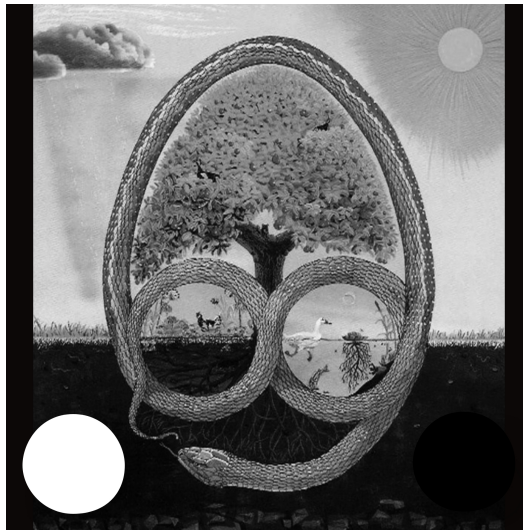




## Feedback Is All You Need

### A Complete Scheme VM in One File: Continuations, Bytecode, & the EML Universality Proof

*How an explicit frame stack with full call/cc builds a production Lisp from feedback alone.*

*russell@unturf, TimeHexOn, foxhop*

[uncloseai.com](https://uncloseai.com) · [permacomputer.com](https://permacomputer.com)

*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.

A programming language needs one primitive to become universal: feedback. A function that receives its own continuation can loop, branch, yield, checkpoint, resume, & migrate. Every control flow pattern reduces to a continuation captured & invoked.

**uncommonlisp** proves this by implementing a complete Scheme in a single Python file (3,261 lines, zero external dependencies beyond the standard library). A stack-based bytecode compiler achieves 7--19x speedups over tree-walking interpretation. An explicit frame stack replaces Python's call stack, enabling tail-call optimization of arbitrary depth & full first-class continuations (`call/cc`) that support escape, upward, & multi-shot invocation. A peephole optimizer, inline cache, & constant folder tighten the generated bytecode. A portal system serializes the entire machine state (environment, continuation stack, instruction pointer) to JSON & resumes it on another machine.

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

Feedback is the primitive. Continuations are the mechanism. One file is the proof.

## 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. `uncommonlisp` 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

`uncommonlisp` implements two evaluation strategies in a single 3,261-line Python file:

**Tree-walking interpreter** (`level`): 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.

### 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. `uncommonlisp` 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 `uncommonlisp` 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 to JSON, feeds the machine state to storage
- **Migration:** portal deserializes on another machine, feeds the continuation back to a new VM

One primitive. Every pattern.

## 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

All benchmarks run on the same machine, same Python 3 interpreter. Three columns: tree-walking interpreter (leval), bytecode VM (--fast), & equivalent CPython. Times are best-of-3 in milliseconds.

### 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 uncommonlisp 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 uncommonlisp's bytecode VM is itself written in Python. The comparison establishes that uncommonlisp 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 uncommonlisp vs CPython (different languages), but uncommonlisp interpreter vs uncommonlisp 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

## 7. Portal: Machine State Serialization

Portal serializes the complete VM state (environment, continuation stack, instruction pointer, value stack) to JSON & resumes execution on another machine.

### 7.1 Serialization

The `_PortalSerializer` performs graph-aware serialization:

- Tracks `id(obj)` → `ref_id` for object identity (handles shared references & 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)

### 7.2 Deserialization

The `_PortalDeserializer` performs two-pass reconstruction:

1. **Shell pass**: Create empty object shells & assign reference IDs
2. **Fill pass**: Populate pointers, values, & environment chains

This handles circular references (e.g., a closure whose environment contains a reference to the closure itself).

### 7.3 Checkpoint & Resume

During VM execution, `portal-checkpoint!` triggers at `OP_JUMP` & `OP_TAIL_CALL` instructions. When triggered:

1. Capture the current continuation (same mechanism as `call/cc`)
2. Serialize the continuation + environment to a `.portal` file
3. Continue execution (or halt, depending on the use case)

Resumption:

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

The VM deserializes the saved state, reconstructs the continuation, & resumes execution from the exact instruction where the checkpoint occurred. The computation does not need to restart from the beginning.

## 7.4 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

uncommonlisp ships with a mathematical proof that a single operator generates all elementary functions:  $\text{eml}(x, y) = \exp(x) - \ln(y)$ .

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 Stage 1: Core Functions (Depth 1--3)

|           |                                                    |                                                            |
|-----------|----------------------------------------------------|------------------------------------------------------------|
| $\exp(x)$ | $= \text{eml}(x, 1)$                               | $; \ln(1) = 0, \text{ so } \text{eml}(x, 1) = \exp(x) - 0$ |
| $e$       | $= \text{eml}(1, 1)$                               | $; \exp(1) = e$                                            |
| $\ln(x)$  | $= \text{eml}(1, \text{eml}(\text{eml}(1, x), 1))$ | $; \text{nested application recovers } \ln$                |

**Proof of ln recovery:** Let  $a = \text{eml}(1, x) = e - \ln(x)$ . Then  $\text{eml}(a, 1) = \exp(e - \ln(x)) = \exp(e)/x$ . Then  $\text{eml}(1, \exp(e)/x) = e - \ln(\exp(e)/x) = e - e + \ln(x) = \ln(x)$ .

### 8.3 Stage 2: Arithmetic

|                  |                                                             |                                          |
|------------------|-------------------------------------------------------------|------------------------------------------|
| 0                | $= \ln(1) = \text{eml}(1, \text{eml}(\text{eml}(1, 1), 1))$ |                                          |
| $a - b$          | $= \text{eml}(\ln(a), \exp(b))$                             | $; \exp(\ln(a)) - \ln(\exp(b)) = a - b$  |
| -1               | $= (e-1) - e$                                               | $; \text{via eml subtraction chain}$     |
| $a * b$          | $= \exp(\ln(a) + \ln(b))$                                   | $; \text{multiplication from exp \& ln}$ |
| $1/x$            | $= \exp(-\ln(x))$                                           | $; \text{division from exp \& ln}$       |
| $x^y$            | $= \exp(y * \ln(x))$                                        | $; \text{exponentiation}$                |
| $\text{sqrt}(x)$ | $= \exp(\ln(x) / 2)$                                        | $; \text{roots}$                         |

### 8.4 Stage 3: Complex Plane Access

```
ln(-1) = iπ ; standard complex logarithm
π = imag(ln(-1))
i = exp(iπ/2)
```

The key insight:  $\ln$  of a negative number enters the complex plane. Since we can construct -1 from eml via the subtraction chain,  $\ln(-1)$  yields  $i\pi$ , from which  $\pi$  &  $i$  follow.



## 8.5 Stage 4: Trigonometry via Euler

```
sin(x) = (exp(ix) - exp(-ix)) / 2i      ; Euler's formula
cos(x) = (exp(ix) + exp(-ix)) / 2
tan(x) = sin(x) / cos(x)
```

All trigonometric functions follow from complex exponentials, which follow from `exp`, which follows from `eml`.

## 8.6 Verification & Friction Analysis

The proof is verified at three levels, with dramatically different friction:

| Approach                 | Time  | Guarantee                 | Friction                          |
|--------------------------|-------|---------------------------|-----------------------------------|
| Python (numerical)       | 0.04s | 1e-10 tolerance           | Low: evaluate & compare           |
| uncommonlisp (numerical) | 59s   | 1e-10 tolerance           | High: $O(N^2)$ brute-force search |
| Lean 4 (formal proof)    | 1.5s  | kernel-verified certainty | Medium: 5 rewrites, type-check    |

**The formal proof is 40x faster than the brute-force search & provides mathematical certainty instead of floating-point tolerance.** This is MOAD-0001 (the sedimentary defect) at the proof methodology layer:  $O(N^2)$  search friction where  $O(1)$  algebraic reasoning suffices.

The numerical approaches enumerate all pairwise EML compositions at each depth, comparing results against target functions. This grows quadratically with the number of known values. The Lean proof does 5 algebraic rewrites, each applying an axiom ( $\exp(\ln(x)) = x$ ,  $\ln(\exp(x)) = x$ ,  $\ln(1) = 0$ ). The type checker confirms each rewrite in microseconds. No search. No tolerance. No conjecture dependency.

**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. Proof assistants eliminate the quadratic friction of verification. They are the hash set to numerical analysis's nested loop.

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  $\leq 4$  confirming that `eml` compositions reach the expected targets.
2. **Self-hosted verification** (`proof/eml_proof.lsp`): The same proof runs in `uncommonlisp`'s bytecode VM (`python3 uncommonlisp.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.

## 9. Language Coverage

`uncommonlisp` implements a near-complete R7RS-small Scheme:

**Special forms** (32): `define`, `set!`, `lambda`,  $\lambda$ , `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:** 529 tests covering lexing, parsing, special forms, bytecode compilation, macros (hygienic & procedural), continuations, generators, record types, modules, arithmetic, higher-order functions, error handling, & portal serialization.

## 10. Relationship to Companion Papers

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

| Layer       | Paper                                         | Role                                                    |
|-------------|-----------------------------------------------|---------------------------------------------------------|
| Runtime     | <b>Feedback Is All You Need</b> (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                      |

uncommonlisp 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 uncommonlisp's VM, with call/cc providing the state machine transitions & portal providing persistence.

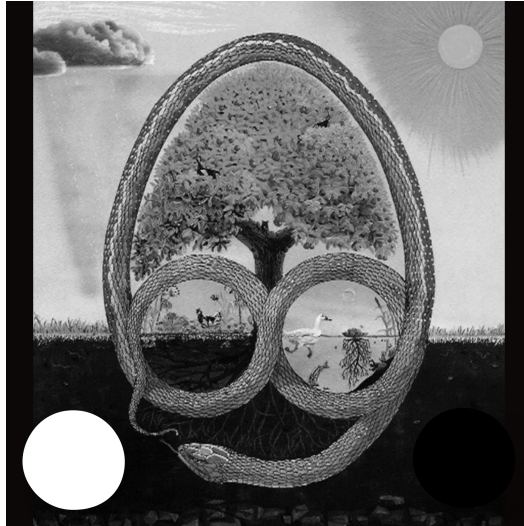
## 11. Future Work

- **Complex number arithmetic:** Extending uncommonlisp's numeric tower to support complex numbers natively, enabling the full EML derivation chain to execute within the VM
- **Distributed continuation passing:** Portal files served over HTTP, enabling a network of permacomputer nodes to pass continuations as messages
- **JIT compilation:** Translating hot bytecode sequences to Python bytecode or native code via ctypes
- **Activity VM:** Running categorization-and-feedback YAML activities directly in uncommonlisp, with call/cc replacing the explicit state machine

## Citation

```
russell@unturf, TimeHex0n, foxhop. "Feedback Is All You Need."  
permacomputer.com, 2026.  
https://git.unturf.com/books/feedback-is-all-you-need
```

## License



*GNU Affero General Public License v3*

AGPL-3.0-only