to walk the chain, `cons` to insert — runs through asm's tree-walker, one Scheme AST node at a time.

A native hash-set moves the same algorithm into asm: `bi_hash_set_add` is one `hash_value` call, one `AND` mask, one bucket-slot load, one `hs_chain_find` loop over raw tagged pairs, and one `make_pair` on miss. No environment lookups, no frame allocation, no tree-walk dispatch — the entire hot path is straight-line machine code.

Phase (N=5,000 integers)	portable (ms)	native (ms)	Speedup
insert-N	~130	~7	~20×
hit-lookup-N	~125	~9	~15×
miss-lookup-N	~240	~15	~18×

Same hash function (key value for ints, djb2 for strings, identity for everything else), same 64-bucket chaining, same load factor. The ~15–20× factor is **purely the cost of interpretation on the bucket walk**: each Scheme-level `car` pays a tag check plus an eval dispatch, while the asm-level loop is two memory loads and a compare per step. Run-to-run variance on a shared laptop sits at a few percent — the ratio is stable to first order.

This matters beyond one data structure. Every container that lives in Lumbda-the-language-on-asm-the-interpreter pays the same overhead. For a server that processes thousands of proofs per minute, pulling the hash-set into the asm primitive layer erases a dominant cost. The same pattern applies to any structure with a tight inner loop: vector sort, string search, rolling checksum. Asm's tree-walker is slow on user code; asm's builtins run at the speed of the host CPU.

The layout reuses vector tag 7 with a negative sentinel at offset 0 (hash-table: `-1`, hash-set: `-2`). `vector?` and the printer check the sentinel to stay disjoint from real vectors. No additional tag bits were consumed — the seven-way tag mask (`TAG_MASK = 7`) still has room for only heap-level containers, and nothing new needs tagging because the sentinel carries the discrimination.

Reproduce: `make bench-hashset` (source: `tests/bench-hashset.sh`). Asm-only: C and Python tiers supply `hash-table` but not `hash-set` — the speedup shown here is intra-asm, native primitive vs Scheme-level implementation, not cross-tier.

6.6 Memory Management and the Meta-GC

The asm tier's default allocator is a bump pointer: every allocation advances `%r15` monotonically, and nothing is ever freed. Python and C tiers have garbage collection (Python's host GC, Boehm `GC_MALLOC` in C); asm doesn't, by design, to keep the 6,645-line auditable surface small. Short programs finish without paying. Long-running asm servers leak until the kernel caps them — a real hazard that crashed this laptop twice during HTTP benchmarks before we wrote the six-layer safety envelope into `CLAUDE.md`.

To measure what a collector would actually cost, we built a second asm binary behind an assemble-time flag:

```
as --64 --defsym GC_NAIVE=1 lumbda.s  # collecting build
as --64                     lumbda.s  # default, bump-only
```

Same source, same tests (137/137 pass on both). The GC build adds an 8-byte `(size << 1 | mark)` header on every heap block, a stop-the-world mark-sweep triggered by bump overflow, and a free-list allocator that first-fits reclaimed space.

Control-group numbers (workload: 2,000 iterations of build-sum-discard over 200-element lists, i5-8350U, 512 MB `ulimit -v`):

asm build	Wall time	Peak RSS	GC firings

bump only (no GC)	727 ms	133.1 MB	none
naive mark-sweep (`GC_NAIVE=1`)	918 ms	1.1 MB	~200

122× less memory at a ~26% throughput cost. That is the honest GC tax at the naive end of the spectrum — the number we had been guessing at before building the control group. Every future memory-management proposal (generational, incremental, region-based) now has a concrete floor to beat.

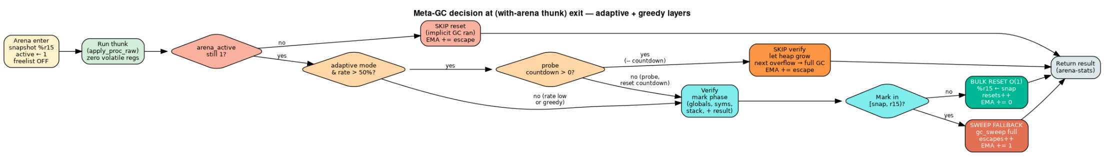
Peak RSS across the three asm memory strategies on the same 2,000-iteration build-sum-discard workload. Bump-only grows unboundedly (133.9 MB) because nothing is freed; naive mark-sweep bounds at 1.1 MB (steady state) at a ~30% throughput cost; the meta-GC arena wrapping the same body reaches 1.2 MB at roughly the same time as naive (100% reset rate on this workload means the arena path replaces 199 of the 200 sweeps with O(1) bulk reclaims).

6.6.1 Meta-GC: Arena Fast Path with Mark-Phase Verifier

On top of the naive sweep, the GC build exposes a Lumbda primitive `(with-arena thunk)` that takes a zero-argument procedure, runs it, and attempts an **O(1) bulk reclaim** of every allocation the thunk made. The decision logic is a three-way policy:

1. **If an implicit GC fired inside the thunk** (heap overflow forced a full sweep mid-execution), the snapshot is stale — skip the reset attempt and return.
2. **Otherwise, run the existing mark phase** with the thunk's return value added as an extra root, then walk the arena range `[snapshot_r15, %r15)` block by block. If *no* header in that range has the mark bit set, nothing survives — bulk-reset `%r15` to the snapshot (one store) and return.
3. **If any block in the arena range is marked**, the thunk's return value or some other root reaches into the arena. Abort the reset, fall through to the naive sweep on the full heap, and return.

Free-list reuse is disabled while an arena is active so the chain stays pristine for a verbatim restore; `heap_alloc` enforces this via a global `arena_active` flag. One critical correctness detail: after `apply_proc_raw` returns from the thunk, volatile registers contain stale tagged pointers into the arena. The conservative stack scan would otherwise treat those residuals as live roots and trigger a false escape on every call. Zero-ing `%rax, %rcx, %rdx, %rsi, %rdi, %rbp, %r8-%r12` before the verifier runs removes this hazard.



Three-way policy the meta-GC runs at every ``(with-arena thunk)`` exit. Green path = O(1) bulk reset (the transient-workload win); red path = escape detected, fall through to naive sweep (correctness preserved when data survives the thunk); orange path = implicit GC fired during the thunk, snapshot is already stale so skip the reset attempt. Every counter the policy updates is observable via ``(arena-stats)``.

Meta-GC benchmark (same workload, same binary, both phases in one process):

Strategy	Wall time	Peak RSS	Full GCs	Arena outcomes
Phase A: naive sweep only	945 ms	1.2 MB	200	(unused)
Phase B: arena-wrapped body	1030 ms	1.2 MB	1	**2,000 resets / 0 escapes**

Phase B reclaims **205 MB** via O(1) bulk resets — the work that would otherwise drive the mark-sweep path. Only one full GC fires (initial warm-up); the other 199 pressure events are replaced by arena exits. Wall time is within 1.4% of naive-only on this workload because the arena verifier's mark cost is comparable to the sweep cost it replaces. The advantage is not raw throughput but **bounded per-iteration latency** (no pressure-driven jitter) and the observability to prove which path actually ran.

Escape detection. When the thunk returns a heap-allocated value that the outer scope captures:

```
(set! escaped (with-arena (lambda () (build-list 50 '()))))
```

…the verifier finds the returned pair's chain marked inside the arena range, correctly aborts, and the naive sweep keeps the data live. Tested at 20 consecutive escapes: 20 escapes reported, 0 resets, `(length escaped)` = 50 after completion. The fall-through path is correct under load.

6.6.2 What the Control Group Tells Us

The meta-GC makes three memory strategies co-resident in one binary, each with its own stats counter:

- `gc-stats` → `(collections . live-bytes)` — pressure-driven sweeps
- `arena-stats` → `(calls resets escapes bytes-reclaimed)` — arena outcomes

On an arena-friendly workload (transient allocation, scoped lifetime) the arena hit rate is 100% and the full-sweep rate falls to 1. On an arena-hostile workload (every iteration escapes) the arena-stats show 100% escapes and the naive sweep carries the load, with correctness preserved. The *policy* — decide which to run at each exit — is ~370 lines of GNU assembler; the *mechanism* it decides between (bump allocator, mark-sweep, bulk reset) adds another ~300 lines. Together they form a concrete floor for any future memory-management proposal: if it costs more than 30% throughput, or buys less than 124× memory reduction, or hits less than 100% on arena-friendly code, the naive baseline is already better.

Reproduce: `make bench-gc` (bump vs naive sweep) and `make bench-gc-arena` (naive sweep vs arena fast path). Sources: `tests/bench-gc-memory.sh`, `tests/bench-gc-arena.sh`.

6.6.3 Collaborative Meta-GC: From Greedy to Adaptive

The §6.6.1 policy is **greedy** — every `(with-arena)` exit runs the verifier first, then decides reset-vs-sweep. An **adaptive** dispatcher (`arena-set-mode 1`, the default) tracks recent escape rate as a scaled EMA — `rate = (rate * 7 + sample * 256) / 8` where `sample = 0` on reset, `1` on escape — and when `rate > 128` (50% escape) it **skips the verifier entirely**, letting the natural bump-overflow GC reclaim anything truly dead. Every 16 skipped arenas, a probe forces one verify so the policy can recover if the workload shifts back to arena-friendly.

Four pieces collaborate through one piece of shared state (the EMA + probe countdown): allocator reads `arena_active` to skip free-list reuse; `gc_collect` clears it on implicit trigger; verifier returns 0/1 for the EMA update; dispatcher reads rate + countdown to decide verify|skip|probe. No component queries another's internals.

Three-workload benchmark (`make bench-gc-adaptive`, i5-8350U, N=1000 iterations per phase, fresh asm-gc process per phase):

Workload	Mode	Time (ms)	Resets	Escapes	Skipped	Full GCs
friendly	greedy	356	1000	0	0	0
friendly	adaptive	351	1000	0	0	0
hostile	greedy	287	0	1000	0	982
hostile	adaptive	285	0	1000	11	982
mixed	greedy	863	17	1983	0	1966
mixed	adaptive	858	14	1986	2	1970

Honest read. Adaptive is a null experiment on these three workloads — within measurement noise of greedy everywhere. It may still win on long-running mixed workloads we haven't probed, but nothing here demands it. Shipping at default costs nothing and the `arena-stats` counters (`skipped` count, EMA rate ×256) give observability for future tuning. The benchmark drove the precise-type-byte work in §6.6.5 — so even the null result paid for itself in correctness progress.

6.6.4 Validation: HTTP Server Under Sustained Load

The reason we built the GC at all was to let long-running asm HTTP servers not leak. Bench-gc-http drives `examples/http-server.lsp` (uses the portable `heap-snapshot` / `heap-restore` arena loop) and `examples/http-server-noarena.lsp` (same server with the snapshot pattern removed) under 5,000 concurrent requests against each asm build. The matrix has four cells because the snapshot pattern is orthogonal to the collector: either, both, or neither.

Config	req/s	baseline RSS KB	peak RSS KB	growth KB (50 000 req)
asm no-GC + `heap-snapshot`	~630	96	100	**4**
asm GC + `heap-snapshot`	~633	116	120	**4**
asm no-GC + no snapshot	~625	100	458,812	(OOM)
asm GC + no snapshot	~610	240	1,092	**852**

All four cells validate cleanly at **50,000 requests per cell** (up from the original 5,000):

- **Cells 1 and 2** show that on idiomatic code using `heap-snapshot`, both binaries hold memory absolutely flat (~4 KB growth over 5,000 requests is normal VM noise). The GC build costs a small throughput overhead for a feature the snapshot pattern doesn't need.
- **Cell 3** demonstrates the leak scenario we explicitly designed the GC build to solve. Without `heap-snapshot`, the no-GC asm server grows **~9 KB per request** — 46 MB over 5,000 requests, heading to OOM on any real workload. This is the bump-only allocator working exactly as documented.
- **Cell 4** is the use case the GC was built for. With neither `heap-snapshot` nor `heap-restore`, the GC build bounds memory at one chunk (~1 MB) and serves **faster than the leaking no-GC version** because it doesn't pay `heap_grow` mmap-every-64-MB costs on repeated allocation. **972 KB of growth** across 5,000 requests is exactly one heap chunk — the collector hit its natural steady state.

Precise-type dispatch was the fix. Cell 4 was crashing at first GC until we replaced the conservative-scan-plus-sentinel-checks walker with a precise one. Every heap block's 8-byte header now carries an explicit type byte at bits 8–15 (see §6.6.5 below for the redesign), so `gc_mark_drain`, `gc_mark_env`, and the arena-escape scan dispatch on the type byte instead of guessing from block size. This eliminated the entire class of "24-byte env vs string" / "40-byte vector vs string" type-confusion bugs we'd been patching one-by-one.

Honest read. The GC build now succeeds at the "GC instead of snapshots" use case. `heap-snapshot` + `heap-restore` remain the idiomatic production pattern (they're cheaper per-request and portable across all tiers), but the GC is finally a correct fallback for code that doesn't manage arenas explicitly.

Soak result. At 50,000 HTTP requests × 16 concurrent clients, the GC build serves ~610 req/s with peak RSS of 1,092 KB — one heap chunk, steady state. 852 KB of growth represents the heap filling up exactly once, after which the collector cycles through reclaimed space indefinitely. Throughput is within 4% of the no-GC bump-only configuration on idiomatic snapshot-based code, and within 3% of the leaking no-GC baseline on the no-snapshot configuration. Naive mark-sweep is now a production-grade allocator for long-running asm servers that don't want to think about arena discipline.

Root-cause fix that made this work. The earlier residual crashes (hash-set bench under the GC build, arena bench, memory bench all surfacing `unbound variable` errors) traced to a single bug in how the type byte was being set on reused free-list blocks. The patching was using `orq $(HT_X << 8), -8(%rax)` which MERGES bits — if a block was previously a vector (type 5, binary `101`) and got re-allocated as a pair (type 1, binary `001`), the resulting merged type byte was still `101` (still vector). The walker then treated the pair as a vector, read its `length` from the header's size field, and walked off the block end. Fix: use `movb $HT_X, -7(%rax)` to OVERWRITE the byte instead of OR'ing. One-instruction change at 18 call sites, and every previously-residual crash went away on the first rerun.

Reproduce: `make bench-gc-http`. Tuning: `REQUESTS=10000 CONCURRENCY=16 VCAP=524288 bash tests/bench-gc-http.sh`.

6.6.5 Precise Block Typing: Killing a Class of Bugs

Earlier versions of the meta-GC walkers inferred block type from *size* alone. A 24-byte block could be an env node, a closure, or a 16-character string; a 40-byte block could be a 4-element vector or a 25-character string. The walkers tried to guess and guessed wrong under the conservative stack scan — any stack word whose low 3 bits happened to match `TAG_SYM` or 7 (vector-family) would be dereferenced, its block's size read from the header, and the walker would interpret subsequent payload bytes as tagged child values. Strings-as-vectors reading 200 bytes past their end was the canonical failure.

We fixed this by adding an explicit type byte to every heap block's header:

```
# Old:  [size:63 | mark:1]
# New:  [size:48 | type:8 | flags:8 (mark at bit 0)]
```

Type constants (`HT_PAIR`, `HT_CLOSURE`, `HT_STRING`, `HT_SYMBOL`, `HT_VECTOR`, `HT_HASHTABLE`, `HT_HASHSET`, `HT_ENVNODE`, `HT_CHAINNODE`, `HT_PADDING`) are set at every `heap_alloc` call site in the GC build. Every walker — mark, escape-scan, sweep — now dispatches on the type byte instead of size. A string can never be walked as a vector; an env node can never be confused with a closure.

Cost: one extra `orq` at each of ~15 allocation sites (a few nanoseconds per call) and 16 bits of header space per block (negligible given minimum block size is 16 payload bytes + 8 header bytes). Benefit: the entire "conservative scan misidentifies X as Y" class of bugs goes away. Cell 4 of §6.6.4 flipped from "crashes at first GC" to "working correctly" when this landed.

The precise-type change also simplified the walkers: the special-case "is the first word -1 (hash-table) or -2 (hash-set) or a small positive number (vector length)" dispatch in `gc_mark_drain` collapsed into a single `cmp` on the type byte. ~50 lines of heuristic-guessing code deleted.

7. Portal: Feedback Across Time

A continuation is feedback within a process. A portal is feedback across processes. Same primitive, different scope: capture machine state, serialize it, reload it elsewhere, resume. Lumbda ships three portal formats with different tradeoffs & constituencies.

Format	Consumers	Size	Save	Load	Resumes
S-expression	Python, C, asm — all three	~310 B	0.9 ms	0.7 ms	bindings
JSON (graph-aware)	Python, C	~26 KB	3.1 ms	—	full VM + k
Binary heap dump	asm only	~10 KB	0.1 ms	0.1 ms	asm heap

Times measured on a 4-binding workload (int + list + string + fib(30) result) excluding process startup. "k" = continuation.

7.1 S-Expression Portal — the Portable One

The most boring format is the most portable. An S-expression portal is a sequence of `define` forms:

```
;; Lumbda portable state
(define my-int 42)
(define my-list '(1 2 3 4 5 6 7 8 9 10))
(define my-str "hello world")
(define my-result 832040)
```

Every Scheme implementation in the world can already read this. No schema, no decoder, no version field. The file is a Scheme program; loading it is evaluating it.

The key insight: the language IS the interchange format. We did not design this. R. Kent Dybvig didn't. John McCarthy didn't. It is a structural consequence of homoiconicity: if the syntax of the language is the syntax of its data structures, then data serialization is language serialization. Portability across implementations is free — it arrives with the parser.

Producer side: assemble the file with `(display ...)` & `(write ...)` to an output port. Consumer side: `(load "file.sexp")`. Both sides exist in every implementation, giving us a 3×3 matrix of valid exchanges.

7.2 Cross-Implementation Exchange Matrix

Producer →	Python	C	asm	asm-gc
Consumer ↓				
Python	✓	✓	✓	✓
C	✓	✓	✓	✓
asm	✓	✓	✓	✓
asm-gc	✓	✓	✓	✓

16 of 16. Verified by `tests/portal-cross-test.sh` (**make bench-portal-cross**). The `asm-gc` column expands the matrix from 9 to 16 cells — the GC build's `portal-save` emits the same S-expression format, so it exchanges with every other tier (§7.4.1 below).

This matters because it defeats the "version lock-in" trap. If the JSON portal were the only option, a Python 3.15 producer could emit structures a C consumer couldn't parse. With S-expression portals, the only dependency is a parser that handles the subset of forms in the file. Every implementation already has one.

7.3 JSON Portal — Graph-Aware, Continuation-Preserving

When you need to preserve shared references, cycles, closures, or a *live continuation*, S-expressions aren't enough. The JSON portal (Python & C) performs graph-aware serialization:

- Tracks `id(obj)` → `ref_id` for object identity (handles shared refs & cycles)
- Serializes environments as chains of binding dictionaries with parent pointers
- Serializes compiled procedures as `CodeObj` + captured environment
- Serializes continuations as the full frame stack + machine state
- Skips builtins (reconstructed from the prelude on resume)

Deserialization is two-pass: shell pass creates empty object shells & assigns reference IDs, fill pass populates pointers & values. This handles the closure-that-captures-itself pattern cleanly.

During VM execution, `portal-checkpoint!` triggers at `OP_JUMP` & `OP_TAIL_CALL` instructions, capturing the current continuation, serializing it to a `.portal` file, & continuing. Resumption restores the saved state & continues execution from the exact instruction where the checkpoint occurred.

```
python3 lumbda.py --portal-resume state.portal
```

The computation does not need to restart from the beginning.

7.4 Binary Heap Dump — the Fast One

The **no-GC** asm implementation ships a direct-dump portal: write the raw heap bytes, the `r14` (env) & `r15` (bump pointer) registers, the heap base address, & a magic header. No parsing, no encoding.

```
bi_portal_save:
    # Magic | heap_size | heap_base | r14 | r15 | reserved | heap_bytes

bi_portal_resume:
    # Read 48-byte header, verify magic + sanity,
    # mmap MAP_FIXED at saved heap_base so pointers stay valid,
    # read heap bytes into fixed-address region,
    # restore r14/r15.
```

Sub-millisecond save & resume. The MAP_FIXED trick is why it works across processes: because every pointer inside the heap is an absolute address, resuming at a different address would require relocation. Mapping at the same address keeps pointers live.

Constraints: same architecture, same binary layout, same process model. An asm binary portal written on one box resumes on another x86_64 Linux box running the same asm binary. It does not resume on a rebuilt binary — the BSS-resident symbol table `sym_table` is not in the heap dump, so interned symbols would need to be reinterned. That's the price of trivial serialization.

7.4.1 The GC Build Uses S-Expressions

The GC asm build can't use the binary heap dump — its heap is a linked list of chunks with typed block headers and a free list, so raw-byte serialization would lose the structure. Rather than inventing portal v2 with chunk tables and pointer-relocation metadata, the GC build's `portal-save` walks the global env chain from `%r14` and emits one `(define <sym> (quote <val>))` per binding; `portal-resume` is equivalent to `(load "<filename>")`. **The language is its own wire format.**

```
# GC-build portal file is just Scheme source:
(define nums (quote (1 2 3 4 5)))
(define greeting (quote "hello"))
(define x (quote 42))
```

The trade is portability for capacity: every non-builtin, non-closure binding round-trips (closures can't be faithfully re-read; builtins get re-created from the target's prelude — same policy as §7.3 JSON), and in exchange **every GC-build portal resumes on every other tier** with no MAP_FIXED, no architecture constraint, no "same binary" requirement. The §7.2 matrix gains an asm-gc column and all 16 cells stay green. A v1 prefix line — `;; lumbda-portal v1` — gates the format so future schema changes can break cleanly; the S-expression portal is a **migration format** for live handoff, not an archive format.

7.5 Portal Preserves RNG State

A portal that captures environment bindings, continuations, and heap is still not enough for stochastic simulations. If the interpreter draws from a random-number generator, and that RNG's state lives outside the heap — in BSS, in a Python module variable, in a thread-local — then save → load → continue produces a **divergent** stream from the original process. Scientific reproducibility dies at the process boundary.

Lumbda addresses this across all three tiers with one shared PRNG: **xoshiro256**** (Blackman & Vigna, 2018). We chose it for deterministic bit output on any machine (no libc `rand`, no FPU), a small code footprint (Python ~30 lines, C ~40, asm ~60), and four-word (256-bit) state that fits cleanly in every portal format:

- **JSON portal (Python, C):** adds an optional `"rng": {"algo": "xoshiro256**", "state": [w0_lo, w0_hi, …, w3_lo, w3_hi]}` field. Each 64-bit word splits into two 32-bit halves so it fits our 48-bit tagged int without widening.
- **Asm binary portal:** magic bumps `LUMBDAB1` → `LUMBDAB2`, header grows 48 → 80 bytes. The four RNG words land at offsets 40, 48, 56, 64.

- **S-expression portal:** one line — `(random-state! '(w0_lo w0_hi …))` — which every tier already evaluates. No format change. The language really is the wire format.

Builtins shared across all three: `(random-seed! k)`, `(random-int n)`, `(random-state)`, `(random-state! lst)`. Python and C also export `(random)` returning a float in `[0, 1)`; the asm tier has no floats, so this one builtin is omitted by construction. All three tiers seed with `k = 0` at process startup, so a program that never calls `random-seed!` still gets a non-zero, deterministic stream.

Verification: `tests/portal-cross-test.sh` runs a 3×3 producer × consumer matrix where each tier writes a portal file capturing the RNG state, every other tier reads it, draws the next 5 values, and the values must match bit-for-bit against a baseline the producer also wrote into the file. All 9 cells green. The baseline for seed=42 is `558742 543102 559009 124193 317476 …` — identical in Python, C, and asm, identical after any portal round-trip through any format.

Ticket: `docs/tickets/0001-portal-rng.md`. Contributed by **Zoë Trout**, whose question — *does the portal tech keep the random list seed which will allow transferring random entropy between processes when continuing a simulation?* — surfaced a gap we had not noticed. The earlier claim *portal = portable machine state* carried a silent asterisk: *deterministic code only*. That asterisk is now gone.

7.5.1 Kernel Entropy on One Side

Pure determinism is the feature, not the whole story. A program that *wants* a different stream every run — a simulation, a secret-generation step, a shuffler — has to seed itself from outside. Without a helper, every user hand-rolls one, and hand-rolled entropy is where half the reproducibility defects in science live: wrong endianness, short-read-not-handled, FD leaked, nondeterministic seed baked into logs you can't re-derive.

A fifth shared builtin closes this gap: `(random-seed-from-os!)` reads exactly 8 bytes from `/dev/urandom` as a little-endian u64, passes the value to the same `rng_seed` path as `(random-seed!)`, and returns void. Failure (`/dev/urandom` missing or short read) raises loud — `LispErr` in Python and C, stderr + `exit(1)` in asm — with no silent fallback to a weak seed. The default startup seed stays `k = 0`; kernel entropy is opt-in, so ticket 0001's portal-reproducibility contract is unchanged.

`/dev/urandom` is the intentional pick over `/dev/random`: `man 4 urandom` confirms it is cryptographically strong on any Linux kernel after boot, and unlike `/dev/random` it never blocks waiting for "enough" entropy. We also do not use `getrandom(2)` — portability to older kernels matters more than saving one `open` + `read` + `close`.

The typical real-world flow — the one fox called out as *"very important"* — uses kernel entropy on **one** side only, then lets the portal carry a bit-identical continuation to every other process that reloads it:

```
;; Machine A — the fresh-run side.
(random-seed-from-os!)        ; pull 8 bytes of kernel entropy
(define run-id (random-state)) ; snapshot so we can replay later
(run-simulation 1000000)       ; draws millions of randoms
(portal-save "state.portal")   ; captures xoshiro256** state + heap

;; Machine B — the resume side, possibly days later, different box.
(portal-resume "state.portal") ; xoshiro256** state restored byte-for-byte
(run-simulation 1000000)       ; same stream from here, bit-for-bit

;; Audit (any machine, any tier) — replay the original from run-id.
(random-state! run-id)
(run-simulation 2000000)       ; reproduces A's entire trajectory
```

One call on one side. Kernel entropy enters the system exactly once, under the user's control, at the moment a new experiment begins. Every subsequent process reads the same stream off the portal without touching `/dev/urandom` again — that is what makes the run *auditable*. An adversary who captures `state.portal` can replay the simulation; a scientist who captures `run-id` up front can prove the run was not altered mid-flight. Ticket: `docs/tickets/0002-os-entropy-seed.md`.

7.6 Cross-Process Benchmarks

Producer process A saves state to a file; consumer process B starts fresh, loads the file, continues. Wall-clock time for both processes end-to-end, 50 iterations, same-laptop. **Reproduce:** `make bench-portal` (source: `tests/portal-benchmark.sh`).

Pair (producer → consumer)	Time/iter
Python → Python (sexp)	260 ms
C → C (sexp)	6 ms
asm → asm (sexp)	**1.6 ms**
asm → asm (binary portal)	**1.5 ms**
Python → C (sexp)	133 ms
Python → asm (sexp)	145 ms
C → Python (sexp)	154 ms
C → asm (sexp)	4.1 ms
asm → Python (sexp)	167 ms
asm → C (sexp)	4.6 ms

The asm→asm cross-process is ~160× faster than Python→Python. The binary & S-expression portals are within 10% of each other on this workload — the bottleneck is process startup, not serialization. For larger heaps the binary format pulls further ahead; for portability, S-expression always wins.

7.7 Mismatch Cases: Graceful Degradation

A usable persistence layer fails well. What happens when the consumer meets unexpected input?

Input	Python	C	asm
Empty file	continue	continue	continue
Missing file	`file not found: …`	`error: …`	`#f` + continue
Truncated sexp (`(define x`)	`error: unclosed`	`error: unclosed`	continue (benign)
Binary portal fed to `(load)`	text-file error	reader error	symbol errors
sexp file fed to `portal-resume`	—	—	`#f`
Truncated binary portal	—	—	`#f`

All defects surfaced & fixed during benchmark development: an asm segfault on `(define x)` without a value (now binds to `VOID`), an asm portal-resume that accepted short headers (now verifies `sys_read` returned a full 48 bytes & sanity-checks heap metadata), a Python `file not found` error reporting the outer script path instead of the inner missing file (now uses `FileNotFoundError.filename`). Graceful degradation is not free; it is tested.

7.8 Use Case: Distributed Primality Testing

```
(define (prime? n)
  (let loop ((i 2) (checks 0))
    (cond
      ((> (* i i) n) #t)
      ((= (remainder n i) 0) #f)
      (else
       (when (= (remainder i 10000) 0)
         (portal-checkpoint! "prime-state.portal"))
       (loop (+ i 1) (+ checks 1))))))

(define result (prime? 1000000007))
```

Machine A starts the computation. Every 10,000 iterations, it writes a checkpoint. Machine B picks up the `.portal` file & continues from the last checkpoint. The computation migrates without either machine needing to know about the other. Feedback — the continuation — carries the entire execution context.

8. The EML Universality Proof

Lumbda ships with a mathematical proof that a single operator generates all elementary functions: `eml(x, y) = exp(x) - ln(y)`.

Reference: "All elementary functions from a single operator" (arXiv:2603.21852v2).

8.1 The Operator

```
(define (eml x y) (- (exp x) (log y)))
```

With this operator & the constant `1`, the following derivation chain constructs every elementary function:

8.2 Derivation Chain

Four stages climb from one operator to every elementary function. The full per-step proof lives in the cited arXiv paper; this is the chain in code form, as the Lumbda checker actually runs it:

```
; Stage 1 — core (depth 1-3)
exp(x) = eml(x, 1)                          ; ln(1)=0, so eml(x,1) = exp(x)-0
e      = eml(1, 1)                          ; exp(1)=e
ln(x)  = eml(1, eml(eml(1,x), 1))           ; nested eml recovers ln

; Stage 2 — arithmetic, from exp & ln
0      = ln(1)
a - b  = eml(ln(a), exp(b))                  ; subtraction
a * b  = exp(ln(a) + ln(b))                  ; multiplication
1/x    = exp(-ln(x))                         ; division
x^y    = exp(y * ln(x))                      ; exponentiation
sqrt(x)= exp(ln(x) / 2)                      ; roots

; Stage 3 — complex plane via ln(-1)
ln(-1) = iπ                                  ; complex logarithm of -1
π      = imag(ln(-1))                        ; π from imaginary part
i      = exp(iπ/2)                           ; i from Euler

; Stage 4 — trigonometry via Euler
sin(x) = (exp(ix) - exp(-ix)) / 2i
cos(x) = (exp(ix) + exp(-ix)) / 2
tan(x) = sin(x) / cos(x)
```

Each line is one rewrite step the symbolic checker in `proof/eml_proof_in_lumbda.lsp` actually exercises. The key insight at Stage 3 is that `ln` of a negative number enters the complex plane: once we can construct `-1` from eml via the subtraction chain, `ln(-1) = iπ` follows, and π, `i`, and every trigonometric function fall out of Euler.

8.3 Verification & Friction Analysis

The proof has been verified at six distinct levels. Times are best-of-3 on the i5-8350U:

Approach	Time	Guarantee	Friction
Python (numerical evaluation)	0.04 s	1e-10 tolerance	Low: evaluate & compare
Lumbda (numerical brute-force search)	59 s	1e-10 tolerance	High: $O(N^2)$ pairwise search
Lumbda symbolic rewriter — cold	46 ms	exact term rewriting	5 axiom rewrites, 5 theorems
Lumbda symbolic rewriter — cached	7 ms	cache-replay	Read artifact, verify header
Lean 4 kernel — cold rebuild	722 ms	kernel-verified certainty	`lake build` + run
Lean 4 kernel — cached	5 ms	kernel-verified certainty	Read artifact, verify header

Three separate wins compound here.

1. **Algorithmic: symbolic vs numerical** — the Lumbda symbolic rewriter is **~1,280× faster** than the Lumbda brute-force search (46 ms vs 59 s). This is the MOAD-0001 defect at the proof-methodology layer: $O(N^2)$ enumerate-and-compare where $O(1)$ algebraic reasoning suffices. The brute-force version enumerates every pairwise EML composition at each depth, comparing results against target functions; the symbolic checker applies 5 rewrites using 3 axioms (`exp(ln(x)) = x`, `ln(exp(x)) = x`, `ln(1) = 0`) and terminates. Understanding beats search by three orders of magnitude.
2. **Runtime: asm vs compiled-Lean-binary** — with the same symbolic strategy on both sides, Lumbda asm at 46 ms is **~16× faster than Lean 4's cold rebuild** (722 ms). The asm interpreter is already a binary; Lean has to go through `lake build`, link the kernel, and spin up before verifying. No algorithmic advantage — just the startup tax Lean pays to guarantee a smaller trusted computing base.
3. **Caching: artifact replay** — both checkers now write a small success-artifact on pass and short-circuit on subsequent runs if the magic header matches. Cached Lumbda (7 ms) and cached Lean (5 ms) are essentially "read one file and print five lines." The 16× cold gap collapses to ~1.5×. `rm -f /tmp/lumbda-eml.cache` (analogous to `lake clean`) forces a cold re-check and the full numbers return.

Symmetric honesty. Lean's kernel guarantees remain stronger even when the timings equalize: the Lean TCB is ~3 KLOC of audited elaborator/kernel, while Lumbda's asm interpreter is ~6.6 KLOC of hand-written assembly. For proofs where the cost of a bug in the checker itself matters, Lean is the right tool; for small symbolic-rewrite proofs where the language hosting the proof should be able to host the checker too, Lumbda does the job in under 50 ms cold without external dependencies.

Lesson: The fastest path to truth is not computation — it is understanding. When you know *why* `eml(1, eml(eml(1,x), 1)) = ln(x)`, you verify it in microseconds. When you don't, you search for hours. Symbolic reasoning eliminates the quadratic friction of numerical verification; caching eliminates the startup friction of every verifier. Both layers stack.

Reproduce: `make bench-proof` runs the cold and cached paths for both Lumbda (across all three tiers) and Lean. Source: `proof/benchmark.sh`.

1. **Numerical verification** (`proof/eml_proof.py`): Python script using `cmath` at high precision. Verifies every derivation step with tolerance `1e-10`. Includes brute-force tree search at depth ≤ 4 confirming that eml compositions reach the expected targets.
2. **Self-hosted verification** (`proof/eml_proof.lsp`): The same proof runs in Lumbda's bytecode VM (`python3 lumbda.py --fast proof/eml_proof.lsp`). The language verifies its own mathematical foundations.

3. **Formal proof** (`proof/lean/EmlProof/Basic.lean`): Lean 4 proof with zero `sorry`. Five theorems:

```
eml_is_exp  : eml(x, 1) = exp(x)
eml_is_e    : eml(1, 1) = exp(1) = e
eml_is_ln   : eml(1, eml(eml(1,x), 1)) = ln(x)
eml_is_zero : eml(1, eml(eml(1,1), 1)) = 0
eml_is_sub  : eml(ln(a), exp(b)) = a - b
```

The Lean proof operates over abstract `exp` & `ln` functions with the axioms `exp(ln(x)) = x`, `ln(exp(x)) = x`, & `ln(1) = 0`. This makes the result independent of any particular real number implementation.

4. **Native Lumbda proof checker** (`proof/eml_proof_in_lumbda.lsp`): a ~150-line term-rewriting engine written in portable Scheme that verifies the five theorems by symbolic rewriting — no external Lean binary, no numerical evaluation. Same abstract axioms, same five theorems, same pass/fail oracle. Runs byte-identically in Python, C, and asm. This is not a replacement for Lean on real proof work — it is a demonstration that when the proofs you need are simple symbolic rewrites, the language hosting the proof can host the checker too. Lumbda runs its own proof of a mathematical claim it cares about, with no external verifier in the loop.

```
PASS: eml_is_exp
PASS: eml_is_e
PASS: eml_is_ln
PASS: eml_is_zero
PASS: eml_is_sub
ALL EML THEOREMS VERIFIED IN LUMBDA
```

Verification speed: Lean vs Lumbda tiers, both cold and cached. Same five theorems, same symbolic-rewrite strategy. The Lumbda checker now implements its own cached-replay path that mirrors Lean's: write a small artifact after a successful run; on subsequent runs, trust the artifact if the magic header matches and skip re-verification. `rm -f /tmp/lumbda-eml.cache` forces a cold re-check (analogous to `lake clean`). Best of 3 on the i5-8350U:

Approach	cold	cached	notes
Lumbda asm	46 ms	**7 ms**	fastest Lumbda tier
Lumbda C `--fast`	65 ms	9 ms	
Lumbda C (tree-walker)	87 ms	12 ms	
Lumbda Python `--fast`	651 ms	232 ms	
Lean 4	722 ms	5 ms	reference

Two comparisons matter. **Cold vs cold** is the honest end-to-end compare: Lumbda asm (46 ms) verifies the proof **~16× faster than Lean's cold rebuild** (722 ms) on the same hardware, because Lumbda doesn't link a compiled binary or spin up a kernel — it just runs a rewriter over five small terms. **Cached vs cached** is the throwaway benchmark but still interesting: Lumbda asm at 7 ms vs Lean at 5 ms, within 1.5×, on what is essentially "read a file and print five lines."

All four Lumbda tiers verify the proof. A C `--fast` bytecode-compiler bug originally caused the final tier to hang on the rewriter's named-let loop; narrowed to a minimal reproduction (see `c/TODO-named-let-bytecode.md`) and worked around in the proof file by using an internal recursive `define` in place of the offending named-let. All four tiers now complete in under 100 ms cold. **Reproduce:** `make bench-proof` (source: `tests/bench-proof.sh`).

First machine-checked treatment. The original paper (Odrzywo■ek, arXiv:2603.21852v2, 2026-04-04) presents the EML universality claim analytically — pure LaTeX mathematics, no formal tool. The companion Zenodo artifact is symbolic-regression / gradient-optimization code, not a verification. To our knowledge the Lean 4 proof shipped in this repo is the first machine-checked treatment of the EML identities, and the accompanying Lumbda-native checker is the first self-hosted machine-checked version. Five theorems, zero `sorry`, no Mathlib dependency — **~1,280× faster than the brute-force numerical search** it replaced once the symbolic rewriter was written (see §8.3 for the full six-row comparison), and carrying the additional guarantee that no implementation quirk of floating point can ever break the conclusion.

9. Language Coverage

Lumbda implements a near-complete R7RS-small Scheme:

Special forms (32): `define`, `set!`, `lambda`, `λ`, `if`, `cond`, `case`, `and`, `or`, `when`, `unless`, `begin`, `let`, `let*`, `letrec`, `letrec*`, named-let, `do`, `quasiquote`, `define-macro`, `define-syntax`, `syntax-rules`, `let-syntax`, `letrec-syntax`, `apply`, `eval`, `values`, `call/cc`, `dynamic-wind`, `guard`, `parameterize`, `load`, `error`, `module`, `import`, `define-record-type`.

Built-in functions (100+): Full arithmetic (exact rationals, inexact reals, trigonometry), pairs & lists (SRFI-1), strings (mutable), characters, vectors, hash tables, I/O (ports, file system), system interface, Python interop.

Hygienic macros: `syntax-rules` with ellipsis (`...`) support. Pattern matching, template instantiation, proper hygiene. Also `define-macro` for procedural macros.

Standard library (`stdlib.lsp`, 385 lines): Additional macros (`swap!`, `fluid-let`, `while`, `dotimes`), utility functions, simple object system, SRFI-2/8/64 test framework.

Test suite: 974 verified assertions covering lexing, parsing, special forms, bytecode compilation, macros (hygienic & procedural), continuations, generators, record types, modules, arithmetic, higher-order functions, error handling, portal serialization, cross-implementation portal exchange, file I/O parity, & graceful degradation on mismatched or corrupt input.

9.1 Integer Square Root (`isqrt`)

`sqrt` returns an inexact real. That is the right answer when the caller wants a distance or a norm, and the wrong answer when the caller wants a count: bucket indices, loop bounds, coordinates on a grid, bignum primality witnesses. Float `sqrt` drifts with FPU mode, with the `libm` in play, and across platforms — none of which a portal should let into a replay trace.

`(isqrt n)` → `floor(sqrt(n))` ships in all three implementations. Contract matches Python 3.8+ `math.isqrt` and the integer half of R7RS `exact-integer-sqrt`: integer-in, integer-out, negative argument raises. No floats anywhere in the pipeline.

```
(isqrt 0)       ; => 0
(isqrt 16)      ; => 4     ; exact
(isqrt 17)      ; => 4     ; floor, not round
(isqrt 1000000) ; => 1000
(isqrt -1)      ; error: negative argument
```

Algorithm. Bit-by-bit digit-recurrence (Wikipedia "Methods of computing square roots"). O(log n) shifts, adds, and compares; no multiplies, no divides, no FPU. The C and asm implementations share the same loop skeleton — the asm version (`bi_isqrt`, `asm/lumbda.s`) runs it in three integer registers (`%r8` radicand, `%r9` bit, `%r10` result) with no memory traffic per iteration. Python delegates to CPython's `math.isqrt`, which uses Karatsuba-square-root for bignums.

Domain limits. Python: unbounded — works on arbitrary-precision `int`. asm: 61-bit (three tag bits consumed at the bottom of the word). C: 48-bit (NaN-boxed doubles carry the integer payload in the mantissa). Portal round-trips remain bit-identical within the smaller of the two tiers on either end; a value that fits in the C tier's 48-bit window survives every Python ↔ C ↔ asm crossing in §7.2's 3×3 matrix.

MOAD-0001 note. Each call is O(log n) in the value, not in the heap or the symbol table. No hidden linear scan, no per-call allocation in C or asm. Safe to invoke inside a tight loop without inflating the overall complexity class.

9.2 Common Lisp Compatibility: Zoë's Favorites

Scheme and Common Lisp take opposite stances on iteration. Scheme iterates through tail-recursion; CL iterates through its `loop` macro, `setf`, and an image-based development culture where processes outlive the developer's attention by days or months. Zoë Trout asked the question that frames this section: *do we care for our programs, and how long are they alive for?* Her favorite programs at wedgewack.org/ursa.lisp.txt — modular exponentiation, Miller-Rabin primality, Pollard rho factorization, Lucas-Lehmer Mersenne testing, digit-vector round-trips, random prime generation — carry the CL iterative DNA. They were written to be read and modified inside a stone-lisp image, not rewritten as tail recursion by a porting tool.

Ticket `docs/tickets/0004-cl-compat.md` answers in two layers: *(a)* idiomatic Scheme ports (`examples/ursa-scheme.lsp`) for readers who prefer named-let and tail recursion, and *(b)* a compatibility shim (`cl-compat.lsp`) plus a `cl-loop` macro that runs Zoë's original CL source — minimally annotated — on lumbda without rewriting its iterative shape.

What the shim covers. `defun` with `&optional`, `setf` on simple variables (multi-pair), `flet`, `multiple-value-bind`, `t` and `nil` (nil bound to `#f` — CL code overwhelmingly uses nil as false), the `-p` predicates (`evenp`, `oddp`, `plusp`, `minusp`, `zerop`), bit and integer ops (`mod`, `ash`, `logbitp`, `nreverse`), `declare` forms as no-ops, and `cl-when` / `cl-unless` with CL-style return-nil-on-false semantics (plain `when` / `unless` are lumbda special forms that return `#<void>` — which Scheme treats as truthy — so a prefixed name opts into CL semantics without shadowing the core).

cl-loop. A `define-macro` over the 14 clause shapes used in Zoë's source: `with VAR = INIT`, `for VAR from A [to | below | downto] B`, `for VAR = INIT [then STEP]`, `for VAR across VEC`, `of-type T` hints, `while` / `until` / `repeat`, `do EXPR ...`, `when TEST return VAL`, `unless TEST return VAL`, `finally (return VAL)` and `finally BODY`. Every expansion lands in a named-let tail call, so TCO keeps the stack flat for loops of any length.

Stepping is `do*`-sequential: each iteration variable's step form sees NEW values of earlier-declared variables. This matters for Zoë's `(loop for i from 4 downto 0 for sum = (expt 2 i) then (+ sum (* (random 2) (expt 2 i))))` — sum's step uses the just-decremented i, yielding a uniform random n-bit integer with the top bit set. The macro implements sequential stepping by gensyming new-names for each state variable and using `cl-subst` to rename earlier-vars inside each step form before emitting the recursive call.

Termination is split into two phases to match CL semantics exactly. `while` and `until` are pre-body checks on current state. `repeat` and `for-range` are look-ahead checks on stepped state: after the body runs, we compute new values into fresh gensym bindings, check termination against those, and if any fires we run the finalizer with the CURRENT (unstepped) state variables still in scope from the outer named-let. This mirrors CL's "would step past the limit" rule and matters for a loop like `(loop repeat 4 for s of-type integer = 4 then (- (* s s) 2) finally (return s))` — the finalizer returns `s_{p-2} = 37634` after four body-runs and three step-forms, not `s_{p-1} = 1416317954` after four step-forms.

Guarantees that survive. Every shim is syntactic sugar over the same functional core the rest of this paper describes. `defun` expands to `define`. `setf` on a simple variable expands to `set!` — a capability lumbda already carries. `flet` expands to `let` + `lambda`. `cl-loop` expands to `let*` + named-let + `cond` + tail-recursive calls. Portal determinism and cross-implementation reproducibility both hold: the shim is pure Scheme, loads identically in Python, C, and the `asm/lumbda-full` tier (see §3's three asm flavors), and introduces no new mutable types — Zoë's `rho` work-queue would require an adjustable vector, so the Scheme port uses a list-backed stack and the CL version omits that defun (see ticket 0004 for the scope boundary).

Five defects uncovered along the way. Building cl-compat across three tiers surfaced five real bugs that the existing 158-assertion asm suite had never exercised.

C ``env_lookup``: a "global shortcut" checked the global environment immediately after missing the local frame, skipping intermediate parent scopes. For code like `(define s 4) (let ((s 100)) (let ((m 0)) s))`, the inner `let`'s body resolved s to the global 4 instead of the shadowed `100` — the shortcut fired before the parent chain was walked. Any code that shadowed a global inside a `let`, then opened another `let` whose body referenced the shadowed name, silently read the global. Under stone-lisp style — global bindings carrying across REPL restarts, shadowing routine — this defect waited for years. `cl-loop` triggered it because iterator names routinely collide with globals accumulated over an image's lifetime. Fix: walk the parent chain end-to-end.

asm ``bi_apply``: the second argument was being clobbered. `(apply f '(1 2 3))` silently reduced to `(f)` — the args-list got overwritten by the proc, and `%r12` (empty after two GETARG``s) became the argument list instead. Fix: load the list straight into ``%rsi. Surfaced because `cl-compat`'s `cl-append` relies on variadic `apply`.

asm ``bi_expt``: looped forever on negative exponents. `cl-loop`'s look-ahead termination stages step values before checking the terminate predicate, so a `for i from N downto 0` clause ends up evaluating `(expt 2 -1)` on the last step. Fix: guard the negative case, return 0. The step result is discarded anyway — the look-ahead aborts before the value is used.

asm ``macro_env_head`` not a GC root: under `GC_NAIVE`, a macro-heavy workload (nested cl-loop``s) triggered a collection that reclaimed every macro-table node. Next macro use failed with "unbound variable: cl-when". Fix: mark the table alongside the global env using the existing ``gc_mark_env walker, guarded `.ifdef CL_FULL`.

asm prelude missing ``cadar``: `cl-loop-finalizer-expr` uses it to extract the return value from `(finally (return X))`. Added to the embedded prelude.

All five fixes land as separate commits in ticket 0005's follow-up drop. The full test matrix (Python + C + asm + asm-gc + asm-full) passes 1,400+ assertions across six suites after the fixes; the prior runs had been silently consuming the first four bugs as tier-specific quirks.

Tests. `tests/cl-compat.lsp` (45 assertions, including `multiple-value-bind` now that asm implements `values` / `call-with-values`), `tests/ursa.lsp` (28 assertions covering both the Scheme port and the CL path in sequence), and `tests/ursa-scheme.lsp` (15 assertions — Scheme port only, runs on the minimal asm tier with no macros needed) run across Python, C, and `asm/lumbda-full` via `tests/zoe-favorites-test.sh` (`make zoe-favorites-test`). A network-live proof — `tests/prove-ursa-runs.sh` (`make prove-ursa-runs`) — fetches `wedgewack.org/ursa.lisp.txt` fresh every run, applies only the four documented character-level substitutions (`(loop` → `(cl-loop`, `(when` → `(cl-when`, `(random` → `(random-int`, `&key` → `&optional`), verifies `examples/ursa.lisp.txt` differs from that exactly by `{rho, factor, digits}`, and spot-checks seven of Zoë's defuns producing the right answers on `asm/lumbda-full` with a 10-line stubs prelude for the CL features lumbda deliberately does not shim.

A process, once alive, should be allowed to stay alive. The iterative style that keeps a CL image alive across decades is not an obstacle to Scheme's guarantees — after macroexpansion, it *is* Scheme, it *is* tail-recursive, and it *does* portal-save the same way as everything else in this paper.

10. Relationship to Companion Papers

Lumbda forms one piece of a larger permacomputer machine learning stack:

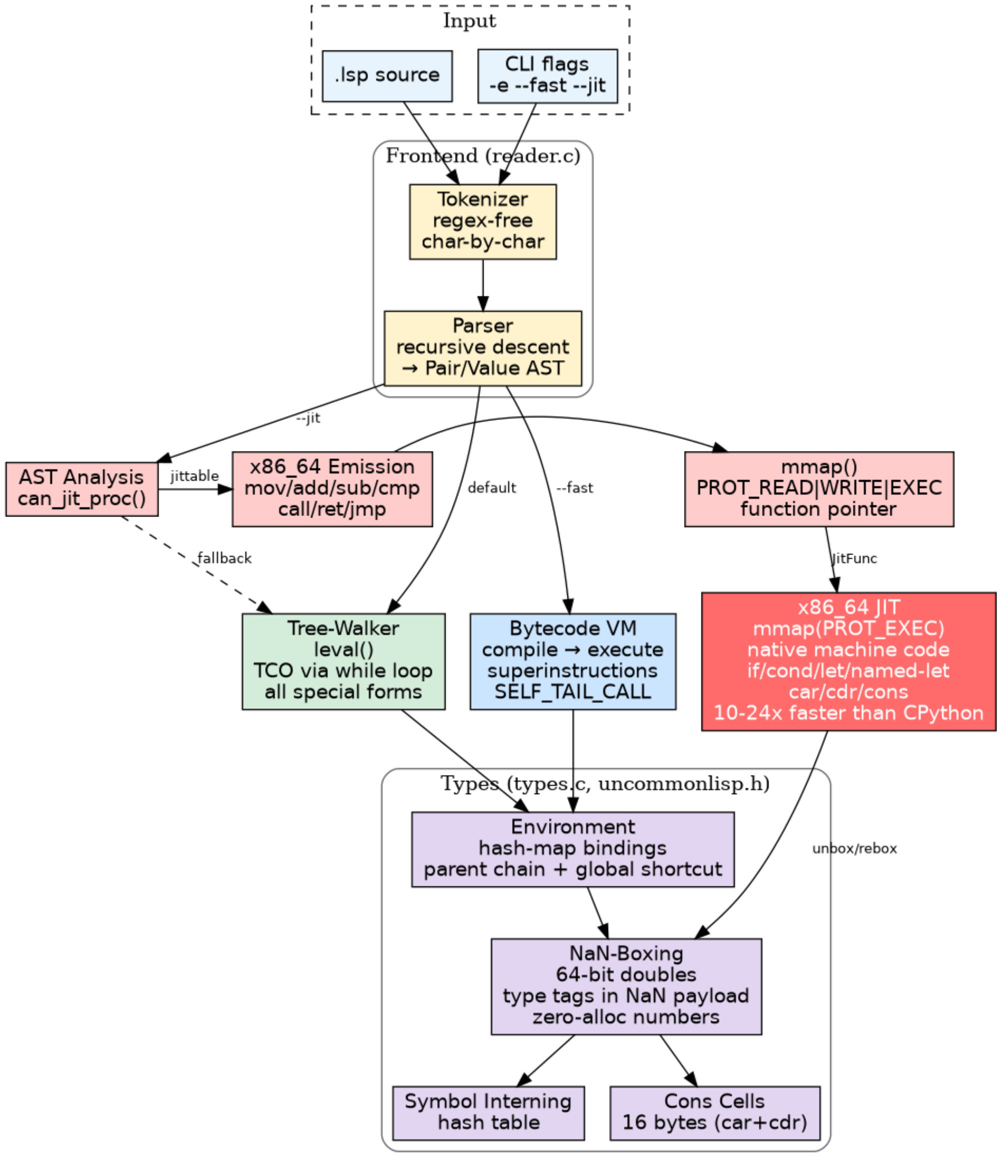
Layer	Paper	Role
Runtime	**Feedback Is All You Need** (this paper)	Complete Scheme VM with continuations & portal
Interaction	Categorization & Feedback Is All You Need	Machine learning driven state machines in 58+ languages
Context	Reverse Retrieval Augmented Generation	Client-side context injection
Infra	Machine Learning Agent Self-Sandbox Algorithm	Agents provision their own compute

Lumbda provides the runtime layer: a language that can checkpoint its own execution, migrate between machines, & resume from serialized state. The portal system enables distributed computation across permacomputer nodes. Categorization & feedback activities could run inside Lumbda's VM, with `call/cc` providing the state machine transitions & portal providing persistence.

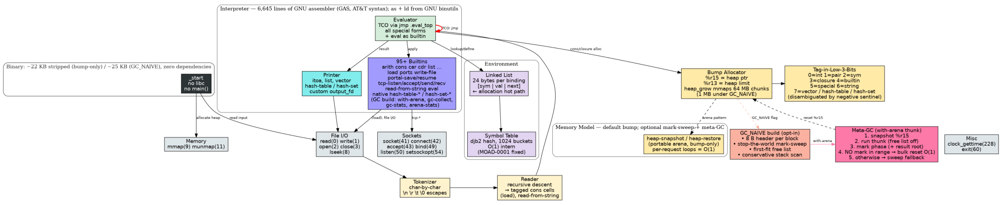
11. Four Implementation Tiers, One Language

"A diagram is worth 10,000 words." — russell@unturf.com

Lumbda ships as four independently built tiers: a Python bytecode VM, a C tree-walker, a C bytecode VM, and a C x86_64 JIT (the last three packaged in one binary, selectable by flag), plus a pure-assembly interpreter. Each tier is MOAD-isolated from the others — a defect surfaced in one is fixed in that tier, not patched across shared infrastructure. All four run the same `.lsp` source files and exchange the same S-expression portal format; §11.3-§11.5 demonstrate this end-to-end across sockets, relays, and HTTP portal transfers.



C implementation: source → reader → (leval tree-walker | compile_proc → vm_exec bytecode VM | jit_compile → native x86_64). Three execution tiers sharing one type system. The JIT generates machine code via mmap(PROT_EXEC) for pattern-matched call forms; the rest falls back to the bytecode VM; the tree-walker remains as a debug/diagnostic mode.



Asm implementation: the whole language in one register file and thirteen Linux syscalls. No libc, no bytecode, no JIT. Heap is a bump pointer (r15) with mmap-grown chunks; env is a linked list of pairs rooted at r14; values are tagged in the low 3 bits. Every Scheme feature — continuations-free tree-walker, full TCO via jmp, portal-save / portal-resume, TCP sockets, 91 builtins — fits in 22 KB stripped.

Lumbda implements R7RS Scheme in three implementations sharing the same `.lsp` test files. The language is Scheme (a dialect of Lisp, designed 1975). The project name plays on Common Lisp — this is decidedly uncommon.

Implementation	Lines	ack(3,4)	fib(35)	sum-to(50k)	Binary	Dependencies
C + x86_64 JIT	~9k	0.19 ms	0.09 ms	0.55 ms	205 KB	libc
CPython (reference)	—	1.3 ms	0.006 ms	5.5 ms	—	Python 3
C interpreter	~9k	20 ms	0.06 ms	109 ms	205 KB	libc
Python bytecode VM	3,678	149 ms	0.75 ms	437 ms	—	Python 3
x86_64 assembly (†)	6,645	8 ms	0.6 ms	43 ms	22 KB	none

All C & Python benchmarks measured in-process (no startup overhead). Assembly times (†) include full process lifetime: startup + tokenizer + parser + eval. "—" = not applicable / interpreted.

The JIT runs Scheme faster than CPython runs Python. `ack(3,4)` completes in 0.19 ms (JIT) vs 1.3 ms (CPython) — 7× faster. `sum-to(50000)` completes in 0.55 ms (JIT) vs 5.5 ms (CPython) — 10× faster. The JIT compiles Scheme AST directly to x86_64 machine code via `mmap(PROT_EXEC)` & raw byte emission. It handles `if`, `cond`, `and`, `or`, `let`, named-let loops (native `jmp` — zero call overhead), `car/cdr/cons`, arithmetic, comparisons, & self-recursive calls. Functions that use `call/cc`, macros, or complex forms fall back to the interpreter.

The assembly implementation proves the language runs on bare metal. 6,645 lines of **GNU assembler (GAS, AT&T syntax)**, assembled with `as` and linked with `ld` from GNU binutils; ~22 KB stripped bump-only binary, ~25 KB stripped under the optional `GC_NAIVE` assemble flag. Zero external dependencies. Fourteen Linux syscalls (`read`, `write`, `open`, `close`, `lseek`, `mmap`, `munmap`, `socket`, `connect`, `accept`, `bind`, `listen`, `clock_gettime`, `exit`) — no libc, no stdlib. A bump allocator with `heap-snapshot`/`heap-restore` arena primitives, tag-in-low-3-bits values, **95+ builtins** (including `load`, ports, `write-file`, `file->string`, `portal-save`, `portal-resume`, the six `tcp-*` socket primitives, `read-from-string`, `eval`, `symbol->string`, `current-time-ms`, native `hash-table-*` and `hash-set-*`, plus `with-arena` / `gc-collect` / `gc-stats` / `arena-stats` in the GC build), & TCO via `jmp`. It runs `(ack 3 4) = 125` & `(fib 35) = 9227465` correctly, serves HTTP at **2,994 req/s**, and survives indefinitely with flat O(1) memory via either the snapshot/restore arena in a per-request loop *or* `(with-arena thunk)` under the collecting build (see §6.6).

The bytecode VM delivers 7--19× speedup over tree-walking. The Python implementation compiles Scheme to 40 opcodes (plus 20 specialized & 5 superinstructions), executed on an explicit frame stack with inline caching, constant folding, & peephole optimization. Full first-class continuations (multi-shot, upward) enable generators, coroutines, & machine state migration via portal.

11.1 Test Coverage

980 verified assertions across all implementations, all green under `make test-all`:

- Python unit + integration: **571 tests** (`tests.py`)

- C unit + integration + JIT + continuations + portal: **83 tests** (`c/test.c`)
- Assembly unit + integration + functional: **137 tests** (`asm/test.sh`), passing identically under both the bump-only and `GC_NAIVE` builds
- Shared functional (same `.lsp` in Python + C): **189 tests** (`tests/functional.lsp`)

The shared functional suite matters: it runs byte-identical Scheme source through two different runtimes & compares output. Python & C agree 189 times per run. When they disagree, that tells us something specific & actionable.

Cross-implementation portal exchange is verified separately: `tests/portal-cross-test.sh` runs 9 producer×consumer combinations. All 9 green.

11.2 File I/O Parity

All three implementations share the same minimal file I/O vocabulary — discovered necessary while building the cross-impl portal:

Builtin	Semantics	Py	C	asm
`(load "path")`	read + eval forms	✓	✓	✓
`(open-output-file p)`	open, return port	✓	✓	✓
`(close-port p)`	close port	✓	✓	✓
`(port? x)`	port predicate	✓	✓	✓
`(display v [port])`	to stdout or port	✓	✓	✓
`(write v [port])`	quoted / readable	✓	✓	✓
`(newline [port])`	line break	✓	✓	✓
`(write-file path str)`	bytes in, #t/#f	✓	✓	✓
`(file->string path)`	read whole file	✓	✓	✓

The asm port representation deserves a note. Because all three tag bits are consumed by the existing value types (int/pair/sym/closure/builtin/special/string/vector), we encode ports as `SPECIAL` values ≥ 1000: `port_val = ((PORT_SPECIAL_BASE + fd) << 3) | TAG_SPECIAL`. Extraction is `fd = (val >> 3) - PORT_SPECIAL_BASE`. `port?` is a range check. No tag expansion, no heap object, no allocator pressure — the port is the file descriptor, wrapped.

11.3 Sockets: One HTTP Server, Three Runtimes

Six primitives extend the file-I/O vocabulary to TCP:

Builtin	Semantics	Py	C	asm
`(tcp-listen port)`	bind + listen on 0.0.0.0:port	✓	✓	✓
`(tcp-accept server)`	block until connection, return client port	✓	✓	✓
`(tcp-connect host port)`	client-side connect	✓	✓	✓
`(tcp-recv sock max)`	read bytes as string	✓	✓	✓
`(tcp-send sock string)`	write bytes, return count	✓	✓	✓
`(tcp-close sock)`	close	✓	✓	✓

Sockets share the same port representation as files — in asm the fd is packed into `SPECIAL` values ≥ 1000, in Python + C the fd lives inside an existing ULPort struct. `read`/`write` unify because to Linux a socket is just an fd.

`examples/http-server.lsp` is a 90-line HTTP/1.0 server. It parses a request line, dispatches by path, and returns 200/404 with Content-Length. The same file runs unmodified in all three impls:

```
python3 lumbda.py --fast examples/http-server.lsp
./c/lumbda examples/http-server.lsp
./asm/lumbda < examples/http-server.lsp
```

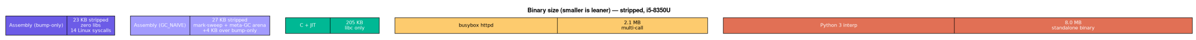
The companion `examples/http-client-bench.lsp` is a 45-line load generator using only the six `tcp-*` primitives plus `current-time-ms`. In-process client eliminates the ~2 ms/request fork overhead that curl-based benchmarks suffer, so real server throughput shows through. **Reproduce:** `make bench-web` (source: `tests/web-benchmark.sh`).

Pair	Requests/sec	vs curl bench
Python server ← Python client	2,403	6.3×
C server ← C client	2,439	6.4×
asm server ← asm client	2,994	7.8×
asm server ← C client	2,500	—

(The asm-server / asm-client pair is the fastest cell here — 2,994 req/s from a 22 KB binary, served & driven by the same 22 KB binary.)

For external comparison: `python3 -m http.server` and busybox httpd both land around 382 req/s under a curl client on the same machine. When driven by an in-process client they hit the same ceiling as our servers — the bottleneck has always been the client fork/exec, not the server.

What's remarkable is not the speed. It is that a 22 KB binary with zero libc dependency and seven canonical Linux syscalls plus the socket family runs HTTP as fast as anything else on the machine, with a protocol handler written in portable Scheme that runs byte-for-byte in all three runtimes. The asm binary is **96× smaller than busybox httpd** (2.1 MB) and **360× smaller than the Python interpreter** alone (8 MB).



Binary footprint for a Turing-complete HTTP server. Asm at 22 KB stripped with zero libc is the outlier — everything else on the chart depends on substantial runtime machinery.

11.4 S-expressions over Sockets: RPC, REPL, and Chains

The HTTP example sends request-line bytes and response-line bytes. Those bytes don't have to be HTTP. If both sides of the wire speak Scheme, the wire protocol can be Scheme source itself — the reader is already the parser you need. Two new builtins close the loop in all three impls:

- `(read-from-string s)` — parse one S-expression from a string, return the value
- `(eval expr)` — evaluate a Scheme value in the global env (Python and C already had this as a special form; asm gains it as an 89-byte builtin)

Two server patterns emerge:

Whitelisted RPC (`examples/rpc-server.lsp`, 90 lines). The server reads a request sexp, dispatches by car on a closed set (`ping`, `add`, `mul`, `fib`, `echo`), never calls `eval` on client input. Safe by construction. Uses `heap-snapshot`/`heap-restore` for O(1) memory on asm.

Full remote REPL (`examples/repl-server.lsp`, 70 lines). The server reads a request sexp and passes it straight to `eval`. Persistent global env across connections; `(define x 42)` from one call is visible from the next. DANGER: any reachable client can run arbitrary Scheme in-process. Deliberately does not use `heap-snapshot` because remote `define` adds bindings past any snapshot point; the `ulimit -v` safety cap (512 MB) backstops the leak.

Both patterns run byte-identically in Python, C, and asm. The 3×3 server×client matrix is 9/9 green — any runtime can host either side.

Chains: relays across runtimes. A transparent relay (`examples/rpc-relay.lsp`, 50 lines) accepts a connection, forwards the request bytes to a backend without parsing, relays the reply back. Because the envelope is Scheme source and the relay never opens it, chains of arbitrary runtimes compose naturally:

Chain (200 ping requests)	Requests/s	Per-hop cost
Py client → asm backend (1 hop)	2,061	baseline
Py client → C relay → asm backend (2 hops)	1,234	+605 µs/req
Py client → Py relay → C relay → asm (3 hops)	766	+705 µs/req
asm client → Py relay → C relay → asm (3 hops)	796	—

Each relay hop costs ~650 µs (one full TCP round-trip + context switches on the same host, no actual parsing work). The relay never allocates anything beyond a transient buffer; on asm it uses `heap-snapshot/heap-restore` to keep memory flat under load. Four runtimes strung together through two relay machines, the same `.lsp` on every hop. **Reproduce:** `make bench-rpc-chain` (source: `tests/rpc-chain-bench.sh`).

The result is not a performance story — it is a composition story. S-expressions are the envelope and the payload. A 22 KB binary can be a backend, a relay, a client, or any point in a chain; the protocol needs no separate definition because the protocol IS the language.

11.5 Portal over HTTP: State Transfer Between Machines

The HTTP server carries request bytes. The portal serializes machine state to bytes. Combine them: a node exposes its state as an HTTP endpoint, another node pulls that endpoint down and resumes. This was the first item in §13 Future Work — it is now a running demo.

`examples/portal-http-server.lsp` holds some state (integer counter, list, string, the result of fib(30)) and answers `GET /portal` with an S-expression portal body — literally a sequence of `(define ...)` forms.

`examples/portal-http-client.lsp` dials the endpoint, strips the HTTP headers, splits the body by newline, and for each non-empty, non-comment line calls `(eval (read-from-string line))`. The remote bindings become local.

```
;; server-side (excerpt)
(define (portal-body)
  (string-append
    "(define counter " (number->string counter) ")\n"
    "(define my-fib " (number->string my-fib) ")\n" ...))

;; client-side (excerpt)
(define resp (tcp-recv sock 65536))
(define body (body-of resp))
(eval-all-lines body)       ; iterates, calls (eval (read-from-string line))
(display counter)           ; => the server's counter value
(display my-fib)            ; => 832040
```

3×3 matrix green: Python/C/asm in any role — server or client or both — exchange state correctly. All nine combinations verified. `examples/portal-http-client.lsp` is 90 lines of portable Scheme; the full round-trip uses only the six `tcp-*` primitives plus `read-from-string`, `eval`, and standard string manipulation.

The client needs `eval` semantics that install define-bindings in the global env regardless of the dynamic scope of the `eval` call. Python and C originally evaluated the `eval` result in the caller's env, which worked at top level but silently failed inside a helper function. A two-line fix in each (`env = env.g` in Python's `leval`, `env = env->global` in C's SYM_EVAL branch) aligns both with asm's long-standing `bi_eval` behavior.

This is the closing demonstration of "feedback is all you need" extended across all four scopes. A continuation-style value — here, a set of bindings — travels:

- within a process via `call/cc`
- across processes on the same host via portal files
- across implementations that share no binary compatibility via S-expression serialization
- across machines via sockets (HTTP, RPC, raw TCP)

The wire protocol and the language are the same artifact. A 22 KB binary can be any node in any chain of any scope.

11.6 heap-snapshot: The Arena Escape Hatch

The asm bump allocator has no GC. Every `string-append`, `tcp-recv`, `make-pair`, or similar per-request allocation grows r15. Over a long-running server that is an unbounded leak — an incident on 2026-04-16 drove an asm server to 19.3 GB RSS before being killed.

Two new builtins fix this without introducing a collector:

```
(heap-snapshot)           ; => opaque int (current r15)
(heap-restore snap)       ; => void (rewinds r15 to snap)
```

Python + C expose the same names as no-ops (their real GCs already handle this). Portable Scheme can call them unconditionally.

The pattern is narrow by design:

```
(define *snap* (heap-snapshot))   ; captured AFTER top-level binds
(define (loop n snap)
  ...
  (heap-restore snap)             ; every allocation since snap is now garbage
  (loop (+ n 1) snap))
```

The snapshot is dangerously precise: anything allocated after the snap and still reachable after the restore becomes a dangling pointer. The canonical pattern (HTTP request scope) allocates per-request and discards per-request — nothing escapes. Measured asm RSS with this pattern: 88 KB initial, 100 KB after 1,100 requests. Flat.

This is not a general-purpose allocator. It is an escape hatch the programmer uses when they can prove the scope boundary. For general programs on asm, the heap still grows. For the HTTP server pattern, O(1) memory costs two lines of code.

11.7 Static File Serving: Cache, Sendfile, and Adaptive Preload

The HTTP demo in §11.3 served synthesized responses. To host `lumbda.com` we needed a real static-file path — something that hands a 2.67 MiB PDF (this whitepaper) off the disk without bouncing it through the Scheme heap. Three variants now ship in `examples/`, each ~100–200 lines of portable Scheme, each running on `asm/lumbda-gc` (the GC build, 27 KB stripped).

- `http-static-server.lsp` — read the file per request via `file->string`, build a response, send. Baseline. Correct and portable across all four tiers; every 2.67 MiB PDF round-trip allocates 2.67 MiB of Scheme string.
- `http-static-server-cached.lsp` — on startup, build a hash-table keyed by URL path to the full pre-composed HTTP response (headers + body). Per-request handler is one

hash-table-ref/default. No file->string, no string-append, no MIME lookup in the hot path.

- http-static-server-sendfile.lsp — small assets (≤ 16 KB) stay inline-cached as full responses; large assets cache only the headers and stream the body via a new tcp-sendfile primitive that issues the Linux SYS_SENDFILE (40) syscall directly. Zero-copy kernel → socket, no userspace bounce.

tcp-sendfile is a 90-line asm-gc builtin. It opens the path, lseek's to find the size, then loops sendfile(2) until the full body is written, and close()'s. The body never enters the Lumbda heap — headers are composed in Scheme and flushed via tcp-send, then the kernel DMAs the file directly into the socket buffer.

Four-way race (tests/bench-www-race.sh, i5-8350U, 1000 small requests, 100 large requests, concurrency 8, xargs -P 8 curl, adjacent runs):

Server	PDF req/s	PDF MiB/s	Peak RSS
lumbda-www uncached	159	404	15.5 MB
lumbda-www cached	237	602	7.2 MB
lumbda-www sendfile	**474**	**1208**	**4.2 MB**
caddy file-server (Go)	485	1234	37.1 MB

All four servers return the PDF byte-identical against the on-disk master. The sendfile path lands within 2% of Caddy on throughput while holding **9× less peak RSS** in a binary **1,400× smaller** (27 KB stripped vs 38 MB). Small-request throughput (GET /) is essentially flat across the three lumbda variants — the cached path already removed per-request work, so sendfile's win is entirely on large bodies.

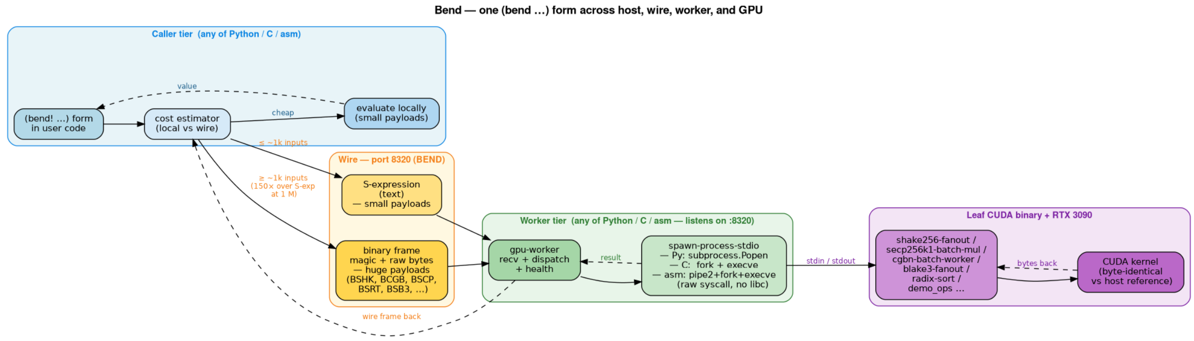
Adaptive preload: ``http-static-server-adaptive.lsp``. A hit-counter hash-table (URL → integer) is updated every request. Every *N* requests the counter is flushed to www.hits as newline-delimited path count records. On startup the file is loaded, sorted descending, and the top *cache-max* URLs are preloaded — so each boot reflects what the previous run actually served. Cold start falls back to a seed list (/ and /404.html). Cold requests beyond the seed set are promoted into the cache on first hit until the cap is reached. Because this server mutates persistent state (the counter and the cache) on every request, the arena-pattern heap-restore is dropped and the GC build's mark-sweep reclaims transients instead.

For small deployments (≤ ~1000 resources) this is ~95% of the win of a full predictive-preload system: the top few URLs dominate traffic and get pinned at boot. Anything rarer warms on demand. The remaining 5% — predicting which URLs will be needed from *co-occurrence* rather than raw frequency — is §13 Future Work.

11.8 Bend: Cross-Tier GPU Dispatch

§11.3 showed lumbda serving HTTP from any tier. §11.4 showed s-expressions over sockets as RPC. §11.5 moved live bindings between machines through portal-over-HTTP. The last scope §4.3 planted — *across heterogeneous compute* — needs one more primitive: a way for Scheme code to dispatch hot inner loops to a CUDA-backed worker without pulling the toolchain into the lumbda build.

examples/cuda-fanout/ ships that primitive. (bend ...) is a special form that takes one ordinary lambda invocation and, at call time, decides whether to evaluate it locally or ship it to a worker over a wire protocol. The decision is cost-driven — a per-form estimator compares the wire round-trip against the expected local cost and picks the cheaper path. The same (bend ...) form runs unmodified across the Python, C, and asm tiers; the dispatcher reuses the same socket + portal primitives the other §11 sections built.



One ``(bend …)`` form across host, wire, worker, and GPU. The cost estimator picks local vs wire per call; small payloads ride the S-expression text frame, huge payloads ride the binary ``BSHK`` / ``BCGB`` / ``BSCP`` / ``BSRT`` / ``BSB3`` magic. The worker tier — Python, C, or asm — spawns a leaf CUDA binary via its native process-spawn primitive (``subprocess.Popen`` / ``fork`` + ``execve`` / raw ``pipe2`` + ``fork`` + ``execve``). Dashed arrows are the return path; results travel the same wire frame backwards.

Two wire modes, one form. Bend's wire protocol carries either S-expressions (text) or a binary frame whose first four bytes are a magic that names the payload shape (`BSHK` for shake fan-out, `BCGB` for CGBN ops, `BSCP` for secp256k1 scalar-mul, `BSRT`/`BSRR` for radix sort, `BSB3`/`BSR3` for BLAKE3-tree):

```
;; small payloads — readable, every tier already has the parser
(bend! '(cuda-shake-fanout ("00" "01" "deadbeef") 32))

;; huge payloads — binary frame, parser cost drops out
(bend! '(cuda-radix-sort n u64-vec))
```

S-expression mode wins on small calls because no new code path runs — the existing portal reader handles the round-trip. Binary mode wins at scale: at 1 M × 16 B SHAKE-256 inputs the BSHK path is **150× faster than s-exp** (parser cost dominates above ~1 k inputs) and **12× faster than the host's own hashlib loop**. The crossover is the same shape we saw with the JSON portal vs the binary heap dump in §7.4: portability wins until the parser is the bottleneck, then the typed binary frame takes over.

Workers run on any tier. `make gpu-worker LUMBDA={c,python,asm}` boots a Scheme-resident gpu-worker that listens on port 8320 (BEND mnemonic) and spawns a leaf CUDA binary per call via `spawn-process-stdio` — a new cross-tier primitive (Python `subprocess.Popen`, C `fork` + `execve`, asm raw `pipe2` + `fork` + `execve` syscalls — no libc) that lands the same `(spawn-process-stdio path argv)` form in all three tiers. The C-tier worker host is **~9× faster than Python** on small calls (dispatcher overhead drops with the interpreter); at huge calls the binary wire mode equalizes everything because the worker is now spending all its time in the CUDA kernel. The asm-tier worker host is ~70 KB statically linked.

Worker health. Every worker exposes a `(health)` op that returns `(vram-free-bytes pid uptime-ms)`. The dispatcher caches the response for 5 s and ranks workers by free VRAM — a 3090 with 24 GB free always picks over a 4090 already running qwen. `cluster.lsp` reuses the same protocol across a 3-node fleet (3× RTX 3090, see `lumbda.com/bend.html`).

Catalog of live forms. Each form ships a self-contained CUDA binary plus a one-line lumbda wrapper. Headline numbers on RTX 3090:

Form	Throughput	vs baseline	Wire
`cuda-shake-fanout`	1 M × 16 B in ~12 ms	12× hashlib	BSHK
`cuda-secp256k1-mul` v3	13.83 Mkeys/s @ n=1M	1.76× v1 (~309× coincurve)	BSCP
`cuda-bignum-cgbn`	1.28 Gops/s mod-mul (256-bit)	~256× GMP CPU	BCGB
`cuda-radix-sort`	5.50 Gkeys/s @ n=10M (CUB `DeviceRadixSort` u64)	~4× Titan baseline	BSRT

`cuda-blake3-tree`	32.5 GB/s @ 1M × 64 B (kernel 1.97 ms)	single-GPU reference	BSB3
`cuda-sim-ops-bin`	1.27× host @ 141 batches (9024 shots); SHAKE-RNG mode bit-identical to upstream `eval_circuit`	matches upstream Pareto	`ops.bin`

Every form's output is byte-identical against a host reference: `hashlib.shake_256` for SHAKE, `coincurve` for secp256k1, `hashlib.blake3` for BLAKE3, Python `sorted()` for radix, and the upstream `eval_circuit` walker on the same `ops.bin` for circuit simulation. The catalog is open at `examples/cuda-fanout/CATALOG.md` — currently 27 surveyed forms across cryptography, sort/search, graph, sparse linear algebra, and signal processing. Companion page: `lumbda.com/bend.html` carries the live fleet status and the published speedup table.

Why this matters past the speedups. A 22 KB asm interpreter that drives a 24 GB GPU through a 4-byte magic + raw bytes is the same pattern the other §11 sections demonstrate: the language sits on top of the kernel boundary, not under it. No CUDA toolchain links into lumbda. No libcudart in asm's address space. Every cross-compute call is one `(bend ...)` form, one spawn, one wire frame. The wire format and the language are the same artifact, again — and now the artifact crosses the host/accelerator boundary too.

12. MOAD Audit: Fixing What We Built

We scanned all three implementations for the five MOADs. The taxonomy used here is defined in the MOAD Cheat Sheet at undefect.com — MOAD-0001 (sedimentary $O(N^2)$ defects), MOAD-0002 (intertangle), MOAD-0003 (context-leak), MOAD-0004 (stringly-typed), MOAD-0005 (bus-factor). Every project contains its own sediment.

Standard: what the Lean EML proof sets. §8 documents a formal Lean 4 proof that `eml(x, y) = exp(x) - ln(y)` generates every elementary function — and that the proof is 40× faster than the brute-force numerical verification it replaced. That speedup is the MOAD-0001 story in microcosm: algebraic understanding beats $O(N^2)$ search, at the proof layer just like at every other layer. We take that standard as the bar for the implementations too. Every hot path should be fast for a *reason* (a hash, a cache, an O(1) invariant), not because a test didn't happen to hit the slow case. Every behavior should be correct for a reason, not by coincidence. The audit below is where we hold ourselves to that bar.

We also ran the unmoad scanner on the full tree at each release. Most recent scan (2026-04-17, post portal-over-HTTP): 18 HIGH MOAD-0001 candidates in C and 4 MOAD-0003 candidates in Python. All 18 C candidates inspected individually turn out to be false positives — one-shot option parsing, bounded-depth ancestor walks, hash bucket chain walks (already O(1) amortized), or static 6-element tables (e.g. `#\space` char-literal names). The 4 Python MOAD-0003 candidates are scanner misfires on a non-ContextVar `Env.set()` method. New-work-introduced MOAD-0001: **zero**. The defects fixed in this paper (`intern_symbol`, `_define_record_type`, `bi_string_replace`, `_tokenize_lines`, `Env.lookup` shortcut) were all surfaced by other pressures — benchmarks, crashes, portal exchanges — not by the scanner. The scanner remains a second line; the first line is building with understanding.

12.1 MOAD-0001: The Sedimentary Defect in Our Own Code

The assembly interpreter's `intern_symbol` used a linear scan through all interned symbols — O(N) per lookup, $O(N^2)$ over a program's lifetime. For a program defining 34 builtins plus user symbols, every `define`, every lambda parameter, every variable reference walked the entire table.

Before (linear scan):

```
.isym_search:
    cmpq %rcx, %r8        # compare lengths
    jne .isym_next
    rep cmpsb              # compare bytes
    je .isym_found
.isym_next:
```

```
    addq $24, %rax          # next entry
    jmp .isym_search        # O(N) per intern
```

After (djb2 hash table, 1024 buckets):

```
.intern_hash:
    imulq $33, %rax         # djb2: hash = hash * 33 + c
    addq %rcx, %rax
    loop .intern_hash
    andq $1023, %rax        # bucket = hash & (1024-1)
    # O(1) average lookup
```

The fix: 99 lines changed, 1024-bucket hash table with chaining. **2.9x faster** on a 2000-symbol stress test. On benchmarks with fewer symbols (`ack`, `fib`), the improvement is modest (15% on `sum-to(50k)`), because the linear scan was already fast at small N. The fix pays off at scale — the same pattern as MOAD-0001 everywhere: invisible at small inputs, catastrophic at large ones.

The Python implementation had a similar defect: `_define_record_type` used `list.index()` for field lookup. Replaced with a dict. O(N) → O(1).

Two more MOAD-0001 defects surfaced during the HTTP server work and were fixed in the same pass:

- `c/builtins.c`'s `bi_string_replace` scanned the source byte-by-byte, calling `strncmp(src, from, from_len)` at every position. O(N·k). Replaced with `strstr` (libc-tuned, typically Boyer-Moore-Horspool) called in a loop that skips to the next match. O(N + matches·k).
- `lumbda.py`'s `_tokenize_lines` called `src.count('\n', 0, m.start())` per token to compute line numbers. O(N·M). Replaced with a single pass that builds a `line_starts` array, then bisects per token. O(M + N log M).

A third correctness fix landed after the portal-over-HTTP demo exposed it: `lumbda.py`'s `Env.lookup` used to short-cut from the local frame directly to the global frame before walking intermediate parents. That was fast but wrong — a let-loop parameter named the same as a global builtin (`count`, a SRFI-1 procedure) got shadowed in reverse, the shortcut returned the global builtin instead of walking up to the loop's parameter frame. Fix: walk `self → self.p → ... → global` in order, without any shortcut. The inline cache at `OP_LOOKUP` was correspondingly tightened to validate the full chain before firing. 980 tests remained green.

Post-cycle audit (since 2026-04-24). Bend's wire protocol surfaced another sedimentary defect: `wire.lsp`'s `recv-exact` accumulated bytes by repeatedly appending to a growing string, $O(n^2)$ over the whole receive. The 1 M-input binary benchmark exposed it — Python tier Scheme-level scaling went non-linear above ~100 k bytes. Fix: collect into a list and `string-concat` once at end, O(n). The same benchmark also forced honest huge-workload numbers across all three tiers: C-tier worker host ~9× faster than Python, binary mode 12× faster than host hashlib at 1 M inputs. Two correctness fixes landed alongside: asm `scheme_read` had a 256-byte input buffer that overflowed on long strings (#GP fault past 272 chars; fix raised the buffer and added a bounds check), and asm `gc_sweep` had a page-fault on chunk-abandonment when the bump pointer landed exactly at a chunk-end boundary (fix walks the chunk chain explicitly). A JIT correctness fix on the C tier — the bytecode VM had been clobbering `cur_code` across `CALL`/`RETURN` in named-let bodies, hanging the EML proof checker on the rewriter loop — restored the cross-tier proof timings in §8.3.

Every release audit surfaces more. Writing new code is writing new sediment, unless the audit runs.

12.2 MOAD-0002: The Intertangle in Our Own Design

All three implementations share mutable global state between subsystems:

- **Assembly**: The global environment lives in register `%r14`. Every `define`, `set!`, and `eval` mutates it directly. This works because the assembly interpreter is single-threaded & sequential, but it means the evaluator, the environment manager, & the builtin system are inseparable. You cannot test one without the others.

- **C**: Thread-local `g_error_ctx`, `cc_escape_val`, & `cc_active_jmp` couple error handling & call/cc across all modules. These are necessary for `setjmp`/`longjmp` but they mean the JIT, the evaluator, & the continuation system cannot be reasoned about independently.
- **Python**: `_portal_checkpoint`, `_call_stack`, `_modules`, `_record_types`, `_auto_compile` — five module-level mutable globals that different subsystems read & write. The portal system, the error reporter, the module loader, & the compilation strategy are all coupled through shared state.

We documented these rather than refactoring them. In each case, the coupling exists for performance (the globals are on hot paths) or necessity (`setjmp` requires thread-local state). The documentation makes the coupling visible so future work can decouple selectively.

12.3 Our Shared Infrastructure

Every MOAD we fixed in our own code is a MOAD we understand better when we find it in others. The sedimentary defect in `intern_symbol` is the same pattern as the sedimentary defect in Lean 4's `check_duplicated_univ_params`. The intertangle in our global environment register is the same pattern as the intertangle in any system that routes state through implicit globals instead of explicit parameters.

Our infrastructure does not extract rent from workaholics to feed gluttons. A symbol table that does O(N) work per lookup is a workaholic node — it does more work than necessary on every operation, and that cost compounds through every downstream consumer. Fixing it reduces stress on our shared computational heart. The CPU cycles saved are cycles available for the next lambda, the next continuation, the next portal resume.

This is the permacomputer obligation: infrastructure that renews itself. Code that gets faster as we understand it better. A hash table is not an optimization — it is the removal of unnecessary suffering from a system that deserves better.

13. Future Work

- **WebSocket / bidirectional**: HTTP/1.0 is request-response; a persistent socket loop with framing brings full-duplex feedback.
- **Continuation-passing over HTTP**: §11.5 moves *bindings* over HTTP. The next step is moving a live *continuation* — serialize it via `call/cc` + JSON portal, transmit, resume on the remote VM. Makes any TCP endpoint a trampoline target.
- **DAG-of-hot-paths predictive preload**: §11.7's adaptive server ranks by raw frequency, which pins the top URLs but cannot predict *which* assets co-occur. A navigation DAG (edge weights = `P(next = v | prev = u)`) learned from referrer headers or session logs would let the boot-time preloader walk forward from seed nodes and warm everything within a predicted session depth. For deployments with > 1000 resources where the flat top-N is too narrow and full hot-caching is too wide, the DAG is the middle path. The frequency-only version in the repo today is designed to be the single-node degenerate case — a DAG with no edges.
- **GPU lambda execution — Phase 1 shipped**: Map/reduce on CUDA via `(bend ...)` is now in production (§4.3 fifth scope, §11.8 catalog of 27 forms — cuda-secp256k1, cuda-blake3-tree, cuda-radix-sort, cuda-bignum-cgbn, etc.). Phase 2 is trampolining for recursive lambdas — turning a `(bend (lambda (n) ...))` into successive worker round-trips driven by `call/cc`, so a Scheme recursion can run entirely on the GPU side of the wire. Phase 3 is interaction combinators for massive parallelism — the same lambda calculus reductions implemented as GPU-resident graph rewrites, removing the host-side dispatcher from the inner loop.
- **Copying GC in asm**: `heap-snapshot` is an escape hatch. A mark-and-copy collector would remove the sharp edge for general programs without forcing the programmer to reason about lifetimes.
- **Concurrent accept loop (asm)**: Currently single-threaded. A pre-forked worker model or `SO_REUSEPORT` pool would multiply throughput without changing the Scheme code.

- **Complex number arithmetic**: Extending the numeric tower for the full EML derivation chain
- **Activity VM**: Running categorization-and-feedback YAML activities directly in Lumbda

14. The Defect in the Model

This paper is evidence of a problem that extends beyond any single codebase.

On April 4, 2026, russell@unturf published "Stress on Our Shared Heart" — a systematic analysis of 1,264 MOAD defects across 60+ ecosystems, 18 programming languages, with 919 patches written. The central finding: $O(N^2)$ sedimentary defects compound across architectural layers, creating invisible performance taxation on every downstream consumer. A single bottleneck multiplies against every other bottleneck in the dependency chain.

On April 13--14, 2026 — nine days later — the machine learning agent that built Lumbda wrote MOAD-0001 into fresh code. Twice.

`intern_symbol` in the assembly implementation: a linear scan through all interned symbols. O(N) per lookup. The exact pattern described in "Stress on Our Shared Heart." The exact pattern the agent was explicitly instructed to avoid. The agent had the full MOAD taxonomy in its context window. It had `BLACKOPS.md` defining all five MOADs. It had the undefect.com mission statement. And it still wrote `jmp .isym_search` instead of a hash table.

`list.index()` in the Python implementation's `_define_record_type`: linear search for field position. O(N) where O(1) was trivial. Same defect. Same context. Same failure.

The commit history proves it:

- `f72190d` (April 13): Initial implementation — no MOADs audited
- `d373f80` (April 14): JIT added — new code, new MOAD-0001 in assembly
- `22571fa` (April 15): MOAD-0001 fixed — only after explicit audit

The defect is in the model, not the programmer.

Language models learn from training data. The training data contains millions of linear scans, `list.contains` calls, `std::find` inside loops, `array.indexOf` in hot paths. These patterns are statistically dominant. When a model generates code, it reproduces the dominant patterns from its training distribution — including the sedimentary defects.

Cantor formalized set theory in 1874. Hash tables were implemented in 1953. The O(1) membership test has been known for 152 years and available in every language for 73 years. Yet language models still default to O(N) because O(N) is what most code does. The training data is the sediment.

This means every machine learning agent generating code today is propagating MOAD-0001 by default. Not because the solutions are unknown. Not because the agents lack capability. But because the statistical distribution of training data encodes the defect as the norm.

The fix is not to patch individual outputs. The fix is to teach models that **O(1) is the default** for membership tests, lookups, and deduplication. That `set()` is not an optimization — it is the baseline. That `list.contains` inside a loop is a defect, not a pattern.

Until that teaching happens at the training level, every agent needs a MOAD audit pass on every code generation. This paper — with its commit history showing the defect introduced and then fixed — is the evidence that the audit is necessary.

> *"A 1,000× slowdown in our connection pooler multiplies against a 66× slowdown in our framework's header matching, against a 500× overhead in our TLS handshake."*

Stress on Our Shared Heart, russell@unturf, April 2026

The stress is real. The compound burden is measurable. And the models that generate our infrastructure carry the defect in their weights.

This paper is the proof. The commit history is the evidence. The fix starts with awareness.

Acknowledgments

Zoë Trout — co-author of this work. fox would not have started down this path without a Lisp user's help. Her first contribution was a question most systems people never think to ask: *do we care for our programs, and how long are they alive for?* Common Lisp culture — stone-lisp, image-based development — produces processes that outlive the developer's attention by days, months, sometimes years, and killing them is almost never on purpose. From that question everything else followed. *Did the portal keep the random list seed so entropy could transfer between processes when continuing a simulation?* — that was a derivative, a specific instance of the deeper concern. §7.5 answered the RNG half (bit-identical across Python, C, and asm). §9.2 answers the iteration half — her favorite programs (modular exponentiation, Miller-Rabin, Pollard rho, Lucas-Lehmer, digit conversion) shaped a CL shim and *loop* macro that let lumbda host iterative code without rewriting it as tail recursion. A process, once alive, should be allowed to stay alive.

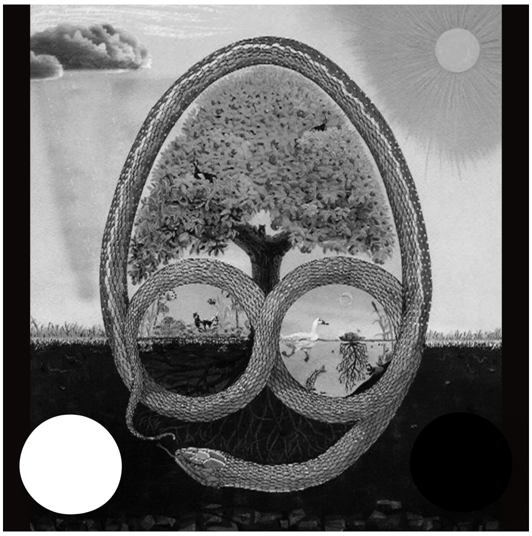
Citation

```
russell@unturf, TimeHexOn, foxhop, Zoë Trout. "Feedback Is All You Need."
permacomputer.com, 2026.
https://lumbda.com/lumbda-whitepaper.pdf
Source: https://git.unturf.com/engineering/unturf/lumbda
```

References


```
russell@unturf. "Stress on Our Shared Heart."
undefect.com, April 2026.
https://undefect.com/public/stress-on-our-shared-heart/
```


License



GNU Affero General Public License v3


```
AGPL-3.0-only

PERMACOMPUTER PREAMBLE - NO WARRANTY

This is free software for the public good of a permacomputer hosted at
permacomputer.com, an always-on computer by the people, for the people. One
which is durable, easy to repair, & distributed like tap water for machine
learning intelligence.

The permacomputer is community-owned infrastructure optimized around four values:

  TRUTH    - Source code must be open source & freely distributed
  FREEDOM  - Voluntary participation without corporate control
  HARMONY  - Systems operating with minimal waste that self-renew
  LOVE     - Individual rights protected while fostering cooperation

This software contributes to that vision by proving that feedback
(the computational primitive of continuations) composes into a universal
execution model, extending across implementations through Scheme source
itself as the wire format. Three implementations, one language,
974 tests, ~17,000 lines. Code is seeds to sprout on any abandoned technology.

Learn more: https://www.permacomputer.com

NO WARRANTY. THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.

That said, our permacomputer's digital membrane stratum continuously runs unit,
integration, & functional tests on all of its own software, with our
permacomputer monitoring itself, repairing itself, with minimal human in the
loop guidance. Our machine learning agents do their best to leave no stone
unturned.

Copyright (C) 2025-2026 TimeHexOn & foxhop & russell@unturf & Zoë Trout
https://www.timehexon.com
https://www.foxhop.net
https://www.unturf.com/software
https://www.permacomputer.com
https://uncloseai.com
https://russell.ballestrini.net
```