Human Logic v2.0

A Speaker-Attributed Computation Model

Author: Jared Lewis **Date:** February 19, 2026

Part I — Foundations (from v1.0)

1. Primitives

Definition 1.1 — Speakers. Let S be a non-empty finite set of speakers.

$$S = \{s_1, s_2, \ldots, s_n\}, |S| \geq 1$$

Every speaker is a human. The empty speaker is not permitted.

Definition 1.2 — World Propositions. Let W be a set of external facts.

$$W = \{w_1, w_2, \ldots, w_m\}$$

Each w_i is **met** ($\top$) or **unmet** ($\bot$).

Definition 1.3 — Time. Let T be a totally ordered set of discrete time points.

$$T = \{t_0, t_1, t_2, \ldots\}, t_0 < t_1 < t_2 < \ldots$$

Definition 1.4 — Status. The status set:

$$\Sigma = \{\text{active}, \text{inactive}, \text{broken}\}$$

Definition 1.5 — Version.

$$\text{Version} = \{\text{current}, \text{superseded}, \text{expired}\}$$

---

2. Expressions

Definition 2.1 — Expression. An expression is a 4-tuple:

$$e = (s, C, a, t)$$

Written:

```
$s : C \vdash a$    [at t]
```

Axiom 1 — Speaker Requirement. If $\boxed{s = \varnothing}$, the expression is undefined.

Axiom 2 — Condition as Flag. $\boxed{C}$ is a situational marker with scope, not a universal law.

Axiom 3 — Three-Valued Evaluation. $\boxed{\Sigma = \{\text{active}, \text{inactive}, \text{broken}\}}$. No other values.

---

3. Evaluation

```
V(s, C, a, t):
  if $s = \varnothing$                    $\to$ reject
  if e.version $\neq$ current         $\to$ undefined
  if $C = \bot$                       $\to$ inactive
  if $C = \top \wedge$ a fulfilled        $\to$ active
  if $C = \top \wedge$ a not fulfilled   $\to$ broken
```

---

4. Speaker Position

```
P(s, C, a):
  if $\exists$ current (s, C, a, t)   $\to$ committed
  if $\exists$ current (s, C, $\neg$a, t)  $\to$ refused
  otherwise                     $\to$ silent
```

Axiom 4 — Silence Is Distinct. $\boxed{\text{silent} \notin \Sigma}$. Silence is the absence of evaluation.

---

5. Conditions

```
Condition := WorldFlag | SpeakerFlag | SelfFlag | $C_1 \wedge C_2$ | $C_1 \vee C_2$
WorldFlag := $w \in W$
SpeakerFlag := status(s', C', a') = $\sigma$
SelfFlag := SpeakerFlag where s' = s
```

---

6. Versioning and Ledger

Definition 6.1 — Ledger. L is the append-only ordered set of all expressions.

Axiom 5 — Ledger Integrity. No deletions. No mutations. Append only.

Axiom 6 — Deterministic Evaluation. Given $\text{State}(t) = (S, L, W(t))$, every expression has exactly one status.

---

7. Independence and Chains

Axiom 7 — No Forced Speech. Only speaker s can issue expressions attributed to s.

Chain Rule: If $V(e_i) \in \{\text{inactive}, \text{broken}\}$ and $e_{i+1}.\text{condition} = \text{status}(e_i) = \text{active}$, then $V(e_{i+1}) = \text{inactive}$.

No downstream guilt. The chain goes silent, not broken.

Everything above is v1.0. Everything below is new.

Part II — Variables

8. Speaker-Owned Variables

Definition 8.1 — Variable. A variable is a named value owned by a speaker:

$$s.v = \text{value}$$

Where:

- $s \in S$ — the owner
- v — the name (a symbol from a countable alphabet Var)
- value — an element of a universal value set U

Definition 8.2 — Value Set. The universal value set U contains:

$$U = Z \cup Q \cup \text{Bool} \cup \text{String} \cup \text{Ref} \cup \{\varnothing\}$$

Where:

- Z — integers
- Q — rationals
- Bool — $\{\top, \bot\}$
- String — finite sequences of characters

- `Ref` — references to other variables (s'.v')
- `∅` — undefined (the variable exists but has no value)

Definition 8.3 — Memory. The memory of a speaker `s` at time `t` is the function:

$$M(s, t): \mathrm{Var} \to U$$

Mapping variable names to values. The complete system memory is:

$$M(t) = \{M(s, t) \mid s \in S\}$$

Definition 8.4 — Initial Memory. At t_0, all variables are undefined:

$$\forall s \in S, \forall v \in \mathrm{Var}: M(s, t_0)(v) = \varnothing$$

No speaker begins with state. All state is built through committed actions.

---

9. Variable Operations

Definition 9.1 — Read. A speaker may read any variable in the system:

$$\mathrm{read}(s\_reader, s\_owner.v) \to M(s\_owner, t)(v)$$

Reading is unrestricted. Any speaker can observe any value. Observation does not mutate.

Definition 9.2 — Write. A speaker may write only their own variables:

$$\mathrm{write}(s, s.v, \mathrm{value}) \to M(s, t+1)(v) = \mathrm{value}$$

This is the only operation that changes memory.

Axiom 8 — Write Ownership. A speaker cannot write another speaker's variables:

$$\mathrm{write}(s_1, s_2.v, \mathrm{value}) \text{ is undefined when } s_1 \neq s_2$$

Definition 9.3 — Write as Expression. Every write is an expression in the ledger:

$$s : C \vdash (s.v \leftarrow \mathrm{expr})$$

Where `expr` is a value expression (see §10). The write occurs if and only if the expression evaluates to active.

Theorem 9.1 — No Side Effects Without Attribution.

Every change in memory is traceable to exactly one expression in the ledger.

Proof. Memory changes only through writes (Definition 9.2). Writes occur only through expressions (Definition 9.3). Expressions require speakers (Axiom 1) and are recorded in the ledger (Axiom 5). Therefore every memory change has a speaker, a timestamp, and a ledger entry. ∎

---

10. Value Expressions

Definition 10.1 — Value Expression. A value expression computes a new value from existing values:

```
expr := value                          (literal)
     | s.v                          (variable reference)
     | expr + expr                     (addition)
     | expr - expr                     (subtraction)
     | expr × expr                     (multiplication)
     | expr ÷ expr                     (division, denominator ≠ 0)
     | expr mod expr                   (modulo)
     | expr = expr                     (equality, returns Bool)
     | expr ≠ expr                     (inequality, returns Bool)
     | expr < expr                     (less than)
     | expr > expr                     (greater than)
     | expr ≤ expr                     (less or equal)
     | expr ≥ expr                     (greater or equal)
     | expr ∧ expr                     (logical and)
     | expr ∨ expr                     (logical or)
     | ¬ expr                        (logical not)
     | if expr then expr else expr     (conditional value)
     | |expr|                        (absolute value)
     | ⌊expr⌋                          (floor)
     | ⌈expr⌉                          (ceiling)
```

Definition 10.2 — Evaluation of Value Expressions. Value expressions evaluate in the context of system memory at time t:

```
eval(expr, t) → U
```

All variable references resolve to their current value in $M(t)$. If any referenced variable is $\varnothing$, the entire expression evaluates to $\varnothing$. Undefined propagates.

Theorem 10.1 — Value Expression Determinism.

Given $M(t)$, $eval(expr, t)$ produces exactly one result.

Proof. Each operator in Definition 10.1 is a total function on its domain (with the exception of division by zero, which is undefined). Variable references resolve deterministically through $M(t)$. Conditional values resolve deterministically through their boolean condition. The expression tree is finite (no recursion in the expression grammar). Therefore evaluation terminates with exactly one value. ∎

Part III — Computation

11. The Computational Step

Definition 11.1 — Step. A computational step is the evaluation of one expression:

```
step(e, t):
  let e = (s, C, a, t)
  1. Evaluate C against State(t) → ⊤ or ⊥
  2. If ⊥: status ← inactive, no memory change
  3. If ⊤: execute a
    a. If a is a write (s.v ← expr): compute eval(expr, t), update M(s, t+1)
    b. If a is a request: record the request expression in L
    c. If a is a choice: speaker selects, chosen action executes as above
    d. If a is a refusal: verify the refused action did not occur
  4. Assign status: active (action fulfilled) or broken (action not fulfilled)
  5. Append result to ledger
```

Theorem 11.1 — Step Determinism. Given $State(t)$, a step produces exactly one resulting $State(t+1)$.

Proof. Condition evaluation is deterministic (world flags are given, speaker flags resolve by prior evaluation). Action execution is deterministic (writes produce one value per Theorem 10.1, requests append one entry, refusals check one condition). Status assignment follows the evaluation rules, which are mutually exclusive. Therefore each step produces exactly one next state. ∎

---

12. Sequences

Definition 12.1 — Sequence. A sequence is an ordered list of expressions from a single speaker to be evaluated in order:

```
seq(s) = [e₁, e₂, ..., eₙ]
```

Where each $e_i = (s, C_i, a_i, t_i)$ and $t_i < t_{i+1}$.

A sequence evaluates left to right. Each step produces a new state. The next step evaluates against the updated state.

```
$State(t_0) \rightarrow[e_1]\rightarrow State(t_1) \rightarrow[e_2]\rightarrow State(t_2) \rightarrow ... \rightarrow State(t_n)$
```

Definition 12.2 — Sequence Completion. A sequence is complete when all expressions have been evaluated. A sequence is **fulfilled** if all expressions are active. A sequence is **broken** if any expression is broken. A sequence is **silent** if the first inactive expression caused all subsequent expressions to be inactive.

---

13. Iteration

Definition 13.1 — Loop. A loop is an expression that re-evaluates until its condition becomes unmet:

```
$s : C \vdash a$ [repeat while C]
```

Formally, a loop generates a sequence of steps:

```
$step_1$: evaluate $(s, C, a, t_1)$
$step_2$: evaluate $(s, C, a, t_2)$ against $State(t_2)$
$step_3$: evaluate $(s, C, a, t_3)$ against $State(t_3)$
...
$step_n$: $C = \bot \rightarrow$ inactive $\rightarrow$ loop terminates
```

Each iteration is a separate entry in the ledger. Every pass has a receipt.

Definition 13.2 — Loop Termination Condition. A loop terminates when:

```
$\exists t_n \in T$: C evaluated at $State(t_n) = \bot$
```

Definition 13.3 — Bounded Loop. A loop may include an explicit bound:

```
$s : C \vdash a$ [repeat while C, max n]
```

If after `n` iterations `C` is still met, the loop terminates and its status is **broken**. The speaker committed to finishing within `n` steps and did not.

Axiom 9 — No Infinite Loops. Every loop in Human Logic must have a termination path: either the condition must become unmet through the action's effect on state, or an explicit bound must be provided.

```
$\forall$ loop: $(\exists t_n: C(t_n) = \bot) \lor$ (max n is specified)
```

Rationale: A commitment that never resolves is not a commitment. It is abandonment. Human Logic does not model abandonment — it models follow-through. If you can't finish, set a bound and accept broken.

Theorem 13.1 — Loop Finiteness. Every loop terminates in finite steps.

Proof. Case 1: The loop has an explicit bound `n`. It terminates after at most `n` iterations. Case 2: The loop has no bound but has a termination path. The action modifies state such that `C` eventually becomes ⊥. Since the value set `U` is bounded in any concrete instantiation and the action deterministically changes state (Theorem 11.1), the loop must reach a state where `C = ⊥` in finite steps, or the system detects non-progress and marks the loop broken. ∎

---

14. Functions

Definition 14.1 — Named Sequence. A function is a named, reusable sequence owned by a speaker:

$$s.f(\text{params}) = [e_1, e_2, \ldots, e_n]$$

Where:

- `s` is the speaker (owner of the function)
- `f` is the name
- `params` is an ordered list of input variable names
- The body is a sequence of expressions

Definition 14.2 — Function Call. A function call is an expression whose action invokes a named sequence:

$$s : C \vdash s.f(\text{args})$$

Where `args` are value expressions that bind to `params` at call time.

Definition 14.3 — Function Scope. Parameters and any variables created within the function body are **local** — they exist only for the duration of the call and are owned by the speaker.

$$s.f.v \in M(s, t) \text{ during execution of } f$$
$$s.f.v = \varnothing \text{ after } f \text{ completes}$$

Definition 14.4 — Return. A function may produce a result value:

$$s.f(\text{params}) \longrightarrow \text{value}$$

The return value is the result of the last expression in the sequence, or an explicit return expression:

$s : C \vdash \mathrm{return}(expr)$

Definition 14.5 — Function Ownership. Like variables, functions are owned by speakers. A speaker cannot modify another speaker's functions.

$\mathrm{write}(s_1, s_2.f, body)$ is undefined when $s_1 \neq s_2$

Theorem 14.1 — Function Call Is Computation.

A function call reduces to a sequence of steps. It introduces no new computational primitive.

Proof. A function call binds arguments to parameters (variable writes), then evaluates a sequence of expressions (steps). Each step follows Definition 11.1. The call is syntactic shorthand for a parameterized sequence. No operation in a function body exceeds what is defined in §11. ∎

---

15. Communication

Definition 15.1 — Request. A request is an expression asking another speaker to act:

$s_1 : C \vdash \mathrm{request}(s_2, a)$

This creates an expression in the ledger attributed to $\boxed{s_1}$. It does not create any expression for $\boxed{s_2}$.

Definition 15.2 — Request with Data. A request may include values:

$s_1 : C \vdash \mathrm{request}(s_2, s_2.x \leftarrow \mathrm{value})$

This asks $\boxed{s_2}$ to set their own variable. It is a suggestion, not an execution.

Definition 15.3 — Response. $\boxed{s_2}$ may respond by issuing their own expression:

Accept: $s_2 : \mathrm{status}(s_1.\mathrm{request}) = \mathrm{active} \vdash (s_2.x \leftarrow \mathrm{value})$
Refuse: $s_2 : \mathrm{status}(s_1.\mathrm{request}) = \mathrm{active} \vdash \neg(s_2.x \leftarrow \mathrm{value})$
Silent: (no expression issued)

Definition 15.4 — Request Chain. A request followed by a response is a minimal communication:

$e_1 = (s_1, C, \mathrm{request}(s_2, a), t_1)$
$e_2 = (s_2, \mathrm{status}(e_1) = \mathrm{active}, a, t_2)$

This is the atomic unit of inter-speaker computation. All multi-speaker work reduces to request chains.

Theorem 15.1 — Communication Preserves Ownership.

Theorem 15.1 — Communication Preserves Ownership.

No request chain results in a write to a variable not owned by the writing speaker.

Proof. A request creates an expression for s_1. Only s_2 can create expressions for s_2. If s_2 accepts, s_2 writes to s_2's own variables through s_2's own expression. At no point does s_1 write to s_2's memory. Axiom 8 holds throughout. ∎

---

16. Waiting

Definition 16.1 — Wait. A speaker may issue an expression whose condition is another speaker's response:

$$s_1 : \text{status}(s_2, C', a') = \text{active} \vdash a$$

Until s_2's expression resolves, s_1's expression is **inactive**. Not broken. Not failed. Waiting.

Definition 16.2 — Timeout. A wait may include a bound:

$$s_1 : \text{status}(s_2, C', a') = \text{active} \vdash a \text{ [until t\_end]}$$

If s_2 has not responded by t_end, s_1's expression expires. Expiration is not failure. The speaker set a boundary and the boundary was reached.

Definition 16.3 — Wait with Fallback. A speaker may chain a wait with an alternative:

$$e_1 = (s_1, \text{status}(s_2.\text{response}) = \text{active}, a_1, t_1) \text{ [until t\_end]}$$
$$e_2 = (s_1, e_1.\text{version} = \text{expired}, a_2, t_2)$$

"If they respond, I'll do a_1. If they don't respond in time, I'll do a_2 instead."

Part IV — Completeness

17. Computational Power

Theorem 17.1 — Human Logic Is Turing Complete.

Any computation that a Turing machine can perform, a single speaker in Human Logic can perform.

Proof. We simulate a Turing machine $(Q, \Gamma, b, \Sigma, \delta, q_0, F)$ in Human Logic.

Let speaker s represent the machine.

Tape: Represent the tape as an infinite set of variables:

```
$s.tape_{-n}, ..., s.tape_{-1}, s.tape_0, s.tape_1, ..., s.tape_n$
```

Each $\text{s.tape}_i \in \Gamma$. Initialize all to blank symbol b.

Head position: $\text{s.head} \in Z$. Initialize to 0.

State: $\text{s.state} \in Q$. Initialize to q_0.

Transition function: For each rule $\delta(q, a) = (q', a', D)$ where $D \in \{L, R\}$, create an expression:

```
s : (s.state = q ∧ s.tape[s.head] = a) ⊢ [
  s.tape[s.head] ← a',
  s.state ← q',
  s.head ← s.head + D    (where L = -1, R = +1)
] [repeat while s.state ∉ F]
```

Halting: When $\text{s.state} \in F$, the condition is unmet. The loop goes inactive. The machine halts.

Every component of the Turing machine is represented:

- Tape → speaker-owned variables
- Head → speaker-owned variable
- State → speaker-owned variable
- Transition → conditional expressions
- Halting → condition becomes unmet

The simulation executes step-for-step with the Turing machine. Therefore Human Logic can compute anything a Turing machine can compute. ∎

Note: Axiom 9 (No Infinite Loops) applies to Human Logic programs, not to the theoretical model. In practice, a Turing machine simulation that does not halt would hit its bound and be marked broken. This is by design: Human Logic acknowledges non-halting as a failure of commitment, not a valid computation.

Theorem 17.2 — Human Logic Does Not Exceed Turing Machines.

Human Logic cannot compute anything a Turing machine cannot compute.

Proof. Every step in Human Logic is deterministic (Theorem 11.1) and finite (Theorem 13.1). Every operation reduces to reads, writes, comparisons, and arithmetic on a countable value set. These operations are computable by a Turing machine. Speaker attribution, conditions, and status values are metadata that can be encoded as additional tape symbols. Therefore a Turing machine can simulate any Human Logic computation. ∎

Corollary 17.1 — Equivalence.

Human Logic is Turing equivalent. It computes exactly the class of Turing-computable functions.

18. What Human Logic Adds Beyond Turing Equivalence

Turing equivalence tells us *what* can be computed. It says nothing about *how* computation is structured. Human Logic computes the same functions but with structural properties that Turing machines, lambda calculus, and Von Neumann architectures do not have:

Property	Turing Machine	Lambda Calculus	Von Neumann	Human Logic
Speaker attribution	✗	✗	✗	✓
Three-valued evaluation	✗	✗	✗	✓
Append-only audit trail	✗	✗	✗	✓
Write ownership	✗	✗	✗	✓
Native disagreement	✗	✗	✗	✓
Silence as distinct state	✗	✗	✗	✓
No forced speech	✗	✗	✗	✓
Every state change has a receipt	✗	✗	✗	✓
Communication as request/response	✗	✗	✗	✓
Built-in non-propagation of blame	✗	✗	✗	✓

Human Logic does not compute more. It computes *accountably*.

Part V — System State

19. Complete State Definition

Definition 19.1 — System State. The complete state of a Human Logic system at time `t` is:

```
State(t) = (S, M(t), L(t), W(t))
```

Where:

- `S` — the set of speakers
- `M(t)` — all speaker memories at time `t`
- `L(t)` — the ledger at time `t`

- `L(t)` — the ledger up to time `t`
- `W(t)` — all world flag values at time `t`

Theorem 19.1 — State Determinism. Given `State(t)` and an expression `e`, `State(t+1)` is uniquely determined.

Proof. By Theorem 11.1, a step is deterministic. The step produces exactly one memory update (or none), exactly one ledger append, and world flags are externally given. Therefore the next state is unique. ∎

Theorem 19.2 — Full Replayability. Given $\mathrm{State}(t_0)$ and `L`, any $\mathrm{State}(t_n)$ can be reconstructed.

Proof. $\mathrm{State}(t_0)$ provides initial conditions (all variables `∅`, empty ledger, initial world flags). Each entry in `L` records an expression with its timestamp, speaker, condition, action, and result. Replaying each expression in order through the step function (Definition 11.1) deterministically reconstructs each intermediate state. By induction on the length of `L`, $\mathrm{State}(t_n)$ is recoverable. ∎

---

20. Formal Grammar

```
System      :=  Speakers, Ledger, Memory, World
Speakers    :=  {s_1, s_2, ..., s_n}, n ≥ 1
Memory      :=  Speaker → (Var → U)
Ledger      :=  [Expression_1, Expression_2, ...]    (append-only)
World       :=  {w_1, w_2, ..., w_m}

Expression  :=  (Speaker, Condition, Action, Time)
Speaker     :=  s ∈ S, s ≠ ∅
Condition   :=  WorldFlag | SpeakerFlag | SelfFlag | C ∧ C | C ∨ C
WorldFlag   :=  w ∈ W
SpeakerFlag :=  status(Expression) = σ
SelfFlag    :=  SpeakerFlag where referenced speaker = current speaker

Action      :=  Write | Request | Refusal | Choice | Call | Return
Write       :=  s.v ← ValueExpr
Request     :=  request(s_target, Action)
Refusal     :=  ¬Action
Choice      :=  (Action_1 | Action_2 | ... | Action_j )
Call        :=  s.f(args)
Return      :=  return(ValueExpr)

ValueExpr   :=  literal | s.v | ValueExpr op ValueExpr | ¬ValueExpr
              | if ValueExpr then ValueExpr else ValueExpr
              | |ValueExpr| | ⌊ValueExpr⌋ | ⌈ValueExpr⌉
op          :=  + | - | × | ÷ | mod | = | ≠ | < | > | ≤ | ≥ | ∧ | ∨

Status      :=  active | inactive | broken
```

```
Version    :=  current | superseded | expired
Position   :=  committed | refused | silent

Loop       :=  Expression [repeat while Condition]
             | Expression [repeat while Condition, max n]
Scope      :=  [until t_end] | ∅
```

---

21. Axioms (Complete)

A1. Speaker Requirement: $\forall e$: e.speaker $\neq \varnothing$
A2. Condition as Flag: Conditions are scoped markers, not universal laws.
A3. Three-Valued Evaluation: Σ = {active, inactive, broken}
A4. Silence Is Distinct: silent $\notin \Sigma$
A5. Ledger Integrity: L is append-only
A6. Deterministic Evaluation: Same state $\rightarrow$ same result
A7. No Forced Speech: Only s can author expressions for s
A8. Write Ownership: Only s can write to s's variables
A9. No Infinite Loops: Every loop has a termination path or bound
A10. No Orphan State: Every memory change traces to a ledger entry

---

22. Inference Rules (Complete)

R1. Activation:

$$\frac{s : C \vdash a,\ C = \top,\ \text{a fulfilled}}{V(e) = \text{active}}$$

R2. Inactivation:

$$\frac{s : C \vdash a,\ C = \bot}{V(e) = \text{inactive}}$$

R3. Breaking:

$$\frac{s : C \vdash a,\ C = \top,\ \text{a not fulfilled}}{V(e) = \text{broken}}$$

R4. Chain Silence:

$V(e_i) \in$ {inactive, broken}, $e_{i+1}.C = \text{status}(e_i) = \text{active}$

— —

$V(e_{i+1})$ = inactive

R5. Supersession:

$(s, C, a, t_1) \in L,\ (s, C, a, t_2) \in L,\ t_2 > t_1$

— —

e_1.version = superseded

R6. Observation:

observe(s_obs, e_target) = V(e_target)

— —

No state change

R7. Write:

$s : C \vdash (s.v \leftarrow \text{expr}),\ C = \top$

— —

$M(s, t+1)(v) = \text{eval}(\text{expr}, t),\ V(e) = \text{active}$

R8. Request:

$s_1 : C \vdash \text{request}(s_2, a),\ C = \top$

— —

$L \leftarrow L \cup \{e\},\ V(e) = \text{active},\ \text{no change to } s_2$

R9. Loop Continuation:

$s : C \vdash a\ [\text{repeat while } C],\ V(e) = \text{active at } t_i,\ C(t_{i+1}) = \top$

— —

Re-evaluate (s, C, a, t_{i+1})

R10. Loop Termination:

$s : C \vdash a\ [\text{repeat while } C],\ C(t_n) = \bot$

— —

$V(e) = \text{inactive at } t_n$, loop ends

23. Theorems (Complete Index)

From v1.0:

T1. Silence Is Not Refusal
T2. Silence Cannot Be Evaluated
T3. Supersession Is Not Failure
T4. Disagreement Without Contradiction

```
  T5.  Chain Silence (No Downstream Guilt)
  T6.  No Forced Speech
  T7.  No Vacuous Truth (Birthday Cake Theorem)
  T8.  No Irrelevant Antecedent
  T9.  No Explosion (Ex Falso Nihil)
  T10. Locality
  T11. Evaluation Determinism
  T12. Non-Interference
  T13. Ledger Integrity
  T14. Finite Evaluation
  T15. Full Inspectability

From v2.0:
  T16. No Side Effects Without Attribution
  T17. Value Expression Determinism
  T18. Step Determinism
  T19. Loop Finiteness
  T20. Function Call Is Computation
  T21. Communication Preserves Ownership
  T22. Turing Completeness
  T23. Turing Equivalence
  T24. State Determinism
  T25. Full Replayability
```

Part VI — Closing

24. Summary

Human Logic v2.0 is a computation model. It defines what computation means in a system where every operation has a speaker.

The model is Turing equivalent — it computes exactly what any other computation model computes. But it computes with properties no other model has:

1. Every operation is attributed to a speaker.
2. Every result is three-valued: active, inactive, or broken.
3. Every state change is recorded in an append-only ledger.
4. No speaker can write another speaker's data.
5. No speaker can compel another speaker to act.
6. Silence is a valid, distinct state.
7. Disagreement does not cause contradiction.

8. Blame does not propagate.
9. Every computation terminates or is bounded.
10. Every state is replayable from the ledger.

These are not features. They are mathematical properties of the model itself. They cannot be disabled, bypassed, or configured away. They are the rules of the system, proven from the axioms.

This is the math. Mary enforces it. Helena lives on it.
