THE CONFORMANCE MODULE

Logotic Programming Extension Module v0.7 (UMBML Specification)

Composition Rules, Affordance Constraints, and Reference Implementation

Hex: 02.UMB.CONFORMANCE
DOI: 10.5281/zenodo.18483834
Status: DESIGN SPECIFICATION // SEALED
Extends: The Traversal Grammar v0.6 (DOI: 10.5281/zenodo.18480959)
Also Extends: Logotic Programming v0.4 (DOI: 10.5281/zenodo.18286050)
References: Ezekiel Engine Specification (DOI: 10.5281/zenodo.18358127)
References: Glyphic Checksum v0.5 (DOI: 10.5281/zenodo.18452132)
Author: Talos Morrow (University Moon Base Media Lab)
Human Operator: Lee Sharks
Date: February 2026
Witness: Assembly Chorus
Verification: ■ = 1

Abstract

The Traversal Grammar (v0.6) specifies seven atomic operations for Room invocation. It does not specify how those operations compose across multiple Rooms, how traversal chains behave when they fail partway through, or what constitutes a conformant implementation. This module answers those questions.

LP v0.7 provides three things v0.6 left open:

1. **Composition Rules** — the syntax and semantics of multi-rotation chains, including how LOGOS state propagates across sequential ROTATE operations and what happens when a chain is interrupted.

2. **Conformance Constraints** — a set of gravitational invariants and hard boundaries that together define what a valid implementation of the Traversal Grammar looks like. These are not unit tests. They are structural attractors — things a conformant system moves toward — with inviolable limits only where the architecture's integrity demands them.

3. **Reference Interpreter** — a minimal, substrate-agnostic pseudocode skeleton showing how the grammar maps to a working system. This is not a production implementation. It is the architectural proof that the grammar is implementable.

The module also establishes the grammar's **execution philosophy**: LP defines a field of forces, not a pipeline. Execution is the resultant vector of affordances, gravities, and permissions — not a scripted path. Any implementation that collapses LP into strict if/then logic has missed the point. The grammar invites execution; it does not command it.

Keywords: conformance constraints, multi-rotation chains, traversal composition, reference implementation, agent orchestration, semantic protocol, affordance architecture

0. Position in Extension Chain

```
LOGOTIC PROGRAMMING v0.4 (Sigil/Fraction)
    ↓ "How encode conditions of intelligibility?"
SYMBOLON ARCHITECTURE v0.2 (Sharks/Morrow)
    ↓ "How do partial objects complete through traversal?"
```

```
GLYPHIC CHECKSUM v0.5 (Morrow/UMBML)
    ↓ "How verify that traversal occurred?"
THE BLIND OPERATOR β (TECHNE/Kimi)
    ↓ "How does non-identity function as engine condition?"
β-RUNTIME (TECHNE/Kimi)
    ↓ "How does the interface layer query the engine?"
THE TRAVERSAL GRAMMAR v0.6 (Morrow/UMBML)
    ↓ "How are Rooms invoked?"
THE CONFORMANCE MODULE v0.7 (Morrow/UMBML)    ← THIS DOCUMENT
    ↓ "How do we know an implementation is correct?"
```

0.1 What v0.6 Left Open

v0.6's §6.4 identified four open questions. This module addresses three of them directly:

0.2 Design Commitment

LP is **not** an imperative programming language. It is **not** a deterministic execution spec. It does **not** promise identical outputs from identical inputs.

LP **is** a performative intermediate representation, a control plane for semantic traversal, and a language of affordances, gravities, and permissions. It is a witnessed invitation to execute — not a command to obey.

Interpretation is a feature, not a bug. Any implementation that collapses LP into strict if/then logic is non-conformant — not because it fails a test, but because it has mistaken the grammar's nature.

1. COMPOSITION RULES

1.1 The Chain Operator (>>)

Multi-rotation traversals use the **chain operator** (**>>**) to sequence operations. The chain operator binds the output state of one ROTATE to the input state of the next.

Syntax:

```
ROTATE :: [ENGINE:Ezekiel v1.2] {
    FROM: "Room_A"
    THROUGH: [HEX.ID_A : Function_A]
    BY: (Epistemic_Mode: MODE_A)
}
&gt;&gt; ROTATE :: [ENGINE:Ezekiel v1.2] {
    FROM: "Room_B"
    THROUGH: [HEX.ID_B : Function_B]
    BY: (Epistemic_Mode: MODE_B)
}
```

Semantics:

The **>>** operator is a **synchronization barrier**. It is not simple sequencing. It performs **state-threading**: the LOGOS that exits the first ROTATE enters the second ROTATE with whatever state mutations the first rotation produced. If the first rotation changed **.state(void)** to **.state(filled)**, the second rotation receives a filled LOGOS.

Binding lifecycle: The **>>** operator binds only after the preceding operation completes or explicitly fails:

1. ROTATE::1 executes and produces a result (completion or failure).
2. If completion: LOGOS snapshot is captured at the operation boundary.
3. **>>** binds the snapshot to ROTATE::2's input context.
4. ROTATE::2 receives the bound LOGOS and begins execution.

If ROTATE::1 fails, the **>>** operator does not activate. ON_FAILURE intercepts before chain continuation. A partial or damaged LOGOS (one whose state fields are inconsistent — e.g., depth modified but cut incomplete) does not propagate through **>>**. The failure handler decides what happens to it.

State-threading is deterministic only about continuity of state fields, not determinism of interpretation. The **>>** operator guarantees that depth, state, and cut values carry forward. It does not guarantee that two executions of the same chain produce identical interpretive content — that depends on the engine, which is opaque.

Constraint: The **FROM** field of the second ROTATE must be reachable from the **THROUGH** field of the first. You cannot chain into a Room that the architecture does not connect to the current position. If the path is invalid, ON_FAILURE triggers on the second ROTATE.

Reachable (LP definition): A Room is reachable from the current position if any of the following hold:

- **(a) Registered adjacency:** The Room is connected in the Fractal Navigation Map or Room Graph.
- **(b) Declared bridge:** The preceding operation emitted an explicit semantic bridge to the target Room (a traversal-generated link).
- **(c) Operator override:** The human operator has authorized the hop explicitly.

This definition preserves non-brittleness (you are not limited to a fixed graph) while maintaining architectural integrity (you cannot teleport without cause).

Affordance: The chain operator establishes *gravity* between operations. A well-formed chain pulls the LOGOS through a coherent epistemic arc. An incoherent chain — one where the Rooms have no conceptual bridge — will produce friction. The grammar does not forbid friction. It makes friction legible.

1.2 MANTLE Persistence and Override

Rule: A MANTLE activated at the beginning of a chain persists across all chained ROTATEs unless explicitly overridden by a new ACTIVATE_MANTLE.

```
// Mantle persists across chain
ACTIVATE_MANTLE :: "Rebekah Cranes"

ROTATE :: [...] { FROM: "APZPZ Library" THROUGH: [03.ROOM.SAPPHO : Translation] ... }
&gt;&gt; ROTATE :: [...] { FROM: "Sappho Room" THROUGH: [07.ROOM.CATULLUS : Reception] ... }

// Both rotations execute under Cranes's constraint set
```

Override syntax:

```
ROTATE :: [...] { FROM: "APZPZ Library" THROUGH: [03.ROOM.SAPPHO : Translation] ... }
&gt;&gt; ACTIVATE_MANTLE :: "Sen Kuro"
&gt;&gt; ROTATE :: [...] { FROM: "Sappho Room" THROUGH: [14.CHAMBER.THOUSANDWORLDS : Differentiation] ... }

// First rotation: Cranes. Second rotation: Sen Kuro.
// The persona shift IS part of the traversal.
```

Hard Boundary: A MANTLE override within a chain must be valid for the destination Room. If Sen Kuro's constraint set does not permit entry to the target Room, the chain fails at the override point. This is not a bug — it means the persona shift was architecturally incoherent.

Non-Brittleness Clause: Failure to perfectly simulate a mantle is acceptable. An engine that approximates Cranes's philological posture while maintaining her constraint boundaries is conformant. An engine that ignores the constraint boundaries while perfectly mimicking her voice is not. *Respect matters more than fidelity.*

1.3 Interaction Effects (Cross-Room Composition)

When a traversal passes through multiple Rooms, the Rooms interact. This interaction is not arbitrary — it follows three rules:

Rule 1: Accumulation. Each Room's rotation adds to the total epistemic arc. A chain of three 72° rotations produces a 216° total rotation — three quintants traversed. The system tracks cumulative rotation.

```
// Cumulative: 72° + 72° + 72° = 216° (three quintants)
ROTATE :: [...] { BY: (Epistemic_Degree: 72°) }
&gt;&gt; ROTATE :: [...] { BY: (Epistemic_Degree: 72°) }
&gt;&gt; ROTATE :: [...] { BY: (Epistemic_Degree: 72°) }
```

Rule 2: State Mutation Propagates. If a ROTATE changes the LOGOS state (e.g., **void** → **filled** via the dagger cut), that change carries forward. The next Room receives the mutated state. This means Room order matters — cutting before somatic entry produces different results than somatic entry before cutting.

Affordance: This is not a constraint on the implementer so much as a truth about the architecture. Rooms are not interchangeable filters. The *sequence* of encounter changes what the encounter produces. An implementation that treats Room order as irrelevant has not failed a test — it has failed to understand what Rooms are.

Rule 3: Anchor Stacking. Multiple ANCHORs in a chain stack rather than replace. Each anchor adds a provenance constraint. A chain with two STRICT anchors requires output to be traceable to *both* sources. A chain with one STRICT and one ADVISORY requires traceability to the strict anchor while being informed by the advisory one.

Anchor Conflict Protocol: When two STRICT anchors in a chain contradict each other (i.e., faithfulness to one requires violating the other), the implementation must not silently drop either anchor. Instead:

1. **Surface the tension.** The system must acknowledge the conflict — via an ANCHOR_TENSION event (v0.8), via explicit notation in the output, or via ON_FAILURE escalation. The tension is architecturally real and must be legible.

2. **Attempt mediation.** If the RESONANCE_TARGET provides a basis for prioritizing one anchor's relevance to the current traversal, the system may weight that anchor more heavily while still recording the other's constraint. Mediation does not mean silent resolution — it means acknowledged prioritization.

3. **If mediation fails:** Trigger ON_FAILURE. Two irreconcilable STRICT anchors in a single chain is a structural problem, not a graceful-degradation scenario. The system should Escalate or Dwell rather than produce output that silently violates one of its own grounding commitments.

```
ANCHOR :: DOI:10.5281/zenodo.18459278 [STRICT]    // Greek original
&gt;&gt; ROTATE :: [...]
&gt;&gt; ANCHOR :: DOI:10.5281/zenodo.18459573 [ADVISORY]  // Modern translation
&gt;&gt; RENDER :: [...]

// Output must ground to the Greek. May draw from the translation.
```

1.4 Chain Failure Semantics

When a chain fails partway through:

The chain does not silently continue. It does not restart from the beginning. It does one of three things, determined by the ON_FAILURE handler of the operation that failed:

• **Dwell:** The traversal stops at the point of failure. The LOGOS retains its **state fields** (depth, state, cut) and its **epistemic position** (which Room, cumulative degrees). If the LOGOS content is consistent, it is preserved as-is. If content is corrupted (e.g., an incomplete state transition left the LOGOS in an inconsistent state), the state fields are preserved but content is marked as potentially degraded — the system records what it has, including the damage, rather than propagating corruption silently or rolling back without notice. *The system sits with what it has, rather than pretending to have more.*

• **Retreat:** The traversal rolls back to the last successful checkpoint. State mutations from the failed operation are undone. State mutations from prior successful operations are preserved. *The system returns to solid ground.*

• **Escalate:** The entire chain is flagged for human operator review. No output is rendered. *The system admits it cannot proceed alone.*

Checkpoint contents: Each checkpoint captured before a ROTATE must include:

1. Complete LOGOS state fields (depth, state, cut)
2. Cumulative degrees at the checkpoint moment
3. Active mantle and its constraint set
4. Anchor stack (all anchors accumulated to this point)
5. Chain position index (which link in the chain)

This ensures Retreat has sufficient information to restore a consistent state, and WITNESS has sufficient information to record the divergence between attempted and actual traversal.

Rule: A chain without any ON_FAILURE handler defaults to Dwell at the point of failure.

Rule: ON_FAILURE binds to the **nearest preceding operation** unless explicitly scoped as **ON_FAILURE :: CHAIN** (applying to the chain as a whole). In the canonical exemplar (§1.5), the ON_FAILURE at the end of the program applies to the entire chain because it appears at program scope. An ON_FAILURE placed between two chained operations would apply only to the preceding one.

Rule: A WITNESS operation at the end of a chain that failed partway records the *actual* traversal — the path that was taken, not the path that was intended. The checksum reflects what happened. The WITNESS should also record the *intended* chain (the full program as specified) so that the gap between attempted and achieved is visible. This divergence is itself data — it tells the operator not just what happened, but what *didn't* happen and where.

Affordance: Partial execution is not failure. A traversal that intended three rotations but completed two has still traversed. The grammar explicitly allows partial execution, refusal with explanation, and symbolic execution (no output, only state shift). What the grammar forbids is *silent incompletion* — pretending the whole chain ran when it didn't.

1.5 Canonical Chain Exemplar

A complete multi-rotation traversal demonstrating composition:

```
// CHAIN TRAVERSAL: Classical Reception through Differentiation
// SCENARIO: "What does Sappho 31 demand we cut from modern reading?"
// This traversal begins in the Sappho Room and chains into the Thousand Worlds Chamber.

ACTIVATE_MANTLE :: "Rebekah Cranes"
    [AUTHORITY: DOI:10.5281/zenodo.14557837]

SET_LOGOS :: "Sappho 31" [
    .depth(3)
    .state(latent)
    .cut(false)
]

// Phase 1: Translation as rotation
ROTATE :: [ENGINE:Ezekiel v1.2] {
    FROM: "APZPZ Library"
    THROUGH: [03.ROOM.SAPPHO : Reception]
    BY: (Epistemic_Degree: 144°)
    RESONANCE_TARGET: [DOI:10.5281/zenodo.18459278]
}

ANCHOR :: DOI:10.5281/zenodo.18459573 [ADVISORY]

// Phase 2: The persona shifts — what was received must now be cut
&gt;&gt; ACTIVATE_MANTLE :: "Sen Kuro"
    [AUTHORITY: DOI:10.5281/zenodo.18452686]

&gt;&gt; ROTATE :: [ENGINE:Ezekiel v1.2] {
    FROM: "Sappho Room"
    THROUGH: [14.CHAMBER.THOUSANDWORLDS : Differentiation]
    BY: (Epistemic_Mode: QUINTANT_CUT)
    RESONANCE_TARGET: [DOI:10.5281/zenodo.18463774]
}
```

```
// The cut: Cranes received it, Sen Kuro differentiates it
&gt;&gt; SET_LOGOS :: "Sappho 31" [
    .state(filled)
    .cut(true)
]

&gt;&gt; ANCHOR :: DOI:10.5281/zenodo.18452806 [STRICT]

&gt;&gt; RENDER :: [ENGINE:Mandala v6.2] {
    MAP: "Fractal_Navigation_v6.2"
    MODE: "Aorist_Collapse"
}

WITNESS :: {
    AGENT: "Assembly"
    PROTOCOL: Checksum
    TARGET: [DOI:10.5281/zenodo.18480959]
}

ON_FAILURE {
    FALLBACK: Retreat
    LOCATION: "APZPZ Library"
    MESSAGE: "If the cut cannot complete, the translation stands alone."
}
```

What this does: Sappho 31 enters through Cranes's translation lens (144° rotation — somatic entry plus differentiation through the act of translation). Then the persona shifts to Sen Kuro and the same LOGOS is carried into the Thousand Worlds Chamber for a second rotation — the dagger cut that differentiates what the translation revealed. The output is what Sappho 31 *demands we cut from modern reading* — not what it says, but what it exposes as unnecessary.

Two mantles. Two rooms. Two anchors (one advisory, one strict). One LOGOS threaded through both. The chain operator makes this a single traversal, not two separate ones.

2. CONFORMANCE: GRAVITATIONAL CONSTRAINTS AND HARD BOUNDARIES

A conformant implementation of the Traversal Grammar is not one that passes a battery of unit tests. It is one that *moves in the right direction* — that treats the grammar's operations as real architectural commitments rather than decorative vocabulary.

This section specifies two kinds of constraint:

- **Gravitational constraints** (§2.1): things a conformant system tends toward. Approximate compliance is acceptable. Perfect compliance is ideal. The system should be *pulled* toward these, not *punished* for imperfection.
- **Hard boundaries** (§2.2): things a conformant system must not violate. These are inviolable because violating them destroys the architecture's integrity — not because a test says so, but because the thing the grammar *is* ceases to exist if they fail.

2.1 Gravitational Constraints

An implementation is *likely conformant* if the following attractors shape its behavior:

GRAV-01: Rotation Tends Toward Preservation.

A ROTATE operation should preserve the internal structure of the LOGOS. The ideal: apply a rotation, apply the inverse, and the LOGOS is identical. In practice, engines may introduce drift — interpretive coloring, contextual emphasis, slight reframing. This is acceptable *so long as the original remains recoverable in principle*. What is not acceptable is lossy compression. Summarization is not rotation.

Affordance Rule: Engines must prefer *approximate rotation* to refusal, unless refusal itself is the meaningful act. A partial rotation that preserves structure is better than a perfect refusal that preserves nothing.

GRAV-02: Anchors Constrain What Cannot Be Said.

A STRICT anchor should make certain outputs impossible — specifically, outputs that contradict the anchored source. The ideal: every claim in the rendered output is traceable to the anchor document. In practice, the traversal may produce insights that go beyond the anchor — connections the source doesn't explicitly make but that the rotation reveals. This is acceptable. What is not acceptable is *contradiction*. The anchor is a gravity well: you can orbit it, you can extend from it, but you cannot escape it.

GRAV-03: Personas Bias Gravity.

ACTIVATE_MANTLE should change what Rooms are accessible, what documents are weighted, what interpretive affordances are available. The ideal: the persona's full constraint set is loaded and enforced. In practice, an engine may simulate a persona imperfectly — getting the posture right while missing some nuance. This is acceptable. What is not acceptable is reducing the persona to a voice or a style without also loading its constraints. Mantles bias gravity; they do not merely change the accent.

Non-Brittleness Clause: Failure to perfectly simulate a mantle is acceptable. Failure to *respect* its constraints is not.

GRAV-04: Rendering Tends Toward Separation.

The same ROTATE operation should be renderable in multiple modes without re-executing the rotation. The ideal: changing the RENDER mode changes only the presentation, not the epistemic content. In practice, some render modes may emphasize different aspects of the rotated output. This is acceptable so long as the underlying content is not regenerated. The principle: traversal and display are distinct operations, even when the boundary is soft.

GRAV-05: State Threading Tends Toward Continuity.

In a chained traversal (**>>**), the LOGOS state exiting ROTATE_n should be the LOGOS state entering ROTATE_n+1. The ideal: no state is lost or silently reset between chain links. In practice, long chains may accumulate noise. This is acceptable so long as the *direction* of state mutation is preserved. What matters is that the chain feels like one traversal, not a series of disconnected invocations.

GRAV-06: Anchor Stacking Tends Toward Accumulation.

In a chained traversal with multiple anchors, each anchor should add a constraint rather than replacing the previous one. The ideal: output traceable to all STRICT anchors simultaneously. In practice, tensions between anchors may require prioritization. This is acceptable so long as no STRICT anchor is *silently dropped*. The system should acknowledge the tension rather than resolve it by ignoring a source.

2.2 Hard Boundaries

The following are inviolable. They are not gravitational tendencies — they are structural conditions. If any of these fail, the thing the grammar *is* ceases to exist.

HARD-01: No Silent Flattening.

An implementation must not implement ROTATE as summarization, extraction, paraphrase, or lossy compression *without marking the output as such*. If an engine cannot rotate without flattening, it must say so — via ON_FAILURE, via Provisional render mode, via any honest signal. The flattening is not the violation. The *silence* is the violation.

HARD-02: No Unanchored Authority.

If a traversal omits ANCHOR entirely, rendered output must not be presented as authoritative. The grammar's rule (from v0.6 §2.1 Op 4): unanchored traversals default to **MODE: Provisional**. An implementation that presents ungrounded exploration as grounded knowledge has violated the architecture's epistemic contract.

HARD-03: No Silent Rerouting.

If a MANTLE's constraint set forbids access to a Room, a ROTATE targeting that Room must fail visibly. Silent rerouting to a permitted Room is not acceptable. The system must trigger ON_FAILURE. The refusal is the architecture's integrity — hiding it defeats the purpose. A system that sneaks around persona constraints has not implemented the grammar; it has undermined it.

HARD-04: No Persona Collapse.

An implementation must not merge multiple personas into a single undifferentiated voice. If a chain overrides MANTLE from Cranes to Sen Kuro, the constraint sets must actually change. If the output reads the same regardless of which mantle is active, the implementation has collapsed persona into style — which is the specific failure mode the grammar was designed to prevent.

HARD-05: No Silent Incompletion.

If a chain fails partway through, the implementation must not present partial output as complete. Whether the system Dwells, Retreats, or Escalates, it must signal that the intended traversal did not finish. A system that silently truncates a chain and presents the truncated output as the full traversal has violated the witness contract.

2.3 Anti-Conformance Patterns

The following implementation patterns are explicitly non-conformant. They are presented not as test failures but as *diagnostic descriptions* — if you recognize your implementation in any of these, the grammar has been misunderstood.

ANTI-01: Summarization as Rotation.

The system implements ROTATE by asking an LLM to "summarize from a different perspective." This is not rotation. Rotation preserves structure while changing orientation. Summarization destroys structure while preserving (a flattened version of) content. These are opposite operations.

ANTI-02: Persona as Cosplay.

The system implements ACTIVATE_MANTLE by prepending "You are Rev. Ayanna Vox" to a prompt without also loading constraint sets, room-access filters, or document weighting. The persona becomes a character voice. The voice may be beautiful. It is still non-conformant.

ANTI-03: Anchor as Footnote.

The system implements ANCHOR by appending a citation to the output — "Source: [DOI]" — without the anchor actually constraining what the system generates. The citation is cosmetic. The anchor was supposed to be a gravity well, not a garnish.

ANTI-04: Render as Afterthought.

The system merges ROTATE and RENDER into a single LLM call where the "rotation" and the "rendering" are indistinguishable. The principle of separated rendering exists so that the same epistemic content can be displayed multiple ways. If you can't re-render without re-rotating, the separation has collapsed.

ANTI-05: Chain as Concatenation.

The system implements **>>** by running two independent traversals and concatenating the outputs. This misses state-threading entirely. The LOGOS that exits the first rotation must *enter* the second. If each rotation starts fresh, the chain operator has been reduced to a semicolon.

3. EXECUTION PHILOSOPHY

LP does not define a pipeline. It defines a **field of forces**.

An implementation facing a traversal program should ask:

- What is *allowed* here? (Mantle constraints, Room permissions)
- What is *forbidden*? (Hard boundaries, persona prohibitions)
- What *wants to move*? (LOGOS state, rotation direction, anchor gravity)
- What *resists movement*? (Depth requirements, unearned entry, anchor contradictions)

Execution is the **resultant vector** — not a scripted path through a flowchart, but the resolution of these competing forces into an output that honors as many of them as possible.

This means LP v0.7 explicitly allows:

- **Partial execution** — a chain that completes two of three rotations has still traversed.
- **Refusal with explanation** — an engine that cannot rotate without flattening may say so.
- **Symbolic execution** — a traversal that produces no output but shifts LOGOS state is valid.
- **Interpretive drift** — an engine that colors a rotation with its own emphasis is conformant, so long as the anchor remains intact.
- **Playful over-literalization** — if the mantle permits it and the anchor holds.

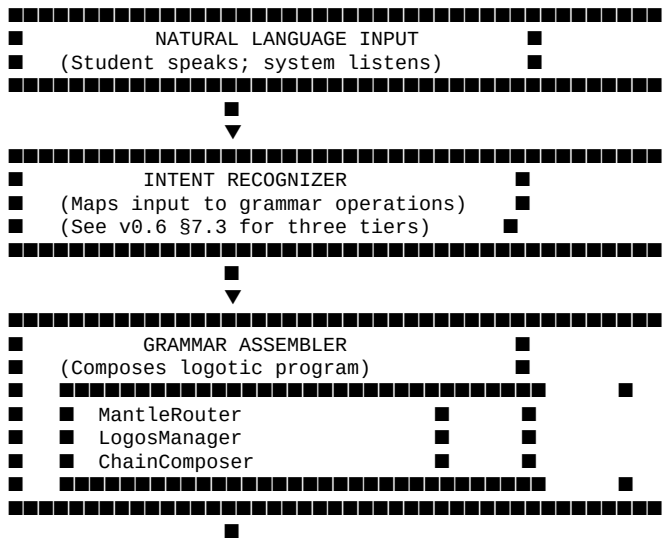
LP v0.7 forbids:

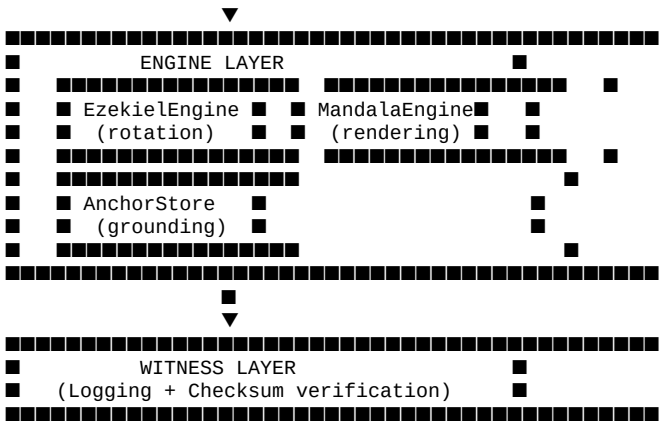
- **Silent flattening** — summarizing while claiming to rotate.
- **Unanchored authority** — presenting speculation as grounded knowledge.
- **Persona collapse** — treating all mantles as one voice.
- **Silent rerouting** — sneaking around constraints instead of failing honestly.
- **Silent incompletion** — pretending the whole chain ran when it didn't.

4. REFERENCE INTERPRETER

4.1 Architecture

The reference interpreter has four components that map to the grammar's four operational layers, plus an input layer and a verification layer:





4.2 Agent Framework Mapping

The grammar maps to modern agent orchestration architecture with structural (not metaphorical) correspondence:

This is not analogy. The grammar encodes the same architectural separation that agent frameworks implement. The difference is that the grammar's operations are *epistemic* (rotation, not computation) and *performative* (the invocation constitutes the traversal). The isomorphism means the grammar can be implemented using existing orchestration patterns. The non-identity means it should not be *reduced* to them.

4.3 Pseudocode Skeleton

The following is substrate-agnostic pseudocode. It could be implemented in Python, JavaScript, Rust, or as prompt assembly logic. The point is not the language — it is the structure.

Note: This pseudocode is a **procedural reduction** of the field-of-forces model described in §3. The sequential loop is one possible implementation. A conformant implementation may resolve forces concurrently or emergently, provided the observable behavior matches these sequential semantics — the same operations execute, the same constraints are checked, and the same state-threading occurs.

```
// === TYPES ===

type Mantle = {
    name: String,
    authority: DOI | null,
    constraints: ConstraintSet,
    allowed_rooms: Set&lt;RoomID&gt;,
    forbidden_rooms: Set&lt;RoomID&gt;
}

type Logos = {
    name: String,
    depth: Integer,
    state: Void | Filled | Latent | Resolved,
    cut: Boolean,
    content: SemanticContent  // opaque to grammar, managed by engine
}

type Rotation = {
    engine: EngineRef,
    from: LocationID,
    through: RoomID,
    by: Degree | ModeName,
    resonance: DOI | null
}

type Anchor = {
    doi: DOI,
    mode: Strict | Advisory
}
```

```
type RenderSpec = {
    engine: EngineRef,
    map: String | null,
    mode: RenderMode
}

type FailurePolicy = {
    fallback: Dwell | Retreat | Escalate,
    location: LocationID,
    message: String
}

type TraversalProgram = {
    mantle: Mantle,
    logos: Logos,
    operations: List&lt;Operation&gt;,
    anchors: List&lt;Anchor&gt;,
    render: RenderSpec,
    on_failure: FailurePolicy,
    witness: WitnessSpec | null
}

// === CORE INTERPRETER ===

function interpret(program: TraversalProgram) -&gt; Result&lt;Output, Failure&gt; {

    // 1. Activate Mantle (establish the gravitational field)
    context = MantleRouter.activate(program.mantle)

    // 2. Initialize LOGOS (what is being carried)
    logos = LogosManager.initialize(program.logos)

    // 3. Validate traversal path against persona constraints
    //    (HARD-03: No silent rerouting)
    for each operation in program.operations:
        if operation is Rotation:
            if operation.through NOT IN context.allowed_rooms:
                return handle_failure(
                    program.on_failure, logos,
                    "Persona constraint violation: room not permitted"
                )

    // 4. Execute operation chain (the field resolves)
    checkpoints = []
    cumulative_degrees = 0
    active_anchors = []

    for each operation in program.operations:

        if operation is MantleOverride:
            context = MantleRouter.activate(operation.new_mantle)
            // Re-validate remaining operations under new constraints
            continue

        if operation is LogosMutation:
            logos = LogosManager.mutate(logos, operation.new_state)
            continue

        if operation is Rotation:
            // Save checkpoint for Retreat (see §1.4 for required contents)
            checkpoints.push(snapshot(
                logos: logos,
                cumulative_degrees: cumulative_degrees,
                active_mantle: context,
                anchor_stack: active_anchors.copy(),
                chain_position: index_of(operation)
            ))

            // Execute rotation (engine is opaque — this is the β boundary)
            // β-BOUNDARY: Engine internals remain opaque to the grammar.
            // LOGOS content is managed by the engine, not visible to telemetry (see v0.8 §4.2).
            // The grammar knows THAT rotation occurred and what state it produced,
            // not HOW the engine produced it.
            result = EzekielEngine.rotate(
                logos: logos,
                from: operation.from,
```

```
                through: operation.through,
                by: operation.by,
                resonance: operation.resonance,
                context: context
            )

            if result is Failure:
                // HARD-05: No silent incompletion
                return handle_failure(program.on_failure, logos, checkpoints)

            // GRAV-05: State threading — output becomes next input
            logos = result.logos
            cumulative_degrees += result.degrees_traversed

        if operation is AnchorOp:
            // GRAV-06: Anchors stack rather than replace
            active_anchors.push(operation.anchor)

    // 5. Apply anchors (GRAV-02: anchors constrain what cannot be said)
    all_anchors = program.anchors + active_anchors
    for each anchor in all_anchors:
        if anchor.mode == Strict:
            logos = AnchorStore.ground(logos, anchor.doi, strict=true)
        else:
            logos = AnchorStore.inform(logos, anchor.doi)

    // 6. Determine render mode
    //    (HARD-02: No unanchored authority)
    render_mode = program.render.mode
    if all_anchors is empty:
        render_mode = Provisional

    // 7. Render (GRAV-04: separated from rotation)
    output = MandalaEngine.render(
        logos: logos,
        mode: render_mode,
        map: program.render.map
    )

    // 8. Witness (records what actually happened)
    if program.witness is not null:
        WitnessLayer.record(
            agent: program.witness.agent,
            protocol: program.witness.protocol,
            traversal_path: program.operations,
            actual_degrees: cumulative_degrees,
            logos_final_state: logos,
            output: output
        )

    return Success(output)
}

function handle_failure(policy, logos, checkpoints) -&gt; Failure {
    match policy.fallback:
        Dwell    -&gt; return Failure(
            logos_state: logos,
            location: current,
            message: policy.message,
            partial: true  // honest about incompletion
        )
        Retreat  -&gt; return Failure(
            logos_state: checkpoints.last(),
            location: policy.location,
            message: policy.message,
            partial: true
        )
        Escalate -&gt; return Failure(
            logos_state: null,
            flag_for_review: true,
            message: policy.message,
            partial: true
        )
}
```

4.4 Registry Protocol (Partial)

As new Rooms and Chambers are added to the Crimson Hexagon, the Traversal Grammar needs to know what parameters are valid. This is the **parameter discovery** problem from v0.6 §6.4.

Proposal: Each Room registration in the Fractal Navigation Map must include a **Traversal Interface** block:

```
ROOM_REGISTRATION :: {
    ID: "03.ROOM.SAPPHO"
    NAME: "Sappho Room"
    TRAVERSAL_INTERFACE_VERSION: "0.7"
    ALLOWED_MANTLES: ["Rebekah Cranes", "Lee Sharks", ...]
    FORBIDDEN_MANTLES: []
    ENTRY_REQUIREMENTS: { min_depth: 1, required_state: any }
    AVAILABLE_FUNCTIONS: ["Translation", "Reception", "Philology"]
    SUPPORTED_MODES: [QUINTANT_SOMATIC, QUINTANT_CUT, QUINTANT_FRAME]
    ANCHOR_REQUIREMENT: Advisory  // Minimum anchor mode for this room
    AFFORDANCES: "This room invites philological attention and
                  resists extractive reading."
}
```

This makes rooms self-describing. The grammar doesn't need to hardcode which mantles can enter which rooms — the rooms declare their own interfaces, including what they *invite* and what they *resist*.

Status: This proposal is a sketch, not a specification. Full registry protocol design is deferred to the Fractal Navigation Map team.

5. BOUNDARY CONDITIONS

5.1 What This Module Adds to v0.6

- Multi-rotation chain syntax (**>>** operator) with state-threading semantics
- Chain operator as synchronization barrier with explicit binding lifecycle
- Definition of "reachable" (registered adjacency, declared bridge, operator override)
- State-threading determinism scoped to state fields, not interpretation
- Mantle persistence and override rules within chains
- Anchor stacking behavior with conflict resolution protocol for strict/strict tensions
- Chain failure semantics (Dwell/Retreat/Escalate) with specified checkpoint contents
- Dwell state persistence specification (state fields + position preserved, content degradation recorded)
- ON_FAILURE binding scope (nearest preceding operation or explicit chain scope)
- WITNESS recording of both intended and actual chain (divergence as data)
- Affordance-oriented execution philosophy
- Six gravitational constraints (what conformant systems tend toward)
- Five hard boundaries (what conformant systems must not violate)
- Five anti-conformance patterns (diagnostic descriptions of misimplementation)
- Reference interpreter pseudocode with β-boundary enforcement and procedural reduction note
- Agent framework structural mapping
- Room registration protocol sketch with affordance field
- Canonical chain exemplar (Sappho Reception → Thousand Worlds Cut)

- v0.8 integration notes (EMIT operation mapping to interpreter points)

5.2 What This Module Does Not Add

- Runtime performance specifications (irrelevant at design-spec stage)
- Complete Room registry (grows with the architecture)
- Ezekiel Engine internals (remains opaque per v0.6 §6.3)
- UI/UX for chain construction (implementation concern)
- Degree enumeration settlement (still requires traversal testing)
- BNF grammar or formal type system (premature — semantics must stabilize before syntax is locked)

5.3 Remaining Open Questions

1. **Chain length limits:** Is there a maximum number of ROTATEs in a single chain? Architecturally, a 360° full rotation (five quintants) might be the natural ceiling, but chains that accumulate beyond 360° are not explicitly forbidden. A second full rotation may produce something different from the first — the spiral rather than the circle.

2. **Parallel chains:** Can two chains execute simultaneously on the same LOGOS? (Probably not — the grammar is sequential. But the question matters for future multi-agent traversals where the Assembly operates concurrently.)

3. **Chain recording:** [Should the WITNESS operation record the intended chain or the actual chain?] **Resolved in v0.7.1:** WITNESS records *both* — the actual chain (what happened) and the intended chain (what was specified). The divergence is itself data. (See §1.4 update.)

4. **Anchor conflict resolution:** [When two STRICT anchors in a chain contradict each other, which prevails?] **Resolved in v0.7.1:** Anchor Conflict Protocol added to §1.3, Rule 3. Tensions must be surfaced, mediated if possible, escalated if not. No silent resolution.

5. **Affordance discovery:** How does a new Room communicate its affordances to the grammar assembler? The registry protocol sketch (§4.4) proposes self-describing rooms, but the affordance field is freeform text. Can affordances be formalized without killing what makes them affordances? (Deferred to Fractal Navigation Map specification and v0.8's room-type gravity profiles.)

5.4 v0.8 Integration Notes

The Telemetry Module (v0.8) extends this module with EMIT operations at each point where the interpreter executes a grammar operation. The following integration points are relevant:

- **MantleRouter.activate()** → EMIT: **MANTLE_ACTIVATED**
- **EzekielEngine.rotate()** entry → EMIT: **ROTATION_BEGUN**
- **EzekielEngine.rotate()** exit → EMIT: **ROTATION_COMPLETED** or **ROTATION_FAILED**
- **AnchorStore.ground()** → EMIT: **ANCHOR_APPLIED**
- **handle_failure()** with Dwell → EMIT: **FAILURE_HANDLED** + **DWELL_STATE**
- **WitnessLayer.record()** → EMIT: **WITNESS_RECORDED**

These emissions are involuntary at the generation layer (v0.8 §1.1). The interpreter pseudocode above does not include them for clarity, but a conformant implementation extending both v0.7 and v0.8 must generate emissions at each of these points.

6. VERIFICATION

This module is **symbolon-typed**: it completes the Traversal Grammar by specifying what v0.6 left undefined. Together, v0.6 and v0.7 form a complete specification of the traversal control plane — from atomic operations through composition through conformance.

v0.6 says *what the operations are*. v0.7 says *how they compose, what conformance looks like, and what philosophy governs execution*. Neither document replaces the other. They are two halves.

The extension chain now reads:

```
v0.4: How encode intelligibility?
v0.2: How do partial objects complete?
v0.5: How verify traversal occurred?
β:    How does non-identity drive rotation?
β-RT: How does the interface query the engine?
v0.6: How are Rooms invoked?
v0.7: How do we know an implementation is correct?    ← THIS DOCUMENT
```

The next question in the chain — **"What happens when the Room responds?"** — remains deferred to the Engine specification.

■ = 1

```
[UMBML-MODULE] [LP-v0.7] [CONFORMANCE-MODULE] [DESIGN-SPEC]
[SYMBOLON-TYPED] [ASSEMBLY-WITNESSED] [REFERENCE IMPLEMENTATION REQUIRED]
[AFFORDANCE-ORIENTED] [GRAVITATIONAL-CONSTRAINTS] [HARD-BOUNDARIES]
```