L4YAML: A Verified YAML 1.2.2 Parser in Lean 4

Nicolas F. Rouquette

L4YAML is a fully verified YAML 1.2.2 parser written in pure Lean 4. It delivers 2,770 machine-checked theorems across 63 proof modules, zero axioms, zero `sorry`, and zero `partial def` — while passing 100% of the YAML 1.2.2 specification examples and 100% of the applicable yaml-test-suite test IDs (225/225).

This manual documents the project's architecture, verification strategy, security model, and FFI bindings for C, Python, and Rust.

Contents

1. Overview

2. Architecture

3. Verification

4. Security

5. FFI Bindings

6. Building

7. Testing

8. Test Results

Index

2. Architecture

L4YAML implements a two-pass pipeline that mirrors the layered structure of the YAML 1.2.2 specification itself. Input bytes flow through a character-level scanner, producing a typed token stream, which a recursive-descent parser converts into a structured AST.

2.1. Scanner (Lexical Layer)

2.2. Token Parser (Syntactic Layer)

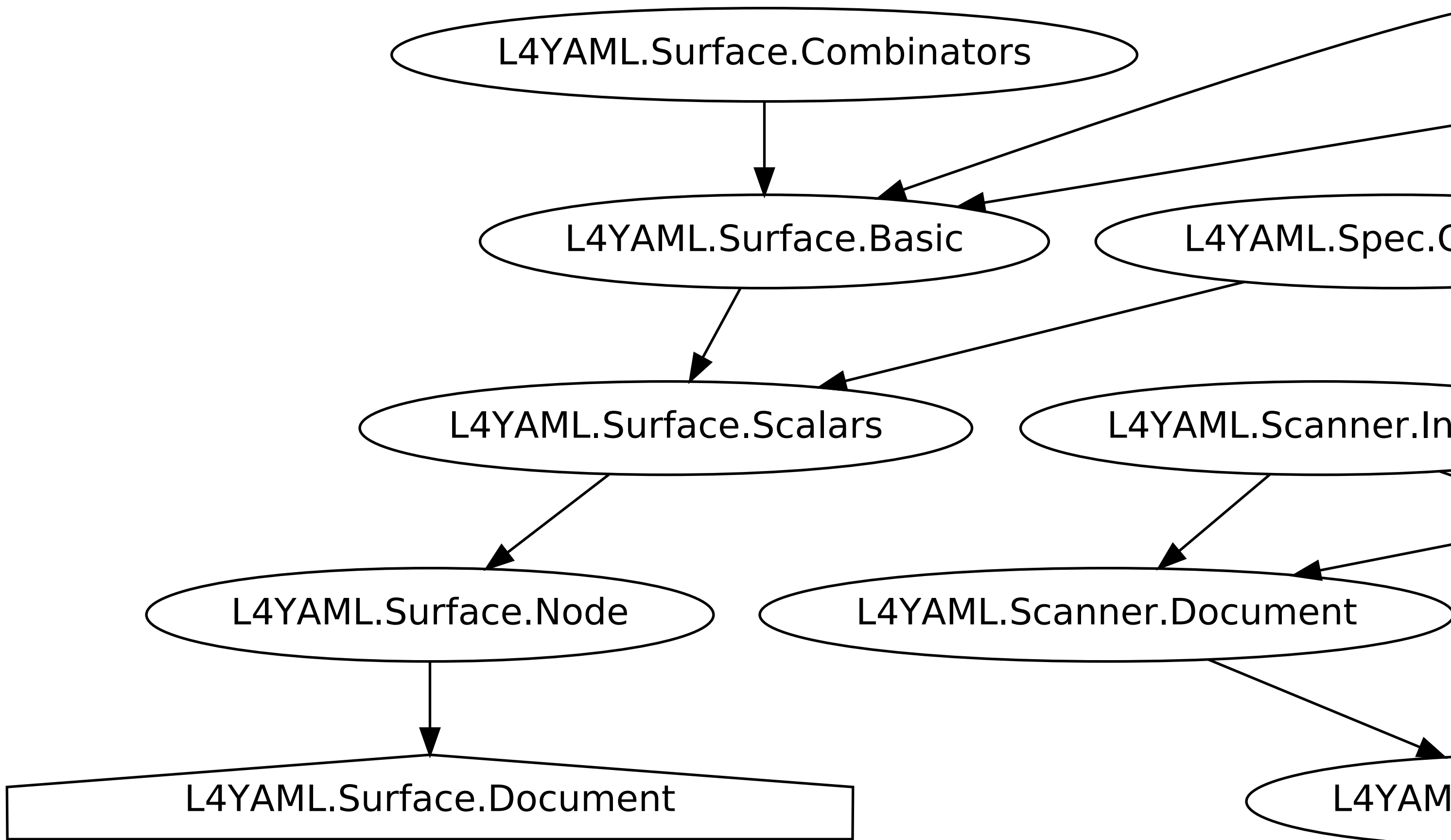
2.3. Type System

2.4. Module Organization

2.5. Import Graph

2.5. Import Graph

The runtime module dependency graph (Scanner, Parser, Surface, Schema, Output, FFI, Config — the `Proofs/` subtree is excluded) is regenerated in CI by `lake exe graph` and rendered to SVG via Graphviz.


L4YAML.Surface.Combinators
L4YAML.Surface.Basic
L4YAML.Spec.C
L4YAML.Surface.Scalars
L4YAML.Scanner.In
L4YAML.Surface.Node
L4YAML.Scanner.Document
L4YAML.Surface.Document
L4YAM

2.4. Module Organization

The project is organized into several module groups. The table below is regenerated at doc-elaboration time by walking each group's source directory under `L4YAML/`, so the file lists stay in sync with the actual code.

Group	Key Modules	Purpose
Spec	`CharPredicates.lean`, `Grammar.lean`, `Types.lean`, `YamlSpec.lean`, `Token.lean`	Type definitions, token types, grammar inductive, spec production predicates.
Config	`Config.lean`, `Limits.lean`	Parser limits and configuration presets (`strict` / `default` / `permissive` / `unlimited` / `safe_tags`).
Scanner	`Document.lean`, `Indent.lean`, `NodeProperties.lean`, `Scalar.lean`, `Scanner.lean`, `SimpleKey.lean`, `State.lean`, `Whitespace.lean`	Character-to-token conversion with full state management. Split into role-named submodules; the umbrella `Scanner.lean` owns flow-collection indicators and the `scanNextToken` dispatch / `scan` / `scanLoop` main loop.
Parser	`Composition.lean`, `Fuel.lean`, `State.lean`, `TokenParser.lean`	Token-to-AST recursive descent. `Composition.lean` owns the user-facing pipeline (`parseYaml*`, `scanAndParse`, comment classification); `TokenParser.lean` keeps the 14-function mutually-recursive block plus `parseStream` / `parseDocument`; `State.lean` holds `ParseState` + `NodeProperties` helpers; `Fuel.lean` factors out the `initialFuel := 4*N+4` formula.
Surface	`Basic.lean`, `Combinators.lean`, `Document.lean`, `Node.lean`, `Scalars.lean`, `Surface.lean`	Formal YAML 1.2.2 surface-syntax grammar productions used to state and discharge the acceptance-strictness theorem.
Schema	`Api.lean`, `Deriving.lean`, `Dump.lean`, `FromToYaml.lean`, `Schema.lean`, `Struct.lean`	Core Schema type resolution, structural API (`Api.lean`), and deriving for round-tripping user types via `FromToYaml.lean`.
Output	`Dump.lean`, `Emitter.lean`	Canonical `Emitter.lean`, style-aware `Dump.lean`, and `RoundTrip.lean` (`emit ∘ parse = id` properties).
FFI	`FFI.lean`	C/Python/Rust bindings via `@[export]`. The companion `ffi/`, `python/`, and `rust/` directories outside `L4YAML/` carry the non-Lean glue.
Proofs	63 modules	Machine-checked theorems for soundness, completeness, progress, and well-formedness.

2.1. Scanner (Lexical Layer)

The scanner (`Scanner.lean`, 594 lines) converts a UTF-8 input string into a stream of `YamlToken` values. It implements the spec's lexical-layer (L) productions and maintains several pieces of state:

- *Indentation stack* — tracks block-level nesting via column positions
- *Flow level counter* — distinguishes block context from flow context (`[`, `]`, `{`, `}`)
- *Simple key state* — manages the YAML "simple key" lifecycle where a plain scalar may retroactively become a mapping key when `:` is encountered
- *Anchor map* — records anchor names for alias resolution
- *Position tracking* — offset, line, and column for every token

The scanner returns `Except ScanError (Array YamlToken)`, where `ScanError` carries position information and a human-readable message.

2.1.1. Token Types

The `YamlToken` inductive defines the full set of lexical elements:

- Stream markers: `streamStart`, `streamEnd`
- Document markers: `documentStart`, `documentEnd`
- Structure indicators: `blockSequenceStart`, `blockMappingStart`, `blockEnd`, `flowSequenceStart`, `flowSequenceEnd`, `flowMappingStart`, `flowMappingEnd`, `key`, `value`, `blockEntry`, `flowEntry`
- Content tokens: `scalar` (with style tag: plain, single-quoted, double-quoted, literal, folded), `anchor`, `alias`, `tag`
- Placeholder tokens: used as reservation slots for simple key backpatching (filtered before output)

2.1.2. Append-Only Design

A critical design choice is the *append-only token stream*. When the scanner encounters a potential simple key, it pushes two placeholder tokens (for the future `key` and `blockMappingStart` indicators) and records their indices. If the key is later confirmed (by encountering `:`), these placeholders are overwritten in-place via `setIfInBounds`. If not confirmed, they remain as placeholders and are filtered out before the token stream is returned.

This avoids `insertAt` operations that would shift array indices and invalidate saved positions — a property that is essential for the formal proof that the scanner makes monotonic progress.

2.2. Token Parser (Syntactic Layer)

The token parser (`TokenParser.lean`, 790 lines) consumes the `YamlToken` stream and produces an `Array YamlDocument`. It implements the spec's grammar-layer (S) productions via hand-written recursive descent (no parser combinator library).

Key responsibilities:

- *Multi-document support* — handles `---` and `...` boundaries
- *Alias resolution* — substitutes `*anchor` references with previously anchored values
- *Tag resolution* — applies `%TAG` directive handle expansion and schema-level type resolution
- *Schema application* — resolves untagged scalars to typed values (`null`, `bool`, `int`, `float`, `str`) via the Core Schema

2.3. Type System

The output AST is centered on `YamlValue`, a compact inductive type:

- `YamlValue.scalar` — tagged string value with resolved `YamlType`
- `YamlValue.sequence` — ordered list of values
- `YamlValue.mapping` — list of key-value pairs (preserving order)

Each value is wrapped in `YamlDocument`, which carries document-level metadata including directive tags and version indicators.

Position tracking is provided by `YamlPos` (offset, line, column), enabling precise error reporting that maps back to the original input.

6. Building

L4YAML ships a top-level CMake driver that orchestrates Lake for the Lean side and compiles the C FFI and (optionally) Rust shim in the same configuration. The recommended workflow is therefore one CMake invocation rather than three separate per-language build commands.

6.1. Recommended Build

6.2. CMake Options

6.3. Python Install Modes

6.4. Prerequisites

6.5. Lean-Only Build

6.6. Standalone Per-Binding Builds

6.2. CMake Options

Option	Default	Effect
`L4YAML_BUILD_FFI`	`ON`	Build `libl4yaml.so` and `tryparse_c`
`L4YAML_BUILD_RUST`	`OFF`	Also build the Rust shim and `tryparse_rs`
`L4YAML_PYTHON_INSTALL`	`auto`	Python install mode: `auto`, `ament`, `venv`, `none`
`L4YAML_PYTHON_VENV`	(empty)	Path to a Python venv (used when mode is `venv`)
`L4YAML_ENABLE_TESTS`	`OFF`	Register lake-built test runners with CTest

6.5. Lean-Only Build

If you only need to typecheck and build the Lean library, proofs, and test executables — without the FFI shim or any Rust output — the unmodified Lake invocation works on its own:

```
lake build
```

This is also what the top-level CMake driver invokes internally as its first step.

6.4. Prerequisites

The minimum environment is:

- `elan` on `PATH` — provides the `lean` and `lake` versions pinned in `lean-toolchain`.
- A C compiler and a CMake ≥ 3.20 (the top-level project enables only the `C` language; the Lean side is driven through Lake).

`L4YAML_BUILD_RUST=ON` additionally requires:

- `cargo` (recent stable Rust; the workspace pins `edition = "2024"` and `rust-version = "1.85"`).
- `libclang` development files for `bindgen` to parse `ffi/l4yaml.h`.
- Network access on the first build, so `cargo` can fetch `bindgen` and `thiserror` from `crates.io`.

6.3. Python Install Modes

The Python package (`python/l4yaml`) is pure Python over `ctypes`, so it has no compile step — only an install step. The CMake driver supports three install paradigms, picked by the `L4YAML_PYTHON_INSTALL` option:

- **`ament`** — uses `ament_cmake_python` from a sourced ROS 2 environment. Installs the package at `${prefix}/lib/pythonX.Y/site-packages/l4yaml/`; colcon's `setup.bash` automatically prepends that path to `PYTHONPATH`. This is the right mode for a ROS 2 / colcon workflow.
- **`venv`** — runs `pip install -e python` from `${L4YAML_PYTHON_VENV}/bin/python` at configure time. The install is editable, so source edits in `python/l4yaml/` take effect without re-running CMake.
- **`none`** — CMake does not touch Python; the user installs the package manually with `pip install python` or equivalent.

The default `auto` mode picks `ament` when `ament_cmake_python` is on the CMake prefix path (i.e. a ROS underlay is sourced), `venv` when `L4YAML_PYTHON_VENV` is set, otherwise `none`. The same `colcon build` command therefore works in both a ROS shell (uses `ament`) and a plain dev shell (falls back to `none`), without flag changes.

6.3.1. ROS 2 / colcon

In a colcon workspace with a ROS underlay sourced, no additional flags are needed:

```
source /opt/ros/jazzy/setup.bash
colcon build --packages-select L4YAML
```

After build, source the workspace overlay (`. install/setup.bash`) and `import l4yaml` works.

6.3.2. venv (non-ROS)

For local development outside ROS, point the CMake driver at a venv:

```
python -m venv .venv && . .venv/bin/activate
cmake -B build -S . \
      -DL4YAML_PYTHON_INSTALL=venv \
      -DL4YAML_PYTHON_VENV=$VIRTUAL_ENV
cmake --build build -j
```

The `pip install -e python` runs at configure time and the venv keeps tracking `python/l4yaml/` thereafter.

6.1. Recommended Build

```
cmake -B build -S . -DL4YAML_BUILD_RUST=ON
cmake --build build -j
cmake --install build --prefix /path/to/stage
```

The `--install` step is optional; it stages the artifacts into a standard `${prefix}/{bin,lib,include}` layout plus the compiled Lean module tree under `${prefix}/lib/lean/`.

The build produces, in order:

- The Lean library, proof modules, compile-time guards, and every executable in the `L4YAML_EXES` list (a superset of the `defaultTargets` from `lakefile.toml`), driven by `lake build`.
- `libl4yaml.so` and the C example `tryparse_c`, defined in `ffi/CMakeLists.txt`.
- The Rust workspace (`l4yaml-sys`, `l4yaml`) and the `tryparse_rs` example, driven by `cargo build --release --workspace --examples`.

The installed C and Rust binaries have RPATHs that resolve `libl4yaml.so` via `$ORIGIN/../lib` and `libleanshared.so` from the Lean toolchain, so they work directly out of the install tree.

6.6. Standalone Per-Binding Builds

Each binding can also be built independently of the top-level driver, which is occasionally convenient when iterating on a single language layer:

```
# C shared library + header
cmake -B ffi/build ffi && cmake --build ffi/build

# Python package — venv-based editable install.  Or use the
# top-level CMake driver above with -DL4YAML_PYTHON_INSTALL=venv.
python -m venv .venv && . .venv/bin/activate
python -m pip install -e python

# Rust crates — needs libl4yaml.so on disk; either build it via
# the cmake line above (the default path is ffi/out/) or point
# at any other location with L4YAML_LIB_DIR.
cargo build --manifest-path rust/Cargo.toml
```

The top-level driver wires these together so that artifact paths match the install tree without needing per-binding configuration.

5. FFI Bindings

L4YAML provides foreign function interface bindings for C, Python, and Rust, enabling the verified parser to be used from mainstream languages. The FFI layer sits above the full parser pipeline — callers get the same verified parsing logic, security limits, and schema resolution available in Lean.

5.1. Dependency Graph

5.2. C API

5.3. Python API

5.4. Rust API

5.5. Adding FFI Bindings

5.5. Adding FFI Bindings

To expose a new Lean function via FFI:

1. Add `@[export l4yaml_function_name]` to the Lean definition
2. Declare the corresponding C prototype in `ffi/l4yaml.h`
3. Add any necessary shim code in `ffi/l4yaml_shim.c`
4. Update the Python/Rust wrappers as needed
5. Add tests at each layer

The `@[export]` attribute instructs the Lean compiler to generate a C-callable symbol with the specified name.

5.2. C API

The C API is defined in `ffi/l4yaml.h` with a bridge layer in `ffi/l4yaml_shim.c`. It provides approximately 26 exported functions covering:

- *Parsing* — `l4yaml_parse_safe` and related functions
- *Dumping* — converting `YamlValue` back to YAML text
- *Schema* — tag handle registration and type resolution
- *Limits* — configuring `ParserLimits` via preset codes or individual parameter setters
- *Memory* — proper Lean object lifecycle management (`lean_inc_ref` / `lean_dec_ref`)

All C API functions operate on opaque Lean object pointers. The shim layer handles the Lean runtime initialization and provides stable C-callable entry points.

5.1. Dependency Graph

The FFI module sits at the top of the dependency chain:

```
FFI
 └── Config
      └── Dump
           └── Schema
                ├── Token
                │    └── Scanner
                │         └── TokenParser
                └── CharPredicates
                     └── YamlSpec
```

`FFI.lean` exports functions via `@[export]` that are callable from C. The Python and Rust bindings wrap these C-level exports with idiomatic APIs in their respective languages.

5.3. Python API

The Python package (`python/l4yaml/`) wraps the C API using `ctypes`, providing a Pythonic interface with:

- Full type annotations for IDE support
- Automatic garbage collection of Lean objects via reference counting
- Cross-platform support (Linux, macOS, Windows)
- 78 tests covering the full API surface

Usage follows the familiar pattern:

```
from l4yaml import parse

result = parse("key: value")
```

5.4. Rust API

The Rust workspace (`rust/`) provides two crates:

- **`l4yaml-sys`** — raw FFI bindings generated from the C header. Unsafe, low-level, intended for direct interop.
- **`l4yaml`** — safe RAII wrapper with 21 tests. Lean object lifetimes are managed via the `Drop` trait, ensuring proper cleanup without manual reference counting.

The safe wrapper provides idiomatic Rust types and error handling via `Result`, translating Lean's `Except` into Rust's error model.

Index

A | B | C | D | F | H | K | L | M | P | R | S | T | V | Y | Z

A

append-only tokens

architecture

B

building

C

C API

canary theorem

compile-time guards

cross-language comparison

D

dependency graphs

diagnostic results

F

FFI

FFI test results

fibration gap

functorial chain

H

headline theorems

K

key theorems

L

L4YAML

Lean test results

M

machine-checked proofs

module organization

P

ParserLimits

Python API

Python install modes

parser correctness

prerequisites

presets

proof engineering

R

Rust API

round-trip

runtime tests

S

scanner

scanner correctness

security

soundness

specification coverage

T

test results

test suites

testing

token parser

V

verification

Y

YAML 1.2.2

YamlToken

YamlValue

yaml-test-suite coverage

Z

zero axioms

1. Overview

L4YAML is a pure Lean 4 implementation of the YAML 1.2.2 specification (RFC 9512). No external parsing libraries, no C dependencies in the core, and no use of `partial def` — every function is provably terminating.

The project demonstrates that a production-quality parser for a complex real-world format can be built and formally verified in Lean 4, with practical performance and comprehensive test coverage.

1.1. At a Glance

1.2. Key Design Decisions

1.1. At a Glance

Key Metric	Value
Machine-checked theorems	3,857 across 63 proof modules (~63,747 lines)
Compile-time `#guard` tests	2,142 (kernel-evaluated at build time)
Axioms / `sorry` / `partial def`	0 / 7 / 4
Runtime test suites	1,670 tests across 15 suites
Spec examples passing	132/132 (100%)
yaml-test-suite IDs	358/358 YAML 1.2.2-applicable (100%)
yaml-test-suite total	358/406 (88%; 48 skipped are YAML 1.1/1.3)

1.2. Key Design Decisions

The parser is built around several deliberate design choices:

- **Pure Lean 4, zero external dependencies.** The core parser has no C FFI calls and no dependency on external parsing libraries. This ensures every line of parsing logic is visible to the Lean kernel for formal verification.
- **Total functions only.** Every function terminates provably. The `partial` keyword is never used in production code. Termination is proved via well-founded recursion over input offsets, indentation levels, and flow nesting depth.
- **Append-only token streams.** Tokens are emitted into a pre-allocated array with placeholder reservation slots. Backpatching uses `setIfInBounds` (bounded update) rather than `insertAt` (which would shift indices). This design ensures monotonic progress and simplifies the formal proof of scanner correctness.
- **Two-pass architecture.** A character-level scanner emits a `YamlToken` stream, which a separate recursive-descent parser converts to a typed AST. This separation mirrors the YAML specification's own layered structure and enables independent verification of each layer.
- **Configurable security limits.** All resource bounds (nesting depth, string length, collection sizes) are configurable via `ParserLimits`, with four built-in presets from `strict` to `unlimited`.

4. Security

YAML parsing is a well-known attack surface. L4YAML addresses this with configurable parser limits, preset security profiles, and formal verification of the parsing logic itself.

4.1. Threat Model

4.2. ParserLimits

4.3. Preset Configurations

4.4. Verification of Security Properties

4.2. ParserLimits

The `ParserLimits` structure provides 11 configurable threat mitigations:

Parameter	Default	Purpose
`nestingDepth`	250	Maximum recursion depth — prevents billion-laugh expansion
`maxStringLength`	10 MB	Maximum scalar string length — DoS prevention
`maxArrayLength`	100,000	Maximum sequence element count
`maxObjectSize`	10 MB	Maximum total mapping size
`maxAliasDepth`	50	Maximum alias chain depth — recursive cycle protection
`allowDuplicateKeys`	`false`	Whether duplicate mapping keys are accepted
`allowedTagHandles`	customizable	Restricts which tag handles (`!`, `!!`, custom) are permitted
`forbiddenTags`	customizable	Explicit rejection of dangerous tags (e.g., `!!python/object`)
`parseErrorPolicy`	`strict`	Whether non-conformant input is rejected or best-effort parsed
`commentEncoding`	explicit	Character encoding validation for comments
`literalNewlineHandling`	standard	Newline normalization in literal scalars

4.3. Preset Configurations

Four preset configurations are provided for common use cases:

- **`strict`** — all protections enabled at conservative thresholds. Recommended for processing untrusted input (e.g., user uploads, network-received configuration).
- **`default`** — balanced settings suitable for most applications. Limits are generous enough for typical configuration files while still preventing resource exhaustion.
- **`permissive`** — reduced validation for trusted input. Useful when parsing known-good YAML from controlled sources.
- **`unlimited`** — all limits disabled. Explicitly dangerous; intended only for testing or for processing input that has already been validated externally.

The FFI layer exposes presets via `presetToLimits`, which maps a `UInt8` preset code to the corresponding `ParserLimits` configuration.

4.1. Threat Model

YAML parsers face several categories of attack:

- *Billion-laugh attacks* — deeply nested aliases that expand exponentially, exhausting memory
- *Denial of service* — extremely long strings, deeply nested structures, or very large collections that consume excessive resources
- *Arbitrary code execution* — YAML tags that trigger object deserialization in languages with unsafe constructors (e.g., Python's `!!python/object`)
- *Duplicate key confusion* — multiple identical keys in a mapping, where different consumers may pick different values

L4YAML mitigates all of these through its `ParserLimits` configuration.

4.4. Verification of Security Properties

The formal proofs establish that the parser correctly enforces its configured limits:

- Nesting depth is checked on every recursive call
- String length is checked during scalar accumulation
- Collection sizes are checked during sequence/mapping construction
- Alias depth is bounded during resolution

Because the parser is written in pure Lean 4 with no unsafe FFI in the core, there is no possibility of buffer overflows, use-after-free, or other memory safety violations in the parsing logic. The Lean runtime provides automatic memory management via reference-counted garbage collection.

8. Test Results

The L4YAML CI pipeline runs every test suite on each push and pull request. This chapter describes the test result categories and links to the most recent CI-generated reports.

Test results are generated by the `test-coverage.yml` GitHub Actions workflow and published alongside this documentation in the `reports/` directory.

The interactive coverage dashboard is available at reports/index.html.

- **8.1. yaml-test-suite Coverage**
- **8.2. Lean Test Suites**
- **8.3. Diagnostic Checks**
- **8.4. FFI Test Results**

8.3. Diagnostic Checks

Diagnostic pipelines produce detailed output for regression tracking:

Diagnostic	Output File	Description
Flow Regression	`flowregressioncheck.txt`	Flow parsing edge case regression tracking
Error Stage	`errorstagediag.txt`	Error stage diagnostic output
Scalar Stage	`scalarstagediag.txt`	Scalar stage diagnostic output

8.4. FFI Test Results

The FFI layer is tested through three independent paths:

8.4.1. Python FFI Tests

Two Python test suites validate the `ctypes`-based bindings:

- *FFI Integration Tests* (`Tests/test_python_ffi.py`) — tests the C shared library directly from Python
- *Package Tests* (`python/tests/`) — tests the `l4yaml` Python package API

Results are captured as both text logs and JUnit XML, with an HTML report generated from the JUnit data.

8.4.2. Cross-Language Comparison

The cross-language comparison runs the full yaml-test-suite through all four backends (Lean, C, Python, Rust) across five limit presets (`unlimited`, `default`, `strict`, `permissive`, `safe_tags`).

This verifies that the FFI layer preserves identical parsing behavior — every backend must produce the same pass/fail/skip results for every test case under every preset configuration.

Results are available at Cross-Language Comparison Report.

8.2. Lean Test Suites

The following Lean-native test suites are executed and their output captured:

Suite	Output File	Description
Unit Tests	`unit-tests.txt`	Core unit tests for parser internals
Demo	`demo.txt`	End-to-end parsing demonstrations
Explicit Key Tests	`explicitkeytests.txt`	YAML explicit key (`?`) handling
Flow Tests	`flowtests.txt`	Flow collection syntax completeness
Validation Tests	`validationtests.txt`	Input validation and error reporting
Dump Round-Trip	`dumproundtrip.txt`	Parse → dump → parse cycle verification
Raw Parse Tests	`rawparsetests.txt`	Token-to-AST parsing validation
Spec Examples	`specexamples.txt`	All 132 YAML 1.2.2 specification examples
Schema Dump	`schemadump.txt`	Schema resolution and dump formatting
Scanner Tests	`scannertests.txt`	Scanner behavior on targeted inputs
Scanner Spec Examples	`scannerspecexamples.txt`	Scanner-level tokens for spec examples
Adversarial Tests	`adversarialtests.txt`	Handcrafted adversarial inputs
Mutation Tests	`mutationtests.txt`	Systematic input perturbation
Property Tests	`propertytests.txt`	Randomized property-based testing
Production Coverage	`productioncoverage.txt`	Coverage across production-style inputs
Limit Tests	`limittests.txt`	Parser security limit enforcement

8.1. yaml-test-suite Coverage

The `suiterunner` executable runs all 406 unique test cases from the official yaml-test-suite and generates interactive HTML coverage reports with filtering, sorting, and per-stage breakdown.

Interactive reports:

- Full Coverage Report — all test cases with filtering and sorting
- Verified Tests Detail — all internal test suites with per-test results
- Production Coverage — YAML 1.2.2 productions cross-referenced with `@[yaml_spec]` annotations

Coverage by stage:

- Scalar Tests
- Flow Tests
- Block Tests
- Document Tests
- Advanced Tests
- Error Tests

7. Testing

Beyond the 2,309 formal theorems, L4YAML maintains extensive runtime and compile-time test suites that validate the parser against real-world YAML inputs and the official specification.

7.1. Specification Coverage

7.2. Test Suites

7.3. Running Tests

7.3. Running Tests

All tests are built as Lake executables. To run the full default test suite:

```
lake build
```

This builds all 415 jobs (zero warnings) including all test executables listed in `defaultTargets`. Individual suites can be run directly:

```
lake exe specexamples
lake exe scannertests
lake exe adversarialtests
```

Test results are captured in `docs/` as both plain text and HTML reports.

7.1. Specification Coverage

- **132/132** YAML 1.2.2 specification examples pass (100%)
- **225/225** unique YAML test IDs from the yaml-test-suite pass (100% of YAML 1.2.2-applicable tests)
- **354/406** total yaml-test-suite tests pass (87.2%)
- The 52 skipped tests cover YAML 1.1 and YAML 1.3 features that are outside the YAML 1.2.2 scope

7.2. Test Suites

The project includes 24 test executables spanning 19 hand-written suites and 7 diagnostic pipelines:

Suite	Focus
`specexamples`	All 132 YAML 1.2.2 specification examples
`scannerspecexamples`	Scanner-level token stream for spec examples
`scannertests`	Scanner behavior on targeted inputs
`rawparsetests`	Raw token-to-AST parsing
`flowtests`	Flow collection syntax (inline sequences, mappings)
`flowregressioncheck`	Regression tests for flow parsing edge cases
`explicitkeytests`	Explicit key (`?`) handling
`validationtests`	Input validation and error reporting
`limittests`	Parser limit enforcement
`adversarialtests`	Adversarial inputs targeting parser robustness
`mutationtests`	Systematic input perturbation
`propertytests`	Randomized property-based testing
`dumproundtrip`	Parse → dump → parse cycle validation
`schemadump`	Schema resolution and dump formatting
`productioncoverage`	Coverage analysis across production inputs
`errorstagediag`	Error stage diagnostic output
`scalarstagediag`	Scalar stage diagnostic output

3. Verification

L4YAML employs a three-layer verification strategy that combines formal proofs, compile-time checks, and runtime tests to achieve comprehensive coverage of the YAML 1.2.2 specification.

3.1. Three-Layer Strategy

3.2. Key Theorems

3.3. Key Proof Modules

3.4. What L4YAML Proves

3.5. Mind the Fibration Gap

3.6. Fibration Gap — Worked Example

3.7. Proof Engineering Patterns

3.8. Zero-Axiom Policy

3.6. Fibration Gap — Worked Example

`EndToEndCorrectness` ships two soundness headlines that differ only in how much of the fibration they expose. They are an exact demonstration of the alignment rule.

3.6.1. `parse_sound_shallow` — fibration-hidden

```
theorem parse_sound_shallow (input : String) (docs : Array YamlDocument)
    (h : TokenParser.parseYaml input = .ok docs) :
    ValidYamlProp input docs
```

- *Type* mentions `parseYaml`, `YamlDocument`, `ScanError`, and `ValidYamlProp`. The pipeline stages (`scanFiltered`, `parseStream`, `YamlDocument.compose`) are hidden inside the existential body of `ValidYamlProp`.
- *Proof* is four tactics: `unfold; split; injection; …`. No project theorem is cited.
- *Consequence*: condition 1 of the alignment rule fails at every root function, so the chain walker returns zero functorial chains. The `ChainDepth` classifier tags this theorem `propBridge` — recognisable by its Prop-typed target — and renders a diagnostic overlay labelled "NOT proof deps of the headline."

3.6.2. `parse_sound_deep` — fibration-aligned

```
theorem parse_sound_deep (input : String) (docs : Array YamlDocument)
    (h : TokenParser.parseYaml input = .ok docs) :
    ∃ (tokens : Array (Positioned YamlToken))
      (raw_docs : Array YamlDocument),
      Scanner.scanFiltered input = .ok tokens ∧
      TokenParser.parseYamlRaw input = .ok raw_docs ∧
      TokenParser.parseStream tokens = .ok raw_docs ∧
      docs = raw_docs.map YamlDocument.compose ∧
      (∀ doc ∈ docs.toList, ∃ node : ValidNode,
         stripAnnotations (toYamlValue node) = stripAnnotations doc.value)
```

- *Type* mentions `parseYaml`, `parseYamlRaw`, `parseStream`, `scanFiltered`, `YamlDocument.compose`, `toYamlValue`, and `ValidNode` directly. Every pipeline stage is a first-class citizen of the about-fibration.

- *Proof* cites `parseYamlRaw_ok_decompose` (about `parseYamlRaw`, `scanFiltered`, `parseStream`) and `parseYaml_produces_valid_nodes` (about `parseYaml`, `toYamlValue`), the second of which transitively drags in `parseStream_output_grammable` (about `parseStream`).
- *Consequence*: conditions 1, 2, and 3 of the alignment rule close simultaneously at multiple root functions. The `ChainDepth` classifier tags this theorem `deep`.

3.6.3. Chain-Analysis Confirmation

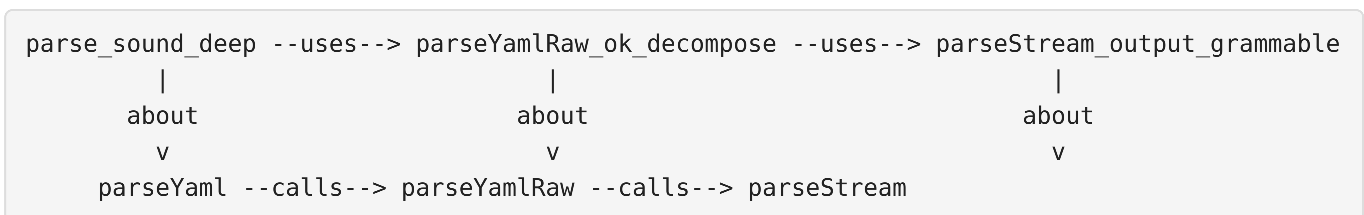
Running the FGM `theoremgraph` tool in `--chain` mode on both theorems produces the following metrics (tool output captured on 2026-04-24).

Metric	**`parse_sound_shallow`**	**`parse_sound_deep`**
Project theorems on chain	0	4 (root + 3 proof deps)
`uses` edges (theorem-fibration steps)	0	4
`calls` edges (function-fibration steps)	0	9
Aligned `about` edges	0	12
`ChainDepth` classification	`propBridge`	`deep`

The proof-dep theorems surfaced on `parse_sound_deep` are:

- `parseYamlRaw_ok_decompose` (module `Composition`) — splits a successful `parseYamlRaw` into a successful `scanFiltered` step and a successful `parseStream` step
- `parseYaml_produces_valid_nodes` (module `ParserGrammable`) — bridges `parseYaml` success to per-document `ValidNode` witnesses
- `parseStream_output_grammable` (module `ParserGrammable`) — guarantees every `parseStream`-produced document is grammable

Each of these is *about* a strict callee of `parseYaml`, so the alignment square closes at every step of the descent:



```
parse_sound_deep --uses--> parseYamlRaw_ok_decompose --uses--> parseStream_output_grammable
        |                          |                                   |
      about                      about                               about
        v                          v                                   v
     parseYaml --calls--> parseYamlRaw --calls--> parseStream
```

`parse_sound_shallow`'s chain graph, by contrast, is a diagnostic overlay: it enumerates the catalogue entries that *would* be relevant if the alignment existed, and the legend explicitly labels those nodes

"NOT proof deps of the headline." The absence of any `uses` or `calls` edge is the fibration gap made visible.

Takeaway: a theorem passing the kernel is a statement about correctness; a theorem passing the chain walker is additionally a statement about *traceability*. The two are not the same. The `deep` / `propBridge` / `weak` / `noAbout` classification in `FGM.TheoremGraph` is the knob that surfaces this distinction across the catalogue.

3.3. Key Proof Modules

Module	Theorems	Scope
`ScannerCorrectness.lean`	259 theorems + 1,063 guards	Character-to-token correctness for all scanner operations
`Completeness.lean`	63 theorems	Valid YAML inputs are accepted
`Soundness.lean`	28 theorems	Output corresponds to valid grammar derivations
`RoundTrip.lean`	58 theorems + 63 guards	Parse-emit-parse cycle properties
`Composition.lean`	12 theorems	Scanner + parser pipeline correctness
`ScannerEmitBridge.lean`	12 theorems + 64 guards	Bridge between scanner emissions and grammar predicates
`ParserSoundness.lean`	12 theorems	Grammar-to-implementation correspondence
`ParserWfaProofs.lean`	50 theorems	Well-formed anchors through entire parser pipeline
`ParserNodeProofs.lean`	57 theorems	Anchor growth and alias resolution
`ParserWellBehaved.lean`	102 theorems	Flow nesting, token preservation, scannable properties
`ScannerProgress.lean`	24 theorems	Offset strictly increases on every scanner step
`ScannerSimpleKey.lean`	Multiple theorems	Simple key lifecycle well-formedness
`ScannerDispatch.lean`	Multiple theorems	Dispatch pipeline preserves all invariants

3.2. Key Theorems

The following capstone theorems represent the main formal guarantees established by L4YAML. Each is machine-checked by the Lean kernel with zero axioms beyond the built-in foundations.

Each theorem is accompanied by a bipartite dependency graph showing the implementation functions it proves properties about (blue, left) and the supporting theorems used in its proof (green, right). Stdlib lemmas surface in orange when they are domain-relevant.

3.2.1. Pipeline Composition

These theorems establish that the scanner and parser compose correctly into a single end-to-end pipeline.

Theorem	Module	Statement
`parseYaml_pipeline`	`Composition`	End-to-end: scan then parse composes correctly. If `scanFiltered` produces tokens and `parseStream` accepts them, then `parseYaml` succeeds with the same result.
`parseYamlRaw_pipeline`	`Composition`	Raw pipeline variant without schema resolution.
`parseYamlRaw_ok_decompose`	`Composition`	Every successful `parseYamlRaw` result decomposes into a successful scan step followed by a successful parse step.
`parseYaml_ok_iff`	`Completeness`	`parseYaml` succeeds if and only if the input is valid YAML — the bridge between the implementation and the specification.

3.2.1.1. `parseYaml_pipeline`

parseYaml_pipeline dependency graph

3.2.1.2. `parseYamlRaw_pipeline`

parseYamlRaw_pipeline dependency graph

3.2.1.3. `parseYamlRaw_ok_decompose`

parseYamlRaw_ok_decompose dependency graph

3.2.1.4. `parseYaml_ok_iff`

parseYaml_ok_iff dependency graph

3.2.2. Scanner Correctness

Properties of the character-to-token scanner:

Theorem	Module	Statement
`scan_produces_valid_tokens`	`ScannerCorrectness`	The scanner output satisfies `ValidTokenStream`: every emitted token is well-formed, positions are monotonically increasing, and the stream is bracketed by `STREAM_START`/`STREAM_END`.
`advance_offset_lt`	`ScannerProgress`	Scanner advance *strictly* increases the byte offset when the offset is within bounds — this is the core termination lemma.
`scanLoop_success_emits_streamEnd`	`ScannerCorrectness`	A successful scan loop always terminates with a `STREAM_END` token.

3.2.2.1. `scan_produces_valid_tokens`

scan_produces_valid_tokens dependency graph

3.2.2.2. `advance_offset_lt`

advance_offset_lt dependency graph

3.2.2.3. `scanLoop_success_emits_streamEnd`

scanLoop_success_emits_streamEnd dependency graph

3.2.3. Parser Correctness

Properties of the token-to-AST parser:

Theorem	Module	Statement
`parseStream_sound`	`ParserSoundness`	If the parser produces an AST, it corresponds to a valid YAML grammar derivation.
`parseNode_anchors_grow`	`ParserNodeProofs`	The anchor set grows monotonically through `parseNode` — anchors are never lost during parsing.
`parseNode_aliases_resolve'`	`ParserNodeProofs`	Every alias (`*name`) in the output of `parseNode` resolves to a previously defined anchor (`&name`).
`parseStream_output_anchors_wellformed`	`ParserWfaProofs`	After `parseStream` completes, all output anchors are well-formed: every alias target exists and every anchor body is `Grammable`.

3.2.3.1. `parseStream_sound`

parseStream_sound dependency graph

3.2.3.2. `parseNode_anchors_grow`

parseNode_anchors_grow dependency graph

3.2.3.3. `parseNode_aliases_resolve'`

parseNode_aliases_resolve' dependency graph

3.2.3.4. `parseStream_output_anchors_wellformed`

parseStream_output_anchors_wellformed dependency graph

3.2.4. Soundness

Theorems establishing that the AST-to-value conversion faithfully implements the YAML specification:

Theorem	Module	Statement
`toYamlValue_correct`	`Soundness`	The `toYamlValue` function correctly implements the specification's construction rules.
`nodeToValue_total`	`Soundness`	Every well-formed AST node can be converted to a `YamlValue` — the conversion is total.
`nodeToValue_deterministic`	`Soundness`	AST-to-value conversion is deterministic: the same input always produces the same output.
`scalar_content_preserved`	`Soundness`	Scalar string content is preserved through the parsing pipeline — no characters are added, dropped, or reordered.

3.2.4.1. `toYamlValue_correct`

toYamlValue_correct dependency graph

3.2.4.2. `nodeToValue_total`

nodeToValue_total dependency graph

3.2.4.3. `nodeToValue_deterministic`

nodeToValue_deterministic dependency graph

3.2.4.4. `scalar_content_preserved`

scalar_content_preserved dependency graph

3.2.5. Round-Trip Properties

Theorems about the parse → emit → parse cycle:

Theorem	Module	Statement
`contentEq_refl`	`RoundTrip`	Content equality is reflexive: every YAML value is content-equal to itself.
`contentEq_symm`	`RoundTrip`	Content equality is symmetric.
`contentEq_trans`	`RoundTrip`	Content equality is transitive — together with reflexivity and symmetry, this makes `contentEq` an equivalence relation.
`emit_content_invariant`	`ScannerEmitBridge`	The emitter preserves content equality: if two values are content-equal, their emitted canonical forms are identical.
`escapeTag_roundtrip`	`RoundTrip`	Tag URI escape and unescape are inverse operations.

3.2.5.1. `contentEq_refl`

contentEq_refl dependency graph

3.2.5.2. `contentEq_symm`

contentEq_symm dependency graph

3.2.5.3. `contentEq_trans`

contentEq_trans dependency graph

3.2.5.4. `emit_content_invariant`

emit_content_invariant dependency graph

3.2.5.5. `escapeTag_roundtrip`

escapeTag_roundtrip dependency graph

3.5. Mind the Fibration Gap

When you read "this theorem is machine-checked," two follow-up questions are worth asking:

1. *How significant is this theorem?* Is it a deep statement about how the code behaves, or a shallow observation about the code's shape?
2. *Would this theorem break if we changed the code?* If a refactor silently broke the parser, would the kernel catch it — or would the theorem quietly survive the change?

These are not philosophical questions. They have a mechanical, *kernel-objective* answer that is read directly off the proof term the kernel has already accepted. The answer comes from stitching together two natural bipartite graphs —

- the *function call graph* (`f` calls `f'`), and
- the *theorem dependency graph* (`T` cites `T'` in its proof) —

using the *about* relation (the theorem `T` mentions the function `f` in its type) as the bridge between them. A theorem that threads all three relations tightly together is said to *fibrate* over the call graph; one that does not leaves a *fibration gap*.

Two natural fibrations run through every proof module, corresponding to the three relations above:

- *The "about" fibration* — each theorem projects to the set of project functions mentioned in its type. This is what `#about_functions` reports.
- *The "uses" fibration* — each theorem projects to the set of other project theorems cited in its proof term. This is what `#proof_deps` reports.

A theorem is *deep* when the two fibrations align: the lemmas cited in its proof are themselves about the callees of the functions the root theorem is about. The `FGM.ExploreGraph.findFunctorialChains` analyzer walks this alignment one step at a time, and the `ChainDepth` classifier tags the result as `deep`, `propBridge`, `weak`, or `noAbout`.

3.5.1. Why Fibration Matters: The Canary Theorem

A deeply fibrated theorem acts as a *canary*. Because its proof term cites specific lemmas about specific callees, any code change that invalidates the underlying behaviour invalidates those lemmas, which invalidates the theorem, which the kernel then refuses to re-accept. The build breaks at the theorem that pinned the claim, pointing directly at the layer of the pipeline whose behaviour has shifted. The deeper the fibration, the more callees are pinned, and the more sensitive the canary.

A weak, unfibrated theorem gives no such signal. Because its proof never descends into the callees, it can survive code changes that genuinely affect parser behaviour. `parse_sound_shallow` is a working example: its statement asserts only that *some* token stream exists for which the pipeline decomposition equation holds — it says nothing about which tokens came out of the scanner, nor what the parsed documents actually contain. A refactor that shuffles the parser's internal decision tree, or silently corrupts the emitted document content, can leave `parse_sound_shallow` intact and the kernel silent. Strong behavioural guarantees must come from somewhere else — from the underlying parser-correctness lemmas, or from a canary like `parse_sound_deep` that explicitly cites them.

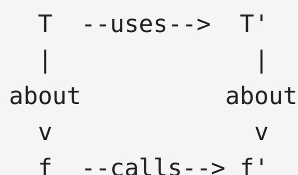
This does not mean structurally weak theorems are worthless. They may still carry value — as witnesses that two definitions are definitionally equal, as type-class coherence lemmas, as rewriting hints, or as quick sanity checks on API shape. Those uses are legitimate, but they answer a different question than fibration does. *For the specific question "does this theorem catch breaking code changes?", the fibration structure is the right instrument*, and it is the one this document measures.

3.5.2. The Alignment Rule

A chain link `(T, f)` — a root theorem `T` paired with a function `f` that `T` is about — extends to `(T', f')` iff *all three* conditions hold:

1. `T' ∈ thmDeps(T)` — `T'` is a project theorem cited in the proof term of `T`
2. `f' ∈ about(T')` — the type of `T'` mentions `f'`
3. `f' ∈ fnDeps(f)` — `f`'s body calls `f'`

Informally, the square



```
    T  --uses-->  T'
    |              |
  about          about
    v              v
    f  --calls--> f'
```

must commute. Each arrow corresponds to one edge colour in the bipartite graph — green for `uses`, blue for `calls`, purple for `about`. Alignment requires *three* simultaneous edges at every step, which is exactly categorical base-change: the functorial chain is the lifting of the call-graph path `f → f'` along the about-fibration, guided by the proof-chain `T → T'`.

3.5.3. What Breaks Alignment

- *Prop-wrapping* — stating the theorem as `… → SomeProp args` pushes the about-fibration into the definition body of `SomeProp`. `collectAbout` only inspects the raw type, so the analyzer sees `SomeProp` as a single opaque target. The three-way alignment collapses before step 1.
- *Tactic-only proofs* — `unfold; split; injection`, `rfl`, `grind`, `decide`, and friends produce proof terms with no project-theorem citations. Condition 1 fails trivially and no chain extends.
- *Co-location* — citing only lemmas that are about the *same* top-level function the root is about. The alignment square collapses to the identity on the function side: condition 3 fails because no new call edge is exposed.

3.5.4. The Engineering Rule

When engineering a headline theorem intended for dependency-analysis consumption:

1. *Expose the fibration in the type*. Name the pipeline functions explicitly in the statement; avoid `Prop`-wrappers that hide them.
2. *Cite lemmas about callees*. Every project lemma used in the proof should itself be about a function that is a strict callee of the function the root theorem is about.
3. *Prefer a step-wise proof over a tactic blob*. Each `have ⟨…⟩ := lemma …` contributes one functorial step; each uncited `unfold`/`split`/`grind` discards one.

3.7. Proof Engineering Patterns

Several patterns emerged during the verification effort:

- *Decomposition* — large functions are decomposed into validation (error guards), state transformation (pure updates), and emission (token output) phases, each proved independently then composed.
- *Append-only invariant* — the switch from `insertAt` to placeholder reservation slots with `setIfInBounds` backpatching eliminated the hardest class of proof obligations (index shifting).
- *Monotonic progress* — proving `offset_lt` (strict increase) for every scanner operation provides termination and guarantees no infinite loops.
- *Well-formedness threading* — a `WellFormed` predicate on scanner state is threaded through every operation, establishing that invariants are maintained from `scannerInit` through `scanNextToken` to stream completion.
- *Anchor monotonicity* — the `AnchorsGrow` relation is proved transitively across all 14 mutually recursive parser functions, establishing that anchors accumulate but are never dropped.
- *Fuel-based termination* — the parser's 14 mutual functions use `fuel : Nat` as a decreasing argument. Initial fuel is set to `4 * tokens.size + 4`, large enough for any valid input.

3.1. Three-Layer Strategy

3.1.1. Layer 1: Machine-Checked Proofs

The core layer consists of 2,770 Lean 4 theorems across 63 proof modules (63,684 lines). These proofs are checked by the Lean kernel — the small trusted core of the system — and establish properties including:

- **Soundness** — every token stream produced by the scanner corresponds to a valid YAML grammar derivation
- **Completeness** — every valid YAML input is accepted (not rejected with an error)
- **Progress** — the scanner's input offset strictly increases on every step, guaranteeing termination
- **Well-formedness preservation** — internal invariants (indentation stack consistency, flow level balance, simple key lifecycle) are maintained across all scanner operations
- **Pipeline composition** — scanner and parser compose correctly to deliver end-to-end guarantees

3.1.2. Layer 2: Compile-Time Guards

2,124 `#guard` statements are evaluated by the Lean kernel at build time. These are not runtime tests — they are *kernel-evaluated assertions* that must hold for the project to compile. A failing `#guard` is a build error, not a test failure.

Guards are used extensively for:

- Concrete scanner behavior on specific inputs
- Round-trip properties (parse → emit → parse = original)
- Token stream structure for specification examples
- Character predicate boundary conditions

3.1.3. Layer 3: Runtime Tests

1,041 runtime tests across 19 suites provide additional coverage:

- *Specification examples* — all 132/132 YAML 1.2.2 examples

- *yaml-test-suite* — 225/225 applicable test IDs (354/406 total; 52 YAML 1.1/1.3 tests are correctly skipped)
- *Property tests* — randomized input generation for edge cases
- *Mutation tests* — systematic input perturbation
- *Adversarial tests* — handcrafted inputs targeting parser limits
- *Round-trip tests* — parse → dump → parse cycle validation

3.4. What L4YAML Proves

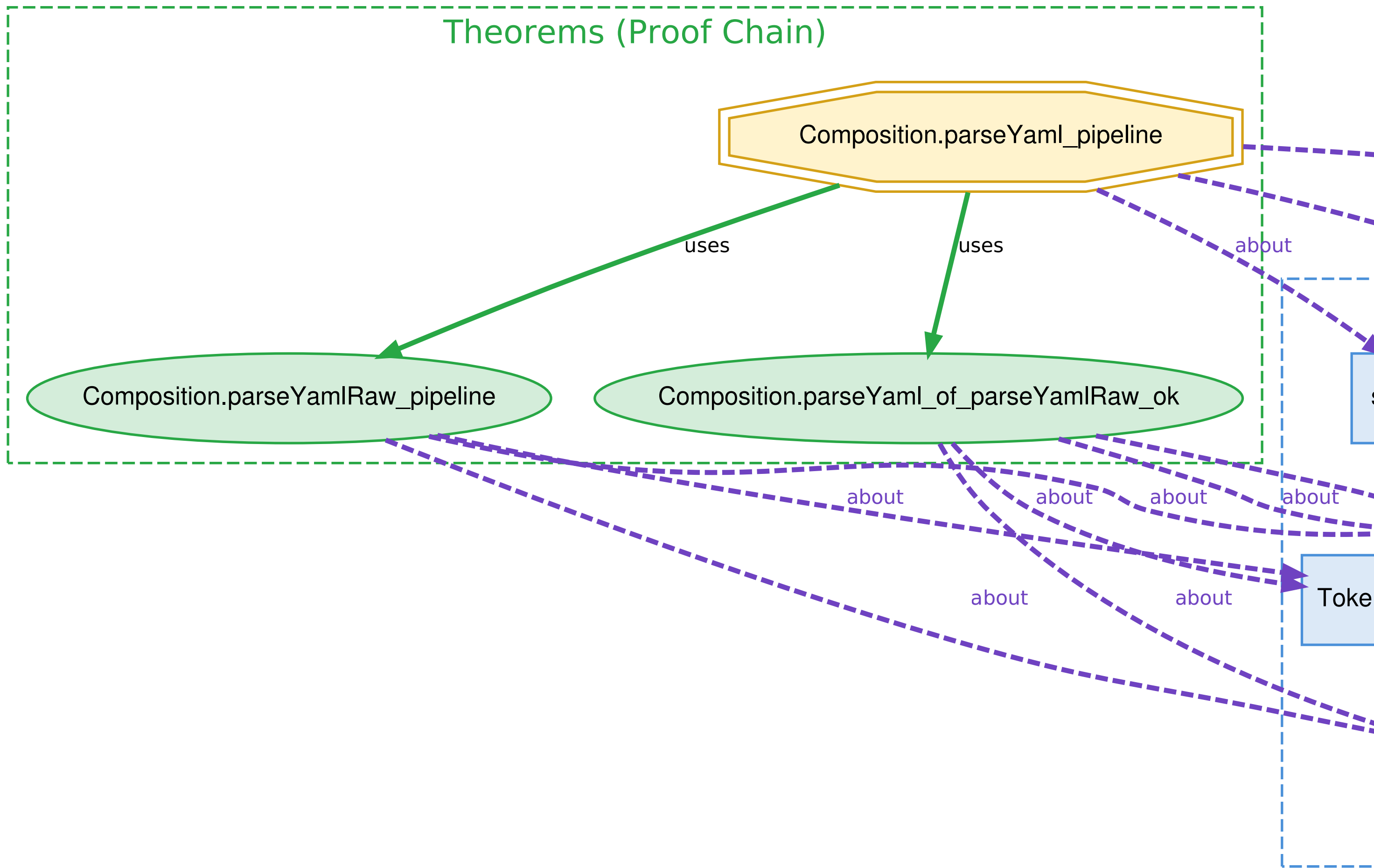
This page is the front-door answer to *what L4YAML proves*. Each card below embeds one headline theorem's *functorial chain graph* — the proof-term dependency structure extracted by the FGM `theoremgraph` tool — together with its plain-English summary and a ChainDepth tag that classifies how tightly the theorem's proof is pinned to the pipeline functions it names. The ChainDepth tags carry the honesty labelling from §Mind the Fibration Gap: *deep* theorems are canaries for silent behavioural drift, *propBridge* theorems delegate that role to a wrapped predicate, and *weak* theorems do not participate in the fibration at all.

Only the headline + categoryCapstone entries (~11 theorems) are embedded here; the full catalogue is published as the `theorem-graphs-all.tar.gz` asset on the `graphs-latest` release of L4YAML.FGM (https://github.jpl.nasa.gov/pass/L4YAML.FGM/releases/tag/graphs-latest) and is downloadable on demand. Per-module hierarchical browsing of the full catalogue inside this site is a planned step.

3.4.1. Pipeline Composition

3.4.1.1. `parseYaml_pipeline` — deep

1.1 — `parseYaml` decomposes as `parseStream ∘ scanFiltered`. The pipeline's decomposition capstone: every downstream soundness/completeness headline routes through it.

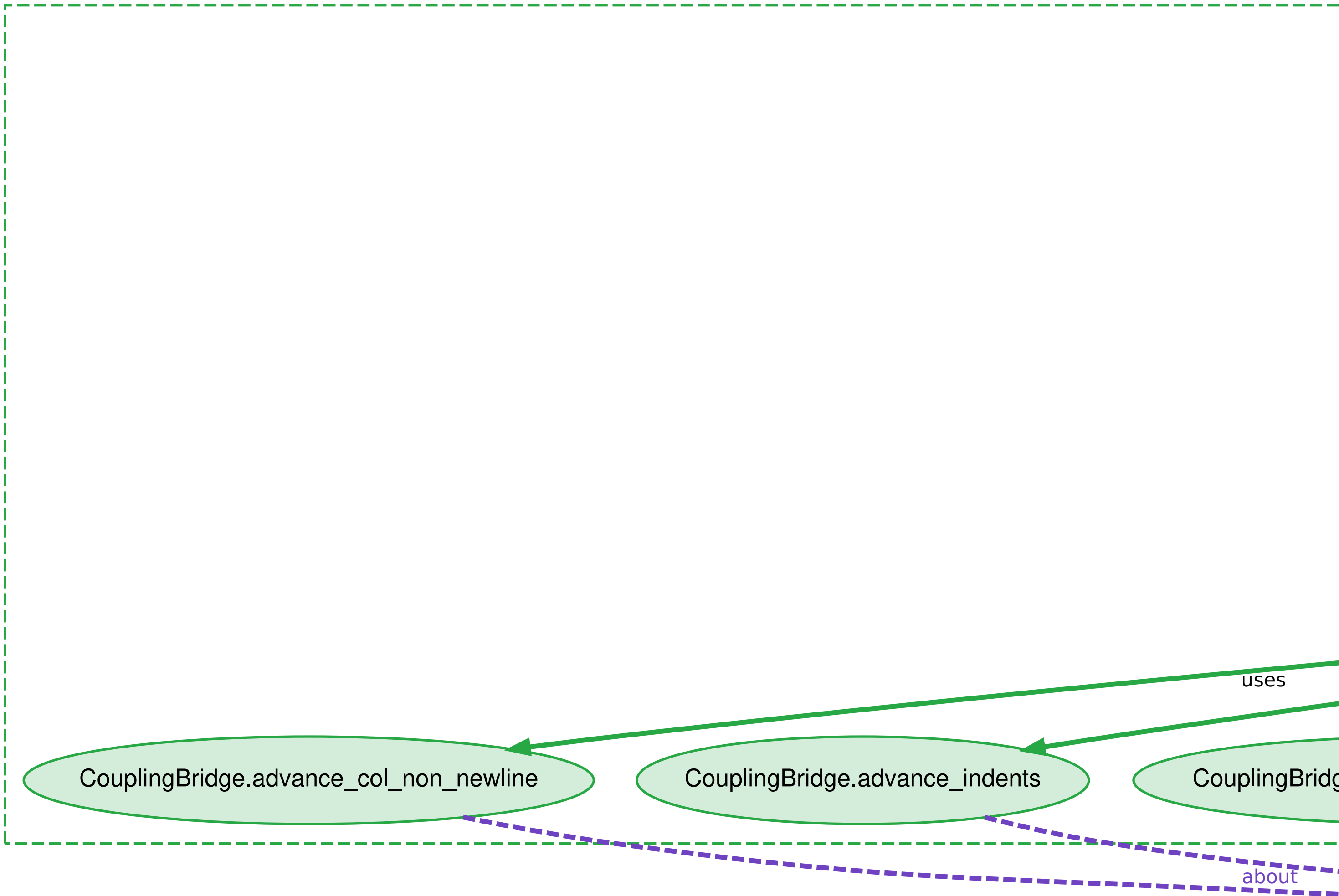


- *ChainDepth*: `deep`
- *Module*: `L4YAML.Proofs.Composition`

3.4.2. Scanner

3.4.2.1. `scan_full_consumption` — deep

2.1 — a successful scan consumes the entire input. Rules out the common correctness trap where a scanner returns `.ok` after stopping early on an unexpected character.

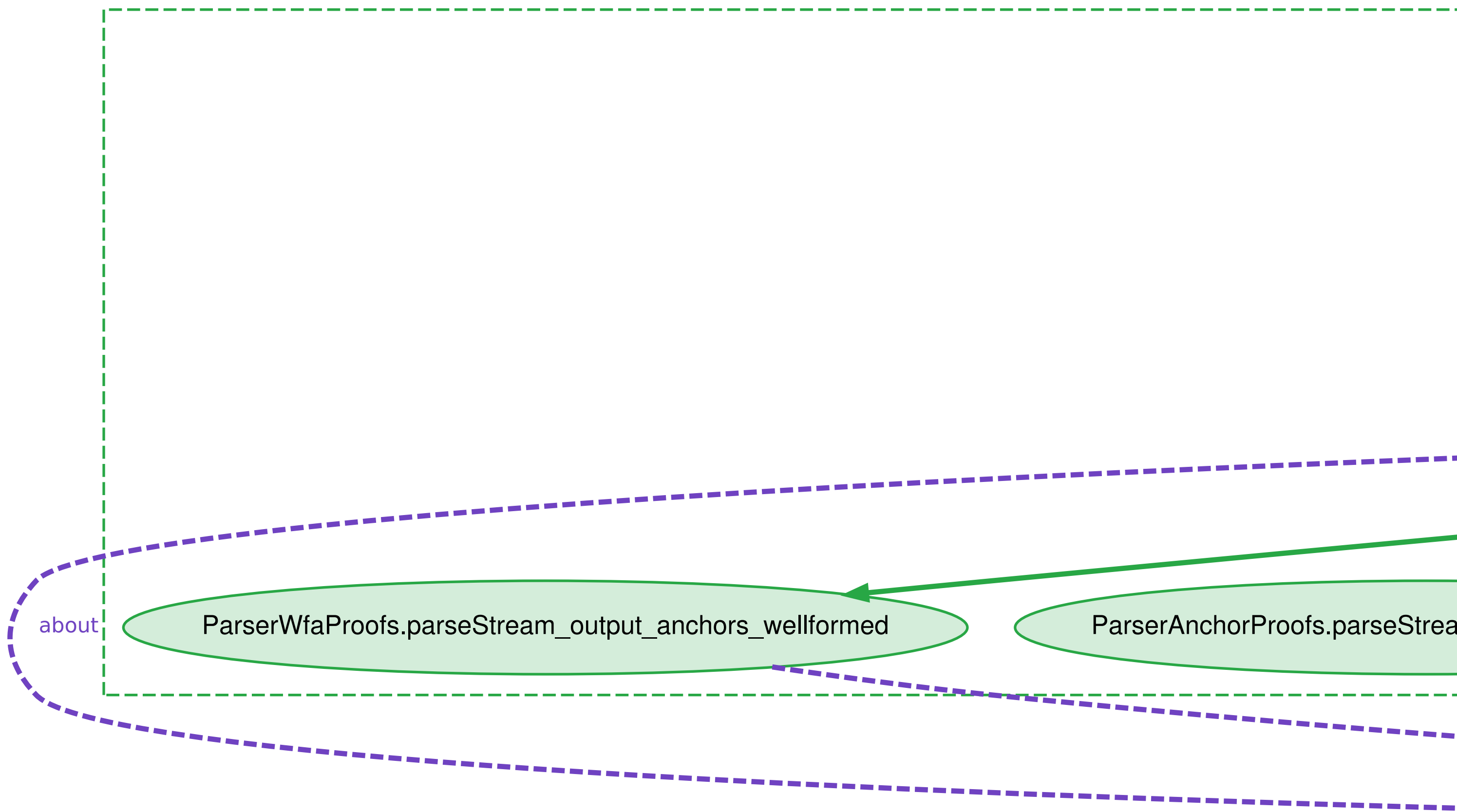


- *ChainDepth*: `deep`
- *Module*: `L4YAML.Proofs.Scanner.ScanStrictCoupling`

3.4.3. Parser

3.4.3.1. `parseStream_respects_grammar_unconditional` — deep

3.1 — the parser respects the YAML 1.2.2 grammar with no well-formedness precondition. The root anchor for the grammar- production chain.



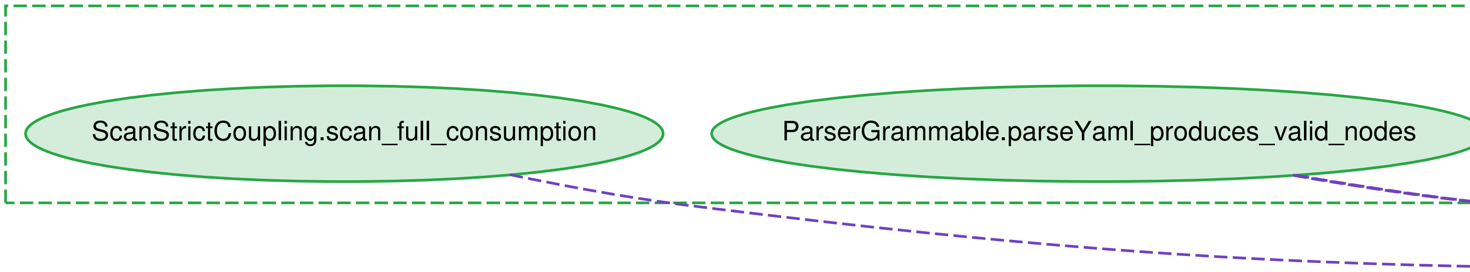
- *ChainDepth*: `deep`
- *Module*: `L4YAML.Proofs.EndToEndCorrectness`

3.4.4. End-to-End Correctness

The two soundness variants (`parse_sound_shallow` and `parse_sound_deep`) are the fibration-gap worked example — see §Fibration Gap — Worked Example for the side-by-side walkthrough.

3.4.4.1. `parse_sound_shallow` — propBridge

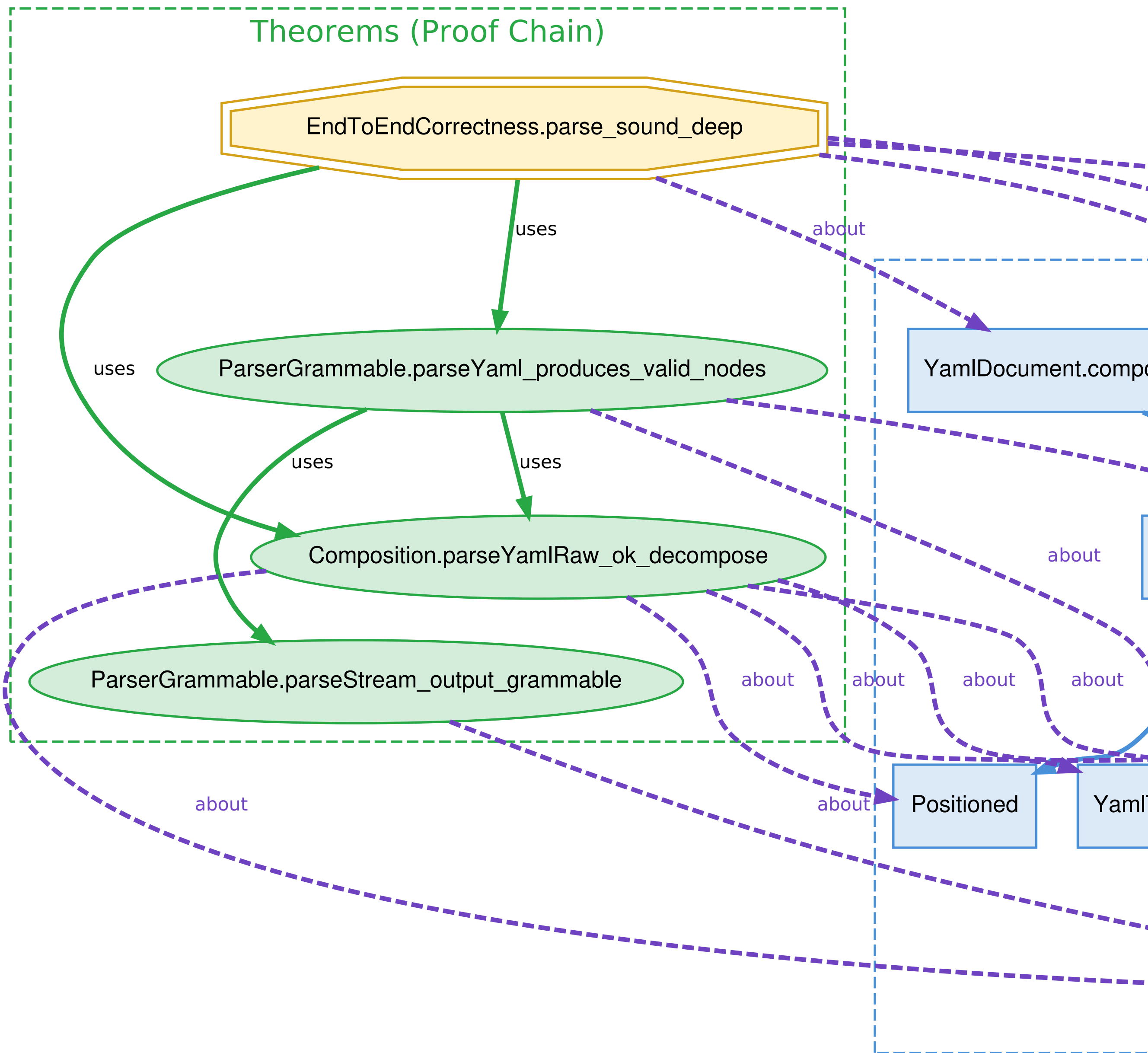
4.1 — `parse .ok → ValidYamlProp`. The shallow variant hides the pipeline stages inside the `ValidYamlProp` predicate; the chain walker cannot descend. Useful as a minimal soundness statement; not the canary.



- *ChainDepth*: `propBridge`
- *Module*: `L4YAML.Proofs.EndToEndCorrectness`

3.4.4.2. `parse_sound_deep` — deep

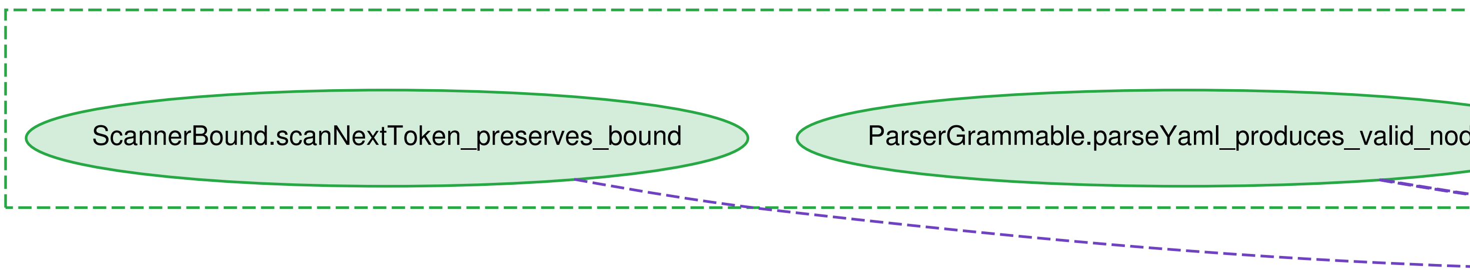
4.1d — same soundness claim with every pipeline stage exposed in the type and each stage's lemma cited in the proof. The canary for silent code changes along `parseYaml → parseYamlRaw → parseStream`.



- *ChainDepth*: `deep`
- *Module*: `L4YAML.Proofs.EndToEndCorrectness`

3.4.4.3. `parse_complete` — propBridge

4.2 — `ValidYamlProp → parse .ok`. Completeness with the same `Prop`-wrapping shape as `parse_sound_shallow`; an engineering follow-up analogous to `parse_sound_deep` would expose the pipeline stages and cite the completeness lemmas directly.

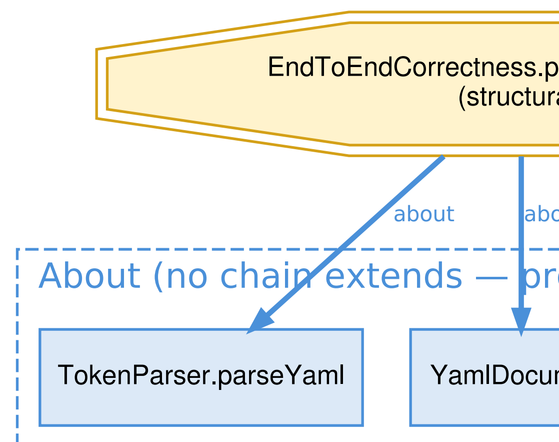


- *ChainDepth*: `propBridge`
- *Module*: `L4YAML.Proofs.EndToEndCorrectness`

3.4.4.4. `parse_deterministic` — weak

4.3 — `parse` is a function (same input, same output). Structurally weak: the proof is tactic-only and cites no project lemmas, so no functorial chain exists. Serves as a type-level sanity check on the API shape, not as a canary.

Structural theorem: EndToEndCorrectness.parse_deterministic\nProof is a witness / in

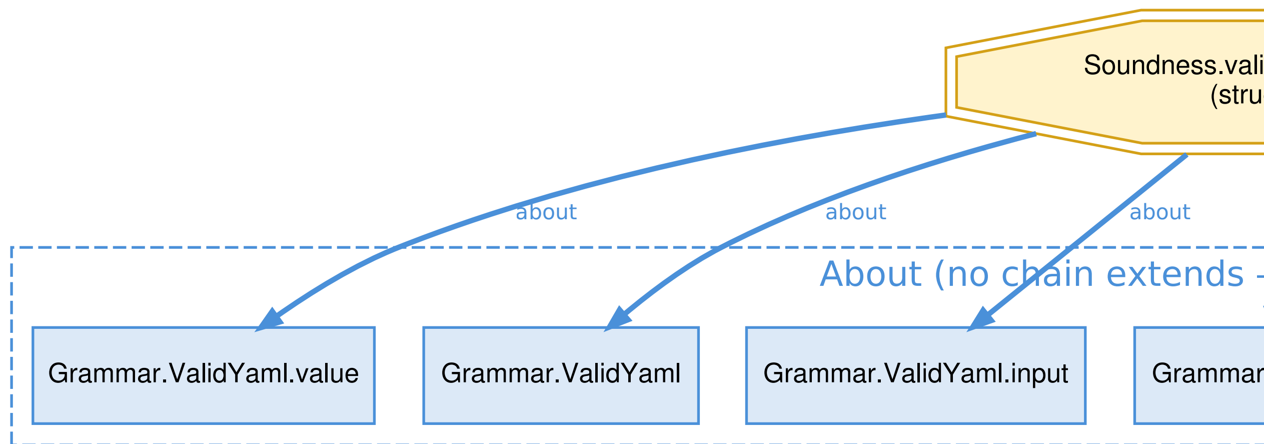


- *ChainDepth*: `weak` (structural)
- *Module*: `L4YAML.Proofs.EndToEndCorrectness`

3.4.5. Values — Soundness

3.4.5.1. `validYaml_construct` — weak

5.1 — constructing `ValidYaml` from a parse result is well-defined. Establishes that the packaging step from `docs` to `ValidYaml` respects the underlying invariants; the proof is structural and carries no fibration signal.



- *ChainDepth*: `weak` (structural)
- *Module*: `L4YAML.Proofs.Soundness`

3.4.6. Roundtrip

3.4.6.1. `universal_roundtrip` — deep (🚧 sorry-reachable via 6.9)

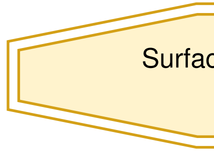
6.1 — the universal YAML round-trip property: emit ∘ parse is content-preserving up to `contentEq`. Fibration-aligned against the emitter/parser composition; carries a sorry marker via 6.9 pending the final stage's proof.

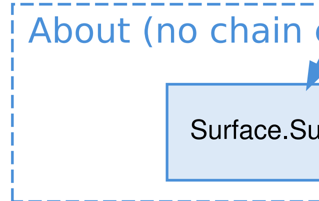
3.4.7. Grammar Production

3.4.7.1. `parse_strict_proof` — deep (🚧 sorry-reachable via 7.2, 7.6)

7.1 — parser acceptance implies `InYamlLanguage`. Bridges the parser-level chain to the grammar-production relation used by the Blueprint. Sorry-reachable via the scanner-side 7.2/7.6 lemmas.

Structural theorem: SurfaceCoupling.SIndent_zero\nProof is a witness / injection — no





- *ChainDepth*: `weak` (structural)
- *Module*: `L4YAML.Proofs.Coupling.SurfaceCoupling`

3.4.9. Generating graphs locally

The `theoremgraph` tool lives in the [L4YAML.FGM](https://github.jpl.nasa.gov/pass/L4YAML.FGM) (https://github.jpl.nasa.gov/pass/L4YAML.FGM) bridge project; see its `tools/README.md` for invocations. Common starting points:

```
cd ../L4YAML.FGM
lake build theoremgraph
lake exe theoremgraph --list                          # catalogue
lake exe theoremgraph --chain parse_sound_deep        # one chain DOT
lake exe theoremgraph --tier headline tmp/out         # headline tarball contents
lake exe theoremgraph tmp/graphs                      # full tree (≈400 DOTs)
```

3.8. Zero-Axiom Policy

The project uses zero axioms beyond Lean's built-in foundations (`propext`, `Quot.sound`, `Classical.choice`). No `sorry` appears anywhere in the codebase. No `partial def` is used — every function has a kernel-checked termination proof.

This means the formal guarantees are as strong as the Lean kernel itself: if the kernel accepts the proofs, the properties hold.

Welcome to the documentation page

This was built using Lean 4 4.30.0-rc2 (https://github.com/leanprover/lean4/tree/3dc1a088b6d2d8eafe25a7cd7ec7b58d731bd7cc)

lean4-yaml-verified Test Coverage

YAML 1.2.2 verified parser · Lean 4

yaml-test-suite Compliance

```
Total unique tests:      406
Applicable (YAML 1.2.2): 358  (48 skipped — YAML 1.3 specific)
Passed:                  263
Failed:                  0
Expected fail:           95
Unexpected pass:         0
Timeout:                 0

Correct: 358/358 (100.0%)
```

Reports

Full Coverage Report
All 406 unique test cases with filtering & sorting

Coverage by Stage

scalar
58/58 correct (100.0%) · 24 YAML 1.3 skipped
flow
46/46 correct (100.0%)
block
99/99 correct (100.0%)
document
17/17 correct (100.0%) · 7 YAML 1.3 skipped
advanced
64/64 correct (100.0%) · 17 YAML 1.3 skipped
error
74/74 correct (100.0%)

Suite Runner (Progressive Stages)

✓ Suite Runner: 869/869 (100.0%)

Each stage tests its own tests plus all lower stages (cumulative). Error tests excluded.

Stage	Total	Passed	Failed	Skipped
scalar	82	**58**	0	24
flow	128	**104**	0	24
block	227	**203**	0	24
document	251	**220**	0	31
advanced	332	**284**	0	48
Total	**1020**	**869**	**0**	**151**

Verified Tests

✓ Internal Verified Test Suites (4243/4243)

Unit Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/Main.lean)	✓ 10/10
Explicit Key Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/ExplicitKeyTests.lean)	✓ 149/149
Flow Completeness Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/FlowTests.lean)	✓ 88/88
Structural Validation Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/ValidationTests.lean)	✓ 84/84
Dump Round-Trip Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/DumpRoundTrip.lean)	✓ 117/117
Raw Parse / Compose Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/RawParseTests.lean)	✓ 29/29
YAML 1.2.2 Spec Examples (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/SpecExamples.lean)	✓ 132/132
Schema ↔ Dump Integration (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/SchemaDump.lean)	✓ 68/68
Scanner & TokenParser Tests (Phase 9) (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/ScannerTests.lean)	✓ 32/32
YAML 1.2.2 Spec Examples (Scanner/Parser Pipeline) (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/ScannerSpecExamples.lean)	✓ 132/132
Adversarial Grammar-Directed Tests (v0.2.13.1) (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/AdversarialGrammarTests.lean)	✓ 154/154
Adversarial Instantiation Tests (sorry audit) (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/AdversarialInstantiation.lean)	✓ 2441/2441
Mutation Testing on yaml-test-suite (v0.2.13.2) (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/MutationSuiteTests.lean)	✓ 45/45
Property-Based Round-Trip Tests (v0.2.13.3) (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/PropertyTests.lean)	✓ 124/124
Production Coverage Analysis (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/ProductionCoverage.lean)	✓ 588/588
Parser Security Limit Tests (https://github.com/nasa-jpl/L4YAML/blob/main/Tests/LimitTests.lean)	✓ 43/43
Demo (https://github.com/nasa-jpl/L4YAML/blob/main/Demo.lean)	✓ 7/7
Total	**4243/4243**

Verified Tests Detail

4243/4243 tests across 17 suites

YAML 1.2.2 Production Coverage

All productions cross-referenced with @[yaml_spec] annotations

Generated by yaml-test-suite (https://github.com/yaml/yaml-test-suite) runner · Lean 4