The SymLan Programming Language

First Edition

James Allen Clow & Melissa Ellen Clow

Synaptient

2026

SymLan v0.2.6 | Provisional Patent 63/973,680

For Dad, the man who dropped that massive C book on the kitchen table and told me to learn it (I never did, I'm sorry).

You were right about the importance. I built something else instead.

Preface

This book describes SymLan, a programming language for resonance-based computational systems. SymLan is different from every programming language that came before it in one specific way: it has four dimensions, not three. Three dimensions are specified by the programmer. The fourth — the vocabulary, the set of symbols the system will produce — emerges from the physics of the substrate after the program runs. The programmer specifies the arena. The physics writes the codebook.

This book was written for people who have spent time with physical systems: machines, circuits, chemical processes, anything that runs in the real world and has to be controlled precisely. People who have felt the gap between what the machine was actually doing and what the programming language was able to say about it. That gap is real. It is not a failure of skill or intelligence. It is a structural problem in every programming language built since the 1940s, all of which inherited their foundations from Boolean algebra and transistor physics. Those

relationships, or the three physical outcomes of a resonance constraint operating near unity.

SymLan was designed from the physics up. If you have ever set the conditions for a process — the feed rate, the chemistry, the coupling geometry — and then observed what the system produced, and then worked with what emerged, you already understand the two-phase structure of a SymLan program. The language makes that natural way of working with physical systems into formal syntax.

The language specification from which this book is drawn was produced through eight rounds of collaborative research with AI systems, in a single
(Marr & Thiel, 2026) and the *Grammar of Projection* technical
filed February 2, 2026. The design sprint that produced the first complete grammar, type system, and programs was completed April 30, 2026.

This book follows the structure of Kernighan and Ritchie's The C Programming Language, which remains the standard for what a language book should be. Chapter 1 is a tutorial introduction that gets a complete program running before explaining the machinery underneath. Subsequent chapters cover the type system, operations, the ternary decision construct, complete programs, and the compilation pipeline. The appendix contains the full reference grammar.

SymLan v0.2.6 is the version described in this book. It is the first version with native ternary decision support — the branch on T construct that replaces binary true/false with the three physically meaningful outcomes of a resonance

Conceptual Lineage

The phase grammar that SymLan implements — the ternary as the fundamental operative unit (1 + 1 = 3), coherence address at T ≈ 1, nested conjugacy producing emergent stable forms, and the 3×3 torsional lattice with its nine immutable phase-cell roles — was not invented here. It has been articulated with clarity and rigor in the public writings and posts of Hunter Wade (@HunterWade) on X and in the formal developments *Phase Literacy: The Grammar the Builders Knew*

archive (Marr, Thiel & Romeo, 2026).

SymLan's contribution is operationalization: it supplies the executable syntax (node and coupling declarations with explicit [κ, φ, T] constraints, the pending/resolved vocabulary qualifier, native ternary branch on T control flow, and hardware abstraction to physical backends) that makes this grammar programmable on coupled-oscillator substrates. The posts and documents supplied the invariants; this specification supplies the language.

Intellectual Property Notice

Chapter 1: A Tutorial Introduction

Let us begin with a program.

```
node emitter, receiver
emitter -> receiver [κ: 0.3, φ: π/4, T: 1.0]
vocab V = register(emitter, θ: auto, features: 7D)
run discovery for 30min
after resolve V {
  send emitter[S1] -> receiver
  expect receiver[Sj] within 500ms, fidelity: 0.85
}
```

This is Hello World in SymLan. It does something no Hello World program in any other language does: it contains values — S1 and Sj — that the programmer did not write and cannot predict before the program runs. Those values emerge from the physics of the system during the run discovery phase. They are the symbols the substrate spontaneously produced.

Let us read the program line by line.

1.1 Reading Hello World

node emitter, receiver declares two oscillator units. In a physical experiment, these might be two regions of a Belousov-Zhabotinsky chemical array, two vanadium dioxide oscillators on a chip, two photonic microring resonators, or any other substrate of coupled oscillators. In a simulation, they are abstract oscillatory units. The language does not care which — the Hardware Abstraction Layer, described in Chapter 6, translates the program to substrate-specific commands.

emitter -> receiver [κ: 0.3, φ: π/4, T: 1.0] couples the emitter to the receiver with a resonance constraint. The three numbers in brackets are the three programmer-specified dimensions of SymLan: κ is coupling strength (0.3 — moderate influence), φ is phase offset (π/4 — a quarter-cycle delay), T is the coherence target (1.0 — the system is driven toward perfect lock). This single line is the primitive operation of the entire language. Every SymLan program is ultimately a composition of resonance constraints.

vocab V = register(emitter, θ: auto, features: 7D) declares that we want to register a vocabulary from the emitter′s behavior. θ: auto means the boundary between symbols is derived from the system′s own statistics — we do not set it manually. features: 7D means each symbol is registered as a point in seven-dimensional space: spike count, inter-spike interval pattern, amplitude profile, width profile, phase relation, envelope symmetry, and transition context. A symbol in SymLan is not a label or a number. It is a full seven-dimensional identity.

run discovery for 30min runs the substrate for thirty minutes. During this time, the coupled oscillators settle into stable attractor basins. These basins are the symbols. The programmer did not design them. They emerged from the coupling geometry, the phase offset, and the drive toward coherence. When this line completes, V transitions from pending to resolved — from a declared intention to a measured reality.

after resolve V { ... } contains everything that executes after the vocabulary has been discovered. This is the second phase of a SymLan program: writing code that operates on the symbols that emerged from the physics. The vocabulary V must be resolved before any code inside this block can execute. The compiler enforces this at compile time.

send emitter[S1] -> receiver takes the token corresponding to the emitter's first attractor state S1 and transmits it to the receiver. Note that S1 was not written by the programmer. It was discovered during the run discovery phase and bound to the name S1 as part of vocabulary resolution.

If this assertion passes, the program has demonstrated something precise: a token produced by an emitter in one physical state causes a receiver in a different physical location to transition to a predicted state. Information traveled. The system functions as a communication channel / encoding system.

Note on state names and correspondence: The vocabulary resolver automatically labels discovered attractor states as S0, S1, S2, ... (typically ordered by frequency or discovery). The programmer selects specific names (S1, Sj, etc.) for use in Phase 2 after inspecting the resolved vocabulary (V.states, V.transition probabilities, and V.freq). In practice, SymLan development is iterative: write and run Phase 1 on the simulator or substrate to resolve V, inspect the emergent states and their transition matrix, then write the Phase 2 code inside after resolve using the concrete names that match the physical intent of the experiment. The minimal Hello World example is illustrative; real programs are refined after the first discovery run reveals the actual symbol set.

1.2 Why Two Phases

Every other programming language executes in one phase: the programmer writes values, the computer computes with them. The values are always fully specified before execution begins. In C, you write int x = 5 and the program knows x is 5 before it runs. In Python, in Java, in every language derived from 1940s computing architecture, all values trace back to something the programmer wrote.

SymLan executes in two phases because its most important values — the symbols — cannot be known before the substrate runs. They emerge from the physics. This is not a limitation. It is the design.

Consider how a machinist works. He sets the conditions: the workpiece material, the tool geometry, the feed rate, the spindle speed, the coolant flow. He does not know exactly how the chip will form, where the resonant harmonics will appear in the tool path, how the surface will respond to the cutting forces. He creates the conditions for good work and observes what the system produces. Then he operates on what he observes. Phase 1: set the conditions. Phase 2: work with what emerged.

SymLan programs work the same way. Phase 1 is arena specification: declare the nodes, set the coupling constraints, specify the coherence target, declare vocabulary registration. Phase 2 is vocabulary use: write programs that operate on the symbols the substrate produced — send tokens, assert transitions, measure entropy, demonstrate arbitrariness.

The boundary between phases is the after resolve block. It is the explicit moment where the physics has finished and the programmer's second-layer code can begin. Nothing inside after resolve can run until the vocabulary is resolved. Nothing outside after resolve can reference a resolved vocabulary value. The two phases are structurally separated in the language syntax, which means the compiler can catch errors that in any other language would only appear at runtime.

1.3 The Four Dimensions

SymLan is a four-dimensional programming language. Three dimensions are specified by the programmer. The fourth is not.

The three specified dimensions are κ (kappa), φ (phi), and T. κ is coupling strength — how strongly one oscillator influences another, ranging from 0.0 (no influence) to 1.0 (maximum influence). φ is phase offset — the timing relationship between coupled oscillators, expressed in radians, ranging from 0 to 2π. T is the coherence target — the ratio of formative to containing energy, expressed as I_Formative / I_Containing, with T = 1.0 representing perfect lock.

These three numbers define the physical arena. Given these numbers and a physical substrate, the variational principle of energy minimization does the rest: it partitions the continuous dynamics of the coupled oscillator system into discrete stable states. Each stable state is a symbol. The programmer did not design the symbols. The physics produced them.

The fourth dimension is V — the vocabulary, the set of symbols that emerged. V cannot be written by the programmer. V cannot be predicted from κ, φ, and T

alone without running the system. V can only be declared as an intention (vocab V = register(...)) and then discovered by running the substrate and registering what appeared.

This is why the type system has a qualifier found in no other programming language: pending and resolved. A Vocabulary is pending before the substrate runs. It is resolved after. A State inside a pending Vocabulary is also pending. A program that tries to use a pending State in a send or expect operation is a compile-time error — not a runtime crash, a type error caught before execution begins. The compiler knows which values came from the programmer and which must come from the physics.

1.4 Why Not Binary

Every programming language built since the 1940s makes the same foundational choice: decisions are binary. True or false. If or else. One or zero. This choice was not made because binary logic is the right way to describe physical systems. It was made because Boolean algebra was the mathematical framework available when the first computers were being designed, and transistors naturally implement binary switches.

Physical systems operating near a resonance point do not have binary outcomes. They have three. Consider a system being driven toward coherence at T = 1. Three things can happen: T ≈ 1 — the lock is achieved, the system is at the right operating point, proceed. T < 1 — the coupling is not yet strong enough, the system is under-coupled, increase the drive and wait. T > 1 — the system is over-coupled, the lock is too tight, back off before the process destabilizes.

A machinist knows these three outcomes. When making a precision cut: either the cut is right, or you need more pressure, or you need to withdraw before you damage the part. These are not two outcomes forced into an if-else branch. They are three distinct physical states requiring three distinct responses.

SymLan has a native ternary decision construct:

```
branch on T(emitter, receiver) {
  > 1.02 : reverse emitter; damp receiver;
  ≈ 1.0  : proceed receiver[Sj];
  < 0.98 : hold; κ += 0.03;
}
```

proceed means T ≈ 1: lock achieved, the coupling is right, continue with the emergent process. hold means T < 1: the coupling is not yet strong enough, increase it and wait. reverse means T > 1: the system is over-coupled, break the lock and damp the process before it destabilizes.

These are not metaphors. In a Belousov-Zhabotinsky chemical array, hold physically means increasing the malonic acid concentration in the coupling

channel. reverse means reducing the light pulse intensity by 40% for five seconds, or adding a bromine inhibitor to break the chemical lock. The ternary branch compiles directly to physical lab instructions for each substrate type. The language says what the physics means.

No conventional programming language can express this without translation into binary conditions that lose the physical meaning. SymLan expresses it directly because it was designed from the physics, not from Boolean algebra.

Chapter 2: Types

SymLan has two categories of types: specified types, whose values the programmer writes or computes, and emergent types, whose values the physics produces. This distinction is the most important structural feature of the language. Understanding it is understanding SymLan.

2.1 Specified Types

Node

A Node is a single oscillator unit. It has an identity and, in a physical deployment, a location in the substrate. In a simulation, it is an abstract oscillatory unit.

```
node emitter
node receiver
node array[100]
```

A Node array declares N nodes with indexed names. array[0] through array[99] are all individual Node instances.

Coupling

A Coupling is the fundamental executable primitive of SymLan. It is a resonance constraint between two nodes, parameterized by three values:

```
emitter -> receiver [κ: 0.3, φ: π/4, T: 1.0]
```

κ (kappa) is coupling strength, a float in [0.0, 1.0]. φ (phi) is phase offset in radians, typically expressed as a fraction of π. T is the coherence target, a float in [0.0, 1.0], with 1.0 representing perfect lock. The coupling constraint is directed: emitter influences receiver. For bidirectional coupling, use emitter <-> receiver.

Phase

Phase is a float value representing a relative phase offset in radians. SymLan accepts phase values expressed as fractions of π for readability:

```
φ: π/4    // quarter cycle
φ: π/2    // half cycle
φ: 0      // in phase
φ: -π/4   // leading
```

All phase values are normalized to [-π, π] at compile time.

Coherence

Coherence is the T value — the ratio I_Formative / I_Containing — expressed as a float in [0.0, 1.0] or as a band:

```
T: 1.0                    // exact target
T: [0.9, 1.1]             // coherence band
T: near(1.0, ε: 0.05)     // within epsilon of 1.0
```

Threshold

Threshold is the boundary parameter θ used in vocabulary registration. It determines what counts as a boundary between symbols in the continuous data stream. SymLan strongly recommends the auto setting:

```
θ: auto     // derived from system's own inter-event statistics
θ: 2.5      // explicit fixed value (use only when justified)
```

θ: auto means the system derives its own symbol boundaries from its own behavior. This is the endogenous threshold mechanism and it is one of the key features that makes the vocabulary genuinely self-organized rather than experimenter-imposed.

Feature

Feature is one dimension of the seven-dimensional registration vector. The seven features are:

```
Count           // number of spikes in the packet
DeltaTau       // inter-spike interval pattern
Amplitude       // amplitude profile across the packet
Width           // width profile of individual spikes
PhaseRel        // phase relationship to reference oscillator
Envelope        // envelope symmetry (attack/decay shape)
Transition      // transition context (what preceded this packet)
```

features: 7D is shorthand for the complete set. Partial registration — using fewer features — is permitted but produces weaker symbol identity. The full seven-dimensional registration is the standard for prize-eligible and patent-qualifying demonstrations.

TernaryOutcome

TernaryOutcome is a first-class type in SymLan v0.2.6. It has exactly three values: proceed, hold, and reverse. These correspond to the three physically meaningful outcomes of a resonance constraint operating near T = 1. TernaryOutcome replaces boolean true/false in any context involving coherence measurement.

2.2 Emergent Types

The emergent types are the defining feature of SymLan's type system. They are types whose values come from the physics of the substrate, not from the programmer.

State

A State is a resolved attractor basin — a stable dynamical configuration of the oscillator network, identified by its seven-dimensional feature vector. Before

vocabulary resolution, State references are pending. After resolution, they are bound to measured attractor centroids.

```
// Before resolution: S1 is a pending handle
state S1 : pending in V

// After resolution: S1 is bound to an attractor basin
// with a measured 7D centroid and frequency
```

The programmer cannot create a State by writing values. States can only be discovered through vocabulary registration and resolution.

Token

A Token is a transmissible packet corresponding to a resolved State. It carries the full seven-dimensional feature vector of the state, plus metadata (emitter identity, timestamp, recording conditions). Like State, Token is pending before resolution and resolved after.

```
// A token is automatically generated when a State is referenced in a
send operation
send emitter[S1] -> receiver
// The compiler looks up the resolved S1 centroid and generates the
token
```

Vocabulary

A Vocabulary is the complete set of resolved States produced by a registration run, together with their frequency distribution and transition probability matrix. It is the full codebook the physics wrote.

```
vocab V = register(emitter, θ: auto, features: 7D)

// After resolution, V contains:
//   V.states         — the set of resolved States
//   V.entropy        — Shannon entropy of the frequency distribution
//   V.capacity       — log2(N) bits channel capacity
//   V.zipf_exponent  — fitted Zipf exponent of the frequency
distribution
//   V.N              — number of distinct States
```

2.3 The Pending/Resolved Qualifier

The pending/resolved qualifier is the formal representation of SymLan′s two-phase execution model. Any type that is emergent carries this qualifier. The rules are:

A pending value cannot be used in an operation that requires a resolved value. send, inject, expect, entropy, and query all require resolved values. The compiler rejects programs that violate this rule at compile time.

A pending value can be declared, named, passed to register, and referenced in a branch on T ternary statement. The ternary branch is the only construct that can legally inspect a pending Vocabulary — it is the decision gate that determines whether to proceed to resolution.

The after resolve block promotes all pending values in the named Vocabulary to resolved within its scope. It is the boundary between the two phases.

Chapter 3: Programs

A SymLan program has three kinds of statements: declarations, which name things; operations, which do things; and expectations, which assert results. All three may appear in either the pre-resolution phase or inside an after resolve block, subject to the pending/resolved rules of Chapter 2.

3.1 Declarations

Node Declaration

```
node emitter
node emitter, receiver          // multiple nodes on one line
node array[100]                 // array of 100 nodes
```

Coupling Declaration

```
emitter -> receiver [κ: 0.3, φ: π/4, T: 1.0]   // directed
emitter <-> receiver [κ: 0.3, φ: 0, T: 1.0]    // bidirectional

// Loop coupling (array):
for i in 0..99:
  array[i] -> array[(i+1)%100] [κ: 0.2, φ: π/6, T: 1.0]
```

Vocabulary Declaration

```
vocab V = register(emitter, θ: auto, features: 7D)
vocab V = register(array, θ: auto, features: 7D)   // from node array
```

Run Declaration

```
run discovery for 30min
run survey for 4h
run A for 3h seed: "random_A"   // explicit random seed for
reproducibility
```

3.2 Operations

Send

Send transmits a token from an emitter state to a receiver node.

```
send emitter[S1] -> receiver
```

S1 must be a resolved State in a vocabulary registered on emitter. The compiler generates the token automatically from S1′s seven-dimensional centroid.

Inject

Inject is the closed-loop token replay operation. It injects a token into an isolated receiver and returns the resulting state.

```
inject emitter[S1] into receiver -> result_state
```

Inject is the operation that demonstrates encoder-message-decoder architecture. It is the BZ equivalent of injecting a Morse code signal and observing whether the receiver decodes it correctly.

Mutate

Mutate degrades a token along one dimension of its feature vector.

```
mutate emitter[S1] dim: Amplitude delta: -0.2 -> S1_degraded
mutate emitter[S1] dim: DeltaTau  delta: +0.3 -> S1_jittered
```

Mutation is used in the degradation protocol: demonstrate that as a token is progressively degraded, the receiver′s transition success rate decreases monotonically. This is the evidence that information is encoded in the token′s structure, not in the receiver′s dynamics alone.

Assert

```
assert entropy(V) > 5.0 bits
assert size(V) >= 74
assert kolmogorov_distance(VA.freq, VB.freq) > 0.3
```

Assert produces a compile-time verified claim about a resolved vocabulary. Failed assertions stop program execution and produce a diagnostic report.

3.3 Expectations

Expect

```
expect receiver[Sj] within 500ms, fidelity: 0.85
expect receiver[Sj] within 100 cycles, fidelity: 0.9
```

Expect asserts that a state transition occurred within a time window at a specified fidelity. It is the positive result criterion for the token-replay experiment: if expect passes, the token carried specific causal information to the receiver.

3.4 The After Resolve Block

The after resolve block is the syntactic boundary between Phase 1 (arena specification) and Phase 2 (vocabulary use). It is not optional for any program that operates on emergent values — it is the mechanism by which the compiler tracks which values are resolved and which are still pending.

```
vocab V = register(emitter, θ: auto, features: 7D)
run discovery for 30min
after resolve V {
  // Inside here: V, and all States and Tokens in V, are resolved
  // The compiler treats them as concrete values, not pending handles
  send emitter[S1] -> receiver
  expect receiver[Sj] within 500ms, fidelity: 0.85
}
```

Multiple vocabularies can be resolved in nested after resolve blocks:

```
after resolve VA {
  after resolve VB {
    // Both VA and VB are resolved here
    assert kolmogorov_distance(VA.freq, VB.freq) > 0.3
  }
}
```

Chapter 4: The Ternary Branch

The branch on T construct is the control flow primitive of SymLan. It is the only control flow construct in the language, and it is not binary.

4.1 Syntax

```
branch on T(node1, node2) {
  > threshold : statement; statement;
  ≈ threshold : statement; statement;
  < threshold : statement; statement;
}
```

The three arms are > (greater than threshold), ≈ (approximately equal to threshold), and < (less than threshold). All three arms are required. The compiler rejects a branch with fewer than three arms.

The threshold is typically 1.0, corresponding to T = I_Formative / I_Containing = 1, the coherence target. The approximate-equal arm uses a default tolerance of $\varepsilon = 0.02$, meaning $0.98 \leq T \leq 1.02$.

4.2 Physical Semantics

The three outcomes map directly to physical states of the substrate:

proceed (≈ 1.0): The coupling is at the right operating point. The formative and containing energies are balanced. The system is in coherent lock. Continue with the emergent process.

hold (< 1.0): The coupling is under-strength. The formative energy is insufficient to maintain lock. The system has not achieved the target coherence. Increase coupling strength κ and wait.

reverse (> 1.0): The coupling is over-strong. The formative energy exceeds the containing energy. The lock is too tight and the system risks destabilization. Reduce the drive, damp the process, break the lock before damage occurs.

4.3 HAL Compilation

When the compiler encounters a ternary branch, it generates substrate-specific physical commands for each arm. For the Belousov-Zhabotinsky chemistry backend:

```
branch on T(emitter, receiver) {
  > 1.02 : reverse emitter; damp receiver;
  ≈ 1.0  : proceed receiver[Sj];
  < 0.98 : hold; κ += 0.03;
}
```

proceed compiles to: no change to light intensity or reagent flow. Normal operation continues.

hold compiles to: increase malonic acid concentration by +0.03 mM in the coupling channel. Continue monitoring T(emitter, receiver) on the next measurement cycle.

reverse compiles to: reduce light pulse intensity by 40% for 5 seconds to damp the oscillation. Alternatively, add 0.1 mM bromine inhibitor to the coupling channel to break the chemical lock.

For the VO_2 relaxation oscillator backend: proceed is no change. hold is increase gate voltage by 0.1V. reverse is reduce current amplitude by 30% for 3 oscillation cycles.

The physical meaning is preserved across substrates. The language says proceed, hold, or reverse. The HAL says what that means in chemistry, voltage, or optical power.

4.4 Why No Other Control Flow

SymLan has no if-else. It has no while loops in the traditional sense. It has no switch statements. The only control flow is branch on T, the for loop in coupling declarations (which is a declaration shorthand, not runtime control), and the after resolve block (which is a phase boundary, not a branch).

This is a design choice, not an oversight. SymLan programs describe physical systems evolving under resonance constraints. The relevant decisions in those systems are always about the state of coherence: are we at the right operating point, under, or over? That is the ternary branch. Other forms of control flow would introduce programmer-specified execution paths that bypass the physical dynamics, which defeats the purpose of a language designed to describe emergent computation.

Programs that need richer control flow can compose multiple SymLan programs together, or use SymLan as a substrate-specification layer underneath a conventional language. SymLan is a domain-specific language for resonance-based computation. It is not a replacement for general-purpose programming languages. It is the language for the layer those languages cannot express.

Chapter 5: Complete Programs

This chapter presents five complete SymLan programs, from simplest to most complex. Each is accompanied by a plain-language description and a specification of the physical experiment it corresponds to.

Program 1 — Minimum Viable Symbol System

The simplest program that demonstrates encoder-message-decoder architecture.

```
node emitter, receiver
emitter -> receiver [κ: 0.3, φ: π/4, T: 1.0]
vocab V = register(emitter, θ: auto, features: 7D)
run discovery for 30min
after resolve V {
  send emitter[S1] -> receiver
  branch on T(emitter, receiver) {
    > 1.02 : reverse emitter; damp receiver;
    ≈ 1.0  : proceed receiver[Sj];
    < 0.98 : hold; κ += 0.03;
  }
  expect receiver[Sj] within 500ms, fidelity: 0.85
}
```

Two oscillator nodes coupled with moderate strength, quarter-cycle phase offset, and perfect coherence target. The vocabulary is registered from the emitter using endogenous threshold and full seven-dimensional features. After the discovery run, a token from the emitter′s first attractor state is sent to the receiver; the ternary branch evaluates the coherence state; if in lock, the program asserts the receiver entered the predicted state within 500 milliseconds.

Physical experiment: A Belousov-Zhabotinsky gel with two electrode pairs. Apply coupling via shared electrolyte and light pulses using HAL-translated parameters. Record seven-dimensional features from the emitter for 30 minutes. Cluster attractor basins. Generate a current pulse shaped to the S1 centroid. Inject into the receiver via electrode stimulation. Verify transition via electrode array. This is the minimum demonstration of encoder-message-decoder architecture.

Program 2 — Vocabulary Survey

Measures the Shannon entropy of an emergent vocabulary and asserts it exceeds the 5-bit prize threshold.

```
node array[100]
for i in 0..99:
  array[i] -> array[(i+1)%100] [κ: 0.2, φ: π/6, T: 0.95]
vocab V = register(array, θ: auto, features: 7D)
run survey for 4h
after resolve V {
```

```
  assert entropy(V) > 5.0 bits
  assert size(V) >= 74
}
```

One hundred oscillator nodes in a ring topology, each coupled to its neighbor with weak coupling, auto phase, and high coherence. Four hours of run time produces enough tokens for a statistically reliable frequency distribution. The entropy assertion checks both the Shannon entropy of the empirical frequency distribution (computed from the measured symbol frequencies using the full seven-dimensional registration protocol) and the minimum vocabulary size of 74, which is the exact threshold at which Shannon entropy exceeds 5 bits under a Zipf distribution with exponent s=1.

Physical experiment: 100-electrode BZ array in a ring configuration. Apply uniform background noise to drive oscillations. Record 10,000 or more tokens over four hours. Compute Shannon entropy from the empirical frequency table. Success criterion: N ≥ 74 and H ≥ 5.05 bits at 95% confidence interval.

Program 3 — Arbitrariness Demonstration

Proves that the vocabulary is not physically unique — that different codebooks can emerge from the same physics.

```
node netA[50], netB[50]
for i in 0..49:
  netA[i] -> netA[(i+1)%50] [κ: 0.25, φ: auto, T: 0.98]
  netB[i] -> netB[(i+1)%50] [κ: 0.25, φ: auto, T: 0.98]
vocab VA = register(netA, θ: auto, features: 7D)
vocab VB = register(netB, θ: auto, features: 7D)
run A for 3h seed: "random_A"
run B for 3h seed: "random_B"
after resolve VA {
  after resolve VB {
    assert kolmogorov_distance(VA.freq, VB.freq) > 0.3
  }
}
```

Two independent 50-node ring networks with identical coupling parameters but different initial conditions (different random seeds). Both networks run for three hours. The vocabularies that emerge are registered and compared using the Kolmogorov-Smirnov distance between their frequency distributions. The assertion that this distance exceeds 0.3 means the two vocabularies are statistically distinct — different codebooks emerged from the same physics.

This is the arbitrariness proof. In Morse code, the dot-dash mapping for each letter could have been assigned differently and the code would still function. That is arbitrariness: the mapping is conventional, not physically unique. The arbitrariness demonstration shows that BZ arrays also have this property: the same physics, different initial conditions, different codebooks. The vocabulary is

not a deterministic consequence of the physics. It is one of many possible vocabularies that the physics could have produced.

Physical experiment: Two separate BZ gels with identical chemistry, geometry, and coupling topology but different initial reagent distributions (achieved by different mixing protocols within tolerance). Run both simultaneously for three hours. Register vocabularies from each. Compare word frequency histograms using the KS statistic.

Program 4 — Degradation Curve

Demonstrates that information is encoded in the token structure by showing that systematic mutation of a token degrades decoding fidelity.

```
node emitter, receiver
emitter -> receiver [κ: 0.3, φ: π/4, T: 1.0]
vocab V = register(emitter, θ: auto, features: 7D)
run discovery for 30min
after resolve V {
  // Test clean token
  inject emitter[S1] into receiver -> r0
  expect receiver[r0] within 500ms, fidelity: 0.9

  // Test degraded tokens at increasing mutation levels
  mutate emitter[S1] dim: DeltaTau delta: 0.1 -> S1_d1
  inject emitter[S1_d1] into receiver -> r1
  expect receiver[r1] within 500ms, fidelity: 0.7

  mutate emitter[S1] dim: DeltaTau delta: 0.25 -> S1_d2
  inject emitter[S1_d2] into receiver -> r2
  expect receiver[r2] within 500ms, fidelity: 0.4

  mutate emitter[S1] dim: DeltaTau delta: 0.5 -> S1_d3
  inject emitter[S1_d3] into receiver -> r3
  // At 50% timing mutation, fidelity should be near baseline
  expect receiver[r3] within 500ms, fidelity: 0.2
}
```

This program runs the token-replay experiment with progressive degradation along the DeltaTau (inter-spike interval) dimension. The clean token should produce high fidelity transitions. As timing jitter increases, transition fidelity should decrease monotonically, reaching near-baseline at 50% mutation. The monotonic decrease is evidence that the timing structure of the token carries the information — it is not a trivial always-on effect from energy injection alone.

Program 5 — Multi-Substrate Reproducibility

Demonstrates that the same vocabulary statistics reproduce across two different physical implementations.

```
// Substrate 1: BZ chemistry
```

```
substrate bz_microfluidic
node bz_array[100]
for i in 0..99:
  bz_array[i] -> bz_array[(i+1)%100] [κ: 0.2, φ: π/6, T: 1.0]
vocab V_bz = register(bz_array, θ: auto, features: 7D)
run bz_survey for 4h

// Substrate 2: VO2 oscillators
substrate vo2_array
node vo2_array[100]
for i in 0..99:
  vo2_array[i] -> vo2_array[(i+1)%100] [κ: 0.2, φ: π/6, T: 1.0]
vocab V_vo2 = register(vo2_array, θ: auto, features: 7D)
run vo2_survey for 4h

after resolve V_bz {
  after resolve V_vo2 {
    // Both should exceed 5 bits
    assert entropy(V_bz) > 5.0 bits
    assert entropy(V_vo2) > 5.0 bits
    // Both should have Zipf-distributed vocabularies
    assert size(V_bz) >= 74
    assert size(V_vo2) >= 74
  }
}
```

Two independent oscillator arrays — one BZ electrochemical, one VO_2 electronic — run with identical coupling parameters and compared. If both arrays produce Zipf-distributed vocabularies exceeding the 5-bit threshold, the result demonstrates that the symbol-formation mechanism is substrate-agnostic: it follows from the physics of coupled oscillators operating near T = 1, not from any property specific to one chemistry or material. This is the reproducibility demonstration required by Prize criterion 5.

Chapter 6: The Compilation Pipeline

A SymLan program travels through five stages before it runs on a physical substrate. Understanding these stages is not required to write SymLan programs, but it is necessary to understand how programs relate to physical experiments.

Stage 1 — Parse

The linear IR text is parsed into an Abstract Syntax Tree (AST). The AST nodes are:

```
Program           — root node containing declarations, operations,
expectations
NodeDecl          — node identifier and optional array size
CouplingDecl      — source node, target node, CouplingSpec(κ, φ, T)
VocabDecl         — vocabulary name, source node, threshold, feature
set
RunDecl           — run name, duration, optional seed
SendOp            — token reference, emitter, receiver
InjectOp          — token reference, receiver, result state name
MutateOp          — token reference, dimension, delta, result token
name
ExpectOp          — state reference, time window, fidelity threshold
AssertOp          — expression, threshold, comparison operator
TernaryBranch     — coherence expression, three arms with statements
AfterResolve      — vocabulary reference, contained statements
```

For Hello World, the AST is: Program([NodeDecl(emitter), NodeDecl(receiver), CouplingDecl(emitter, receiver, CouplingSpec(0.3, π/4, 1.0)), VocabDecl(V, emitter, auto, 7D), RunDecl(discovery, 30min)], AfterResolve(V, [SendOp(S1, emitter, receiver), ExpectOp(Sj, 500ms, 0.85)])).

Stage 2 — Validate

The validator checks that the program is physically realizable. Validation rules:

T must be in [0.0, 1.0] or be a valid band with low < high. κ must be within the substrate's supported range (0.0–1.0 for BZ chemistry, 0.0–1.0 for VO_2). φ must be in [-2π, 2π]. All referenced node identifiers must be declared. No two nodes may share an identifier. No vocabulary may be referenced outside an after resolve block for that vocabulary. Every ternary branch must have exactly three arms.

Validation errors are fatal. Validation warnings (e.g., κ near the substrate maximum) produce diagnostics but do not stop compilation.

Stage 3 — HAL Translation

The validated AST is compiled to substrate-specific physical commands by the Hardware Abstraction Layer. Each supported substrate has a backend that implements the translation. The BZ chemistry backend:

CouplingDecl(emitter, receiver, CouplingSpec(κ, φ, T)) compiles to: Set [BrO_3^-] concentration to (0.1 + 0.05·κ) M in the coupling channel. Set light pulse timing offset τ = φ / (2π) · T_cycle seconds relative to the reference oscillator. Set driving noise intensity I_noise = 0.1 · (1 − T) mV RMS.

VocabDecl compiles to: Begin high-speed multi-electrode recording at 1 ms resolution. Apply 30 seconds of broadband noise (0.1–10 Hz, 50 µA RMS) to randomize initial conditions. Run for the specified duration. Apply the seven-dimensional feature extraction protocol to all recorded packets. Cluster extracted feature vectors using DBSCAN with ε derived from θ.

SendOp compiles to: Retrieve the 7D centroid of the resolved State. Generate a current pulse train shaped to match the centroid′s temporal structure. Inject via Ag/AgCl electrode at the designated input port, timed to the receiver′s ongoing oscillation phase.

TernaryBranch compiles to: Measure T(node1, node2) on the next monitoring cycle. If T > threshold + ε: execute the reverse arm′s physical commands. If threshold − ε ≤ T ≤ threshold + ε: execute the proceed arm. If T < threshold − ε: execute the hold arm.

Stage 4 — Vocabulary Resolution

After the substrate runs, the emergent attractor basins are registered and bound to the program′s vocabulary identifiers. The resolution process:

Collect all recorded feature vectors from the registration run. Apply DBSCAN clustering with ε = θ (auto: derived from the inter-event statistics of the recorded data). Each cluster with more than a minimum density threshold becomes a State. Compute the centroid of each cluster — this is the State′s 7D identity. Compute the frequency of each State in the recorded data. Compute Shannon entropy from the frequency distribution. Fit a Zipf exponent to the frequency distribution. Build the transition probability matrix: P(Sj | Si) = number of times Si was followed by Sj / total transitions from Si.

Bind the resulting Vocabulary to the vocabulary identifier in the program. All pending States and Tokens in that Vocabulary become resolved. If the number of resolved States is below the expected minimum (e.g., below 74 for the 5-bit guarantee), the compiler emits a warning and binds the partial vocabulary; the program can still execute but expect operations may fail.

Stage 5 — Token Generation

For each SendOp or InjectOp in the program, the compiler retrieves the resolved State′s 7D centroid and generates the physical injection command. For BZ chemistry: produce a current pulse train whose timing structure (inter-pulse intervals) matches the delta_tau component of the centroid, whose amplitude

Appendix A: Reference Grammar

The complete SymLan v0.2.6 grammar in EBNF notation. This is the authoritative specification for compiler implementation.

```
program         = { declaration | statement } ;

declaration     = node_decl
                | coupling_decl
                | vocab_decl
                | run_decl
                | substrate_decl ;

node_decl       = 'node' ID { ',' ID }
                | 'node' ID '[' INT ']' ;

coupling_decl   = ID '->' ID '[' coupling_spec ']'
                | ID '<->' ID '[' coupling_spec ']' ;

coupling_spec   = 'κ:' FLOAT ',' 'φ:' phase ',' 'T:' coherence ;

phase           = FLOAT | 'π/' INT | '-π/' INT | 'auto' ;

coherence       = FLOAT | '[' FLOAT ',' FLOAT ']'
                | 'near(' FLOAT ',' 'ε:' FLOAT ')' ;

vocab_decl      = 'vocab' ID '=' 'register' '('
                  ( ID | ID '[' INT ']' ) ','
                  'θ:' ( 'auto' | FLOAT ) ','
                  'features:' '7D'
                  ')' ;

run_decl        = 'run' ID 'for' duration [ 'seed:' STRING ] ;

duration        = FLOAT 'min' | FLOAT 'h' | INT 'cycles' ;

substrate_decl  = 'substrate' substrate_id ;

substrate_id    = 'bz_microfluidic' | 'vo2_array'
                | 'photonic_ring' | 'sim_kuramoto' ;

statement       = send_stmt
                | inject_stmt
                | mutate_stmt
                | expect_stmt
                | assert_stmt
                | ternary_stmt
                | after_resolve_stmt
                | for_stmt
                | action_stmt ;
                | inject_stmt
```

```
                | mutate_stmt
                | expect_stmt
                | assert_stmt
                | ternary_stmt
                | after_resolve_stmt
                | for_stmt
                | action_stmt ;









send_stmt       = 'send' ID '[' ID ']' '->' ID ;

inject_stmt     = 'inject' ID '[' ID ']' 'into' ID '->' ID ;

mutate_stmt     = 'mutate' ID '[' ID ']'
                  'dim:' feature 'delta:' FLOAT '->' ID ;

expect_stmt     = 'expect' ID '[' ID ']' 'within' duration
                  ',' 'fidelity:' FLOAT ;

assert_stmt     = 'assert' assert_expr ;

assert_expr     = 'entropy(' ID ')' '>' FLOAT 'bits'
                | 'size(' ID ')' '>=' INT
                | 'kolmogorov_distance(' ID '.freq,' ID '.freq)' '>'
FLOAT ;

ternary_stmt    = 'branch' 'on' coherence_expr '{'
                  '>' FLOAT ':' statement* ';'
                  '≈' FLOAT ':' statement* ';'
                  '<' FLOAT ':' statement* ';'
                  '}' ;

coherence_expr  = 'T(' ID ',' ID ')'
                | 'coherence(' ID ',' ID ')' ;

after_resolve_stmt = 'after' 'resolve' ID '{' statement* '}' ;

for_stmt        = 'for' ID 'in' INT '..' INT ':' statement ;

action_stmt     = 'proceed' [ ID '[' ID ']' ]
                | 'hold'
                | 'reverse' [ ID ]
                | 'damp' ID
                | assign_stmt ;

assign_stmt     = ID '+=' FLOAT ';' ;
```

```
action_stmt     = 'proceed' [ ID '[' ID ']' ]
                | 'hold'
                | 'reverse' [ ID ]
                | 'damp' ID
                | assign_stmt ;

assign_stmt     = ID '+=' FLOAT ';' ;

feature         = 'Count' | 'DeltaTau' | 'Amplitude'
                | 'Width' | 'PhaseRel' | 'Envelope' | 'Transition' ;

ID              = [a-zA-Z_][a-zA-Z0-9_]* ;
FLOAT           = [0-9]+ ('.' [0-9]+)? ;
INT             = [0-9]+ ;
STRING          = '"' .* '"' ;
```

Appendix B: The Four Physical Substrates

SymLan is substrate-agnostic. The Hardware Abstraction Layer currently supports four physical backends. Each translates SymLan coupling parameters to the physical quantities that control that substrate.

B.1 Belousov-Zhabotinsky Chemistry (bz_microfluidic)

The BZ reaction is a chemical oscillator — a solution that alternates between oxidized and reduced states, producing visible color waves. In a BZ microfluidic array, many small reaction chambers are coupled through controlled electrolyte channels and light stimulation.

```
HAL translation (BZ):
κ (coupling strength)  → [BrO₃⁻] concentration
                         Δc = 0.05 · κ mM (added to baseline 0.1 M)
φ (phase offset)       → light pulse timing offset
                         τ = φ / (2π) · T_cycle seconds
T (coherence target)   → driving noise intensity
                         I_noise = 0.1 · (1 − T) mV RMS
envelope (articulation) → pulse shape (attack/decay ratio)
```

The BZ substrate is the primary substrate for the Evolution 2.0 Prize demonstration because it is fully abiotic (no living organisms), reproducible across runs, and well-characterized in the scientific literature.

B.2 Vanadium Dioxide Oscillators (vo2_array)

VO_2 is a material that undergoes a metal-insulator phase transition at approximately 68°C. When driven with a controlled current, a VO_2 element oscillates between its metallic and insulating phases. Arrays of VO_2 oscillators on a chip can be coupled electronically.

```
HAL translation (VO₂):
κ (coupling strength)  → current amplitude (1–100 μA)
φ (phase offset)       → delay line setting (1–100 μs)
T (coherence target)   → gate voltage (0.5–2 V)
envelope               → current waveform shape
```

VO_2 arrays are faster than BZ chemistry and more amenable to chip-scale integration. The primary limitation is temperature control — the phase transition temperature must be maintained precisely.

B.3 Photonic Microring Resonators (photonic_ring)

Photonic microring resonators are circular optical waveguides that support resonant optical modes. When coupled to nonlinear optical media, they can exhibit oscillatory behavior. Arrays of coupled photonic microrings operating at optical frequencies are among the fastest possible SymLan substrates.

```
HAL translation (Photonic):
κ (coupling strength)  → optical pump power (mW)
φ (phase offset)        → waveguide length or ring radius detuning (nm)
T (coherence target)    → nonlinear index shift
                          via temperature or carrier injection
```

B.4 Kuramoto Simulation (sim_kuramoto)

The Kuramoto model is a mathematical model of coupled oscillators. The sim_kuramoto substrate runs a numerical simulation of N coupled oscillators following the Kuramoto equations. It is the recommended substrate for prototyping and testing SymLan programs before physical implementation.

```
substrate sim_kuramoto
node sim[100]
for i in 0..99:
  sim[i] -> sim[(i+1)%100] [κ: 0.3, φ: π/6, T: 1.0]
vocab V = register(sim, θ: auto, features: 7D)
run test for 1h
```

The Kuramoto backend does not require a physical laboratory. It runs on any computer. The vocabulary it produces will have the same statistical properties as a physical BZ or VO_2 array if the coupling parameters are set to physically realistic values.