

Formalizing a Charting DSL:

ANTLR4 Grammar Engineering and ASDL Intermediate Representations for Pine Script™

HOOX Research / PYNE Contributors
jango_blockchained
Independent Open Source Research
contact@hoox.sh

Abstract

Domain-specific languages (DSLs) for financial charting combine expression-oriented semantics with significant whitespace, versioned surface syntax, and a dense builtin surface inventory. Building reliable tooling—parsers, formatters, linters, language servers, and interpreters—requires a front-end that is both faithful to community scripts and amenable to static analysis. We present the language front-end of PYNE (package `hoox-pyne`), an independent open toolchain for Pine Script™: a hand-engineered ANTLR4 grammar with Python-side indent tracking, a Zephyr-ASDL abstract syntax tree (AST) schema, a visitor-based parse-tree builder, a precedence-aware unparser, a structural linter, and a qualifier-aware type model. The pipeline is

source $\rightarrow$ FileStream/InputStream $\rightarrow$ Lexer $\rightarrow$ CommonTokenStream $\rightarrow$ Parser $\rightarrow$ parse tree $\rightarrow$ AST builder $\rightarrow$ A

Hand-edited artifacts are limited to `antlr4/resource/*.g4` and `asdl/resource/Pinescript.asdl`; generated lexer/parser and node classes are committed so end users need no Java toolchain. We detail SLL-first / LL-fallback prediction ($\approx 5.4\times$ on complex scripts), annotation attachment for `//@version` and related markers, parse-cache design (SHA-256 LRU with call-site scrubbing; multi-parse $\approx 1000\times$ on hits), unparser thread-local reuse ($\approx 1.95\times$ on a 99-script corpus), corpus sanitization for scraped sources, and evaluation against 138+ real scripts with 1100+ automated tests in the broader project. The grammar is approximate and unofficial: PYNE is not affiliated with TradingView, Inc.

Keywords. grammar engineering, ANTLR, ASDL, abstract syntax trees, parse-unparse round-trip, domain-specific languages, Pine Script™

Disclaimer. Pine Script™ and TradingView® are trademarks of TradingView, Inc. PYNE is an independent, unofficial implementation and is not affiliated with, authorized by, sponsored by, or endorsed by TradingView, Inc.

1 Introduction

Charting and algorithmic trading communities rely on specialized languages that embed market data series, drawing primitives, and strategy execution into a compact surface syntax [Fowler, 2010, Hudak, 1996, TradingView, Inc., 2024]. Pine Script™ is one such language: scripts are short relative to general-purpose codebases yet dense in temporal operators, qualifiers (`const`, `simple`, `series`, `input`), user-defined types (UDTs), methods, and (from v6) enums and multiline strings. Tooling that merely tokenizes for editor highlighting is insufficient for format-on-save, structural search, type-directed completion, or faithful interpretation [Microsoft, 2023, Nystrom, 2021].

PYNE [HOOX Project, 2026] is an independent, AGPL-licensed implementation of a Pine Script™ toolchain in Python. This paper isolates the *language front-end*: everything from source text to a stable ASDL tree and back, including the grammar engineering practices that make the rest of the stack possible. Companion papers in the series address overall architecture, compilation to NUMBA, series runtime semantics, strategy parity, and LSP integration; here we focus on:

1. **Grammar design** for a whitespace-significant, expression-oriented charting DSL, including indent tokens, string forms, colors, and v6 surface extensions.
2. **Asdl intermediate representation** as a single source of truth for node shapes, shared by the builder, unparser, linter, evaluator, and compiler.
3. **Construction and unparsing** with annotation attachment and precedence-aware pretty-printing.
4. **Static analysis hooks**: a rule-based linter and a type model for qualifiers, collections, and UDTs.
5. **Corpus engineering**: sanitization of scraped community sources and a regeneration workflow that keeps generated artifacts reproducible.
6. **Evaluation**: parse success, round-trip stability, and performance tables for SLL/LL, unparse, and cache warm paths.

Contributions.

- A documented ANTLR4 grammar and lexer base for Pine Script™ v5/v6 surface features, with generated Python targets committed for zero-Java installs.
- An ASDL schema and visitor infrastructure analogous to CPython’s `ast` module [Python Software Foundation, 2024, Wang et al., 1997].
- A lossless-enough round-trip for structure and `//@`-annotations, powering CLI format and LSP document formatting.
- Measured front-end optimizations (SLL-first parse, annotation skip, thread-local unparser, SHA-256 parse cache) with correctness checks via AST dump identity and re-parse stability.

2 Background

2.1 Pine surface evolution (v1–v6)

Pine Script™ evolved from a narrow indicator dialect into a multiparadigm charting language [TradingView, Inc., 2024]. Early versions emphasized single-line expressions and a small set of drawing builtins. Later major versions introduced:

v3–v4

Multi-line scripts, `var` persistence, expanded `security` patterns, and broader strategy APIs.

v5

Namespaced builtins (`ta.*`, `math.*`, `strategy.*`, ...), libraries (`import` / `export`), methods, UDTs (`type`), `switch`, and richer type annotations on parameters.

v6

`enum` declarations; triple-quoted multiline strings (`"""..."""` / `'''...'''`); stricter boolean typing; integer division producing floats in more contexts; collection helpers such as `sort_field` on `array/matrix` sort; and continued library-surface growth.

Community scripts frequently mix versions, rely on soft conventions (camelCase identifiers, trailing commas), and embed annotation comments (`//@version=6`, `//@description`, `//@function`) that are not ordinary statements yet must survive tooling passes.

2.2 Challenges of a whitespace-significant, expression-oriented DSL

From a language-implementation perspective [Aho et al., 2006, Cooper and Torczon, 2012, Nystrom, 2021], Pine Script™ combines several properties that stress classical parser generators:

1. **Indentation as structure.** Blocks after `if/for/while/switch/type/enum/=>` use Python-like `INDENT/DEDENT` tokens. Indent width is conventionally four spaces; line wrapping may use non-multiple-of-four indents that must *not* introduce structure.
2. **Structures as expressions.** `if`, `for`, `switch`, and related forms appear both as statements and as right-hand sides of assignments (“structure expressions”), so the grammar cannot segregate expression and statement nonterminals as cleanly as C or Java.
3. **Soft keywords and contextual tokens.** Names such as type constructors interact with primary-expression suffixes (attributes, calls, subscripts, template specs like `array.new<float>`).
4. **Literal diversity.** Numbers (decimal/binary/octal/hex, floats, optional underscores), colors (`#RRGGBB`, `#RRGGBBAA`), and strings (escaped single-line and v6 triple-quoted multiline) coexist with comment channels and paste noise.
5. **Versioned semantics outside pure syntax.** Boolean strictness and division typing are semantic; the front-end must still accept the surface forms so later stages can apply version policy.
6. **Scraped corpora.** Real evaluation sets often arrive wrapped in markdown fences, UI chrome (“Expand (*N* lines)”), prose, and HTML—not clean `.pine` files.

These challenges motivate a hybrid design: ANTLR for adaptive `LL(*)` parsing [Parr, 2013, Parr et al., 2014], a hand-written lexer base for indent and newline policy, and an ASDL IR that decouples consumers from parse-tree volatility [Wang et al., 1997, Python Software Foundation, 2024].

3 Grammar Design

3.1 Artifact layout and generation

The front-end distinguishes *hand-edited resources* from *generated, committed* products:

Kind	Path (under <code>src/pynescrypt/ast/grammar/</code>)	Role
Hand	<code>antlr4/resource/PinescriptLexer.g4</code>	Tokens, fragments, channels
Hand	<code>antlr4/resource/PinescriptParser.g4</code>	Syntactic productions
Hand	<code>antlr4/resource/PinescriptLexerBase.py</code>	Indent / newline engine
Hand	<code>antlr4/resource/PinescriptParserBase.py</code>	Parser super-class hooks
Hand	<code>asdl/resource/Pinescript.asdl</code>	Node schema
Generated	<code>antlr4/generated/*</code>	Python lexer/parser/visitor
Generated	<code>asdl/generated/*</code>	Node classes

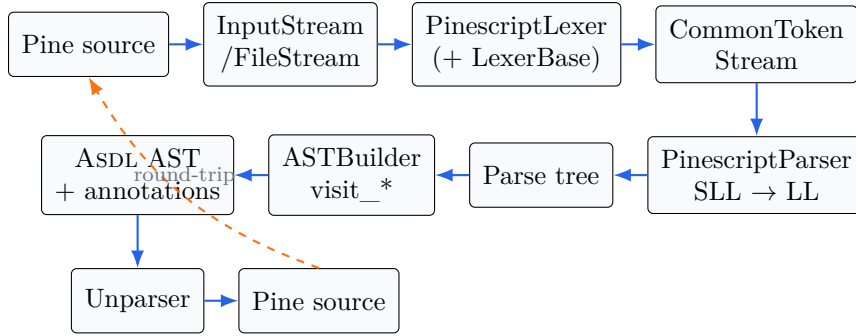


Figure 1: Front-end pipeline: lex $\rightarrow$ parse $\rightarrow$ build $\rightarrow$ annotate $\rightarrow$ unparse. Solid arrows form the primary data path; the dashed arrow denotes structural round-trip used by formatters and tests.

Regeneration uses project tooling (e.g. `hatch run lint:gen-parser`); committing generated Python removes the Java/ANTLR runtime dependency for ordinary installs [Parr, 2013, 2022].

Figure 1 summarizes the end-to-end front-end pipeline.

3.2 Lexical structure

3.2.1 Keywords, operators, and channels

The lexer grammar declares keywords (`and`, `or`, `var`, `varip`, `type`, `method`, `enum`, ...), multi-character operators before single-character ones (so longest-match prefers `<=` over `<`), bitwise operators (`&`, `|`, `^`, `<<`, `>>`, `~`), and assignment forms (`=`, `:=`, `+=`, ...). Comments travel on `COMMENT_CHANNEL`; whitespace is `HIDDEN`. Additional robustness rules accept `#`-style line comments when not part of a color literal, markdown backticks as comment noise, and selected Unicode punctuation as hidden “paste noise.”

3.2.2 Indentation tracking

`PinescriptLexerBase` (Python super-class of the generated lexer) implements the indent engine:

- Stack of indent lengths; emit synthetic `INDENT`/`DEDENT` tokens when structure changes.
- Ignore newlines inside open `(/[`, after operators (line continuation), and for wraps whose indent is not a multiple of four.
- Collapse consecutive newlines; ensure a trailing newline at EOF.
- Special handling of multiline string literals so wrap indentation does not become structural.

This design mirrors classical off-side-rule lexers [Aho et al., 2006] while remaining local to the ANTLR token stream.

3.2.3 Strings, colors, numbers, identifiers

Listing 1 excerpts the v6 triple-quote rules and color fragments. Numbers support base prefixes and digit separators; identifiers use Unicode letter classes (`\p{L}`) so non-ASCII names in community scripts parse cleanly.

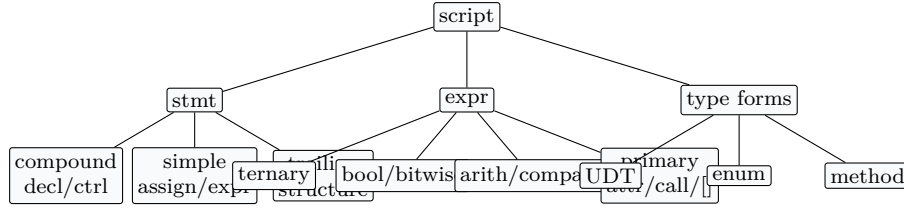


Figure 2: Hierarchical overview of major syntactic families in `PinescriptParser.g4`. Structures double as expressions on assignment RHSs.

Listing 1: Lexer excerpts: triple-quoted strings and color literals (simplified from `PinescriptLexer.g4`).

```

1 // v6 multiline triple-quoted strings
2 TRIPLE_DQ_STRING : '"""' ( ~'"' | '"' ~'"' | '"""' ~'"' )* '"""' -> type(
    STRING ) ;
3 TRIPLE_SQ_STRING : '\'\'' ( ~'\'' | '\'' ~'\'' | '\'\'' ~'\'' )* '\'\''
    -> type(STRING) ;
4
5 // #RRGGBB or #RRGGBBAA (HASH_COMMENT must not steal these)
6 fragment COLOR_LITERAL_RGBA : '#' HEX_DIGIT{8} ;
7 fragment COLOR_LITERAL_RGB : '#' HEX_DIGIT{6} ;

```

3.3 Syntactic productions

3.3.1 Scripts and statement forms

Start rules are `start_script` (full program, mode `exec`) and `start_expression` (mode `eval`). A script is a sequence of statements: compound forms (assignments with structure RHSs, type-/enum/method/ function declarations, control structures) and simple statements (assignments, expressions, imports, `break/continue`), optionally packed on one line with commas. A distinctive production, `trailing_structure_statements`, accepts real-world lines such as

```

Ex = 0.0, Ey = 0.0, for i = 0 to n
...

```

3.3.2 Declarations: functions, methods, types, enums

Function and method declarations use `=>` and a local block (indented or inline). Optional leading type specifications encode return types (`int ilog2(int n) => ...`). Type declarations introduce UDT fields under `INDENT/DEDENT`; enum declarations introduce member definitions with optional values. `export` prefixes support library authors.

3.3.3 Expressions

Expressions are layered by precedence (Figure 2): conditional (ternary), disjunction, conjunction, bitwise OR/XOR/AND, equality, inequality, shifts, additive, multiplicative, unary, and primary (attribute/call/subscript/template). Logical `and` binds less tightly than bitwise OR, matching Pine’s C-like bitwise layering. Primary expressions use labeled alternatives for attribute, call, and subscript forms.

Listing 2 shows the function and enum productions.

Listing 2: Parser productions for functions and enums (from `PinescriptParser.g4`).

```

1 function_declaration
2 : EXPORT? type_specification? name LPAR parameter_list? RPAR RARROW
    local_block ;

```

```

3
4 enum_declaration
5   : EXPORT? ENUM name NEWLINE INDENT enum_definitions DEDENT ;
6
7 enum_definition: name_store (EQUAL expression)? NEWLINE ;

```

3.4 SLL / LL prediction strategy

ANTLR’s adaptive LL(*) engine can fall into expensive full-context prediction on ambiguous decision points [Parr et al., 2014, Parr, 2013]. PYNE uses the standard two-stage strategy in `helper.py`:

1. Parse with `PredictionMode.SLL` and `BailErrorStrategy`.
2. On `ParseCancellationException`: `token_stream.seek(0)`, `parser.reset()`, switch to `PredictionMode.LL` with `DefaultErrorStrategy`, and re-parse.

On stratified mid-size scripts, SLL-first produced identical AST dumps to pure LL with zero fallbacks in the measured sample, while reducing wall time by approximately $5.4\times$ (Section 9). Strategy instances and a shared builder are reused to avoid per-parse allocation.

3.5 Error listeners

`PinescriptErrorListener` is a singleton attached to both lexer and parser after default listeners are removed. Syntax failures surface as `pynscript.ast.error.SyntaxError` with optional `SyntaxErrorDetails` (filename, line, column, source excerpt). Indentation violations raise a dedicated `IndentationError`. The linter maps these structured locations into diagnostic chips for editors (Section 7).

4 ASDL Intermediate Representation

4.1 Schema philosophy

Zephyr ASDL [Wang et al., 1997] describes tree shapes as sum and product types. Python’s compiler has long used ASDL for the `ast` module [Python Software Foundation, 2024]; PYNE follows the same pattern so that:

- Node classes are generated, not hand-synchronized.
- Location attributes (`lineno`, `col_offset`, `end_lineno`, `end_col_offset`) are uniform.
- Visitors and transformers can be written once against a stable API.

The module root distinguishes full scripts from expression-mode trees:

Listing 3: ASDL module root and statement taxonomy (excerpt from `Pinescript.asdl`).

```

1 module Pinescript
2 {
3   mod = Script(stmt* body, string* annotations)
4         | Expression(expr body)
5
6   stmt = FunctionDef(identifier name, param* args, stmt* body,
7                     int? method, int? export, string* annotations)
8         | TypeDef(identifier name, stmt* body, int? export, string*
9                   annotations)

```

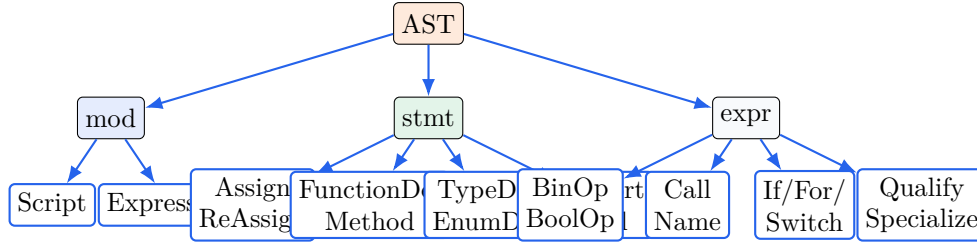


Figure 3: ASDL node hierarchy (conceptual). Control structures are expression nodes when used as values; declarations live under `stmt`. Auxiliary products include `param`, `arg`, `case`, and `Comment`.

```

9      | EnumDef(identifier name, stmt* body, int? export, string*
10      | annotations)
11      | Assign(expr target, expr? value, expr? type, decl_mode? mode,
12      | int? export, string* annotations)
13      | ReAssign(expr target, expr value)
14      | AugAssign(expr target, operator op, expr value)
15      | Import(identifier namespace, identifier name, int version,
16      | identifier? alias)
17      | Expr(expr value) | Break | Continue
18      attributes (int lineno, int col_offset,
19      | int? end_lineno, int? end_col_offset)
20 }

```

4.2 Node taxonomy

Figure 3 sketches the conceptual hierarchy.

Expressions. Algebraic nodes (`BoolOp`, `BinOp`, `UnaryOp`, `Compare`, `Conditional`) coexist with `Call`, `Attribute`, `Subscript`, `Name`, `Tuple`, and `Constant`. Control forms `If`, `ForTo`, `ForIn`, `While`, and `Switch` are *expressions* in the schema, reflecting Pine’s expression-oriented structures. Type constructors use `Qualify` (`const/input/simple/series`) and `Specialize` (generic arguments).

Enums of operators and modes. `decl_mode` $\in \{\text{Var}, \text{VarIp}\}$; `type_qual` $\in \{\text{Const}, \text{Input}, \text{Simple}, \text{Series}\}$; `expr_context` $\in \{\text{Load}, \text{Store}\}$; plus operator, boolean, unary, and comparison enumerations including bitwise forms.

4.3 Visitor and transformer patterns

`NodeVisitor` dispatches on concrete types with a type-object method cache (avoiding class-name string lookups). `NodeTransformer` rewrites trees bottom-up: returning `None` removes a node; returning an `AST` replaces it; returning a list splices into list fields [Gammma et al., 1994, Python Software Foundation, 2024]. Helpers `walk`, `iter_fields`, and `iter_child_nodes` support ad hoc analysis without subclassing. The unparser and several evaluator passes are visitors; desugaring and future IR lowers can be transformers.

5 AST Construction

5.1 Builder mapping: parse tree $\rightarrow$ ASDL

`PinescriptASTBuilder` extends the ANTLR-generated `PinescriptParserVisitor`. Each `visit_*` method maps a parser context to one or more ASDL nodes. Design choices include:

- **Singleton operator/context nodes** (`Load`, `Store`, `Add`, ...) to reduce allocation on hot paths.
- **Location attachment** from start/stop tokens, with a single-line fast path when the stop token contains no newline.
- **Store contexts** on assignment targets (names, attributes, subscripts, tuples).
- **Number and string decoding** for `Constant` values, including multiline string normalization.
- **Shared builder instance** across parses (stateless visits).

Deeply nested ternaries raise the recursion limit temporarily to 5000 during walk and build.

5.2 Annotation attachment

Annotation comments (`//@...`) are not productions in the statement grammar. After a successful `exec`-mode build, the helper:

1. Skips the entire annotation pass if the source contains no `@` (cheap filter for micro-snippets and many fixtures).
2. Collects statement nodes via `StatementCollector`.
3. Scans the token stream for `COMMENT` tokens containing `@`, classifying kind/value via `PinescriptCommentParser`.
4. Orders comments and statements by $(lineno, col)$, groups consecutive comments, and attaches them to the following node.

Script-level annotations (kind suffix `S`) populate `Script.annotations`; function (`F`), type (`T`), and variable (`V`) annotations attach to the corresponding statement nodes. Algorithm 1 abstracts the attachment procedure.

Round-trip emits these annotation strings as full lines before the relevant construct, preserving version directives and documentation markers used by the LSP hover surface.

6 Unparsing and Formatting

6.1 Pretty-printing model

`NodeUnparser` walks the ASDL tree and appends source fragments to a list buffer. Operator precedence is tracked with a `Precedence` lattice so parentheses appear only when required for associativity or binding strength. Binary operators assign the next-lower precedence on the right-hand child to keep left-associative chains free of redundant parentheses.

Algorithm 1 AnnotationAttachment

Require: Script AST T , source text S , token stream τ

Ensure: Annotations attached on T and eligible statements

```
1: if "@"  $\notin S$  then
2:   return  $T$  ▷ no annotation comments possible
3: end if
4:  $stmts \leftarrow \text{StatementCollector}(T)$ 
5:  $cmts \leftarrow \{c \in \tau \mid c.type = \text{COMMENT} \wedge "@" \in c.text\}$ 
6:  $items \leftarrow \text{sort}(cmts \cup stmts \text{ by } (line, col))$ 
7: Group consecutive comments vs. statements
8: Attach script-level @*S comments to  $T.annotations$ 
9: for each (comment group  $G$ , statement  $s$ ) pair do
10:   Attach  $G$  filtered by kind suffix to  $s.annotations$  if applicable
11: end for
12: return  $T$ 
```

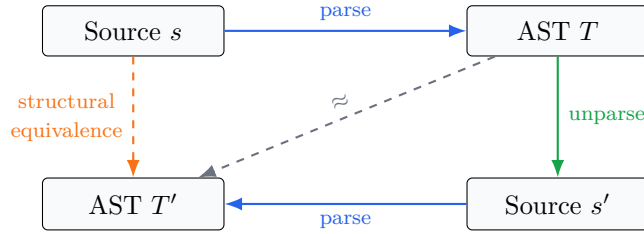


Figure 4: Round-trip commutative diagram. Formatters implement $s \mapsto s' = \text{unparse}(\text{parse}(s))$; tests require $\text{parse}(s')$ to match $\text{parse}(s)$ up to agreed normal forms (and often check re-parse stability of s').

Normalization policy. The round-trip goal is *semantic and structural* fidelity, not byte-identical reproduction:

- Indentation is four spaces; operators are spaced consistently.
- Free-floating non-annotation comments may move or drop.
- Multiline string constants prefer triple-quoted forms when they contain newlines.
- Annotation lists on script/function/type/assign nodes are re-emitted.

Figure 4 depicts the intended commutative diagram for formatters and tests.

6.2 Performance engineering

Unparse micro-benchmarks on a 99-script corpus showed dispatch and buffer construction dominating. Optimizations include:

- **Thread-local reuse** of a single `NodeUnparser` (resets buffers between calls; keeps type→method cache warm).
- **Type-keyed dispatch** instead of class-name strings.
- **Precomputed indent prefixes** and operator token strings.
- **Lightweight context managers** for delimiters/blocks (avoiding `contextlib` generator overhead on every parenthesized subexpression).

Table 1: Built-in linter rules (structural and style).

Code	Severity	Description
E001	error	Syntax error from <code>parse</code> (with line/column when known)
W001	warning	Missing <code>//@version</code> declaration
W002	warning	Version < 5 deprecated; suggest v5/v6
W101	warning	Legacy <code>security(</code> pattern; prefer <code>request.security</code>
W102	warning	Histogram plot style suggestion
W103	warning	<code>var int x = na</code> initialization style
C001	style	<code>ta.*</code> assignment naming heuristic (camelCase)
C002	style	Line length > 120
C003	style	Single-line indented <code>if</code> style caution
C004	style	File should end with newline

These changes yielded approximately $1.95\times$ higher unparse throughput on the corpus loop (Section 9) with byte-identical output relative to the pre-optimization unparser.

6.3 LSP formatting

The language server exposes document formatting as `parse` $\rightarrow$ `unparse`, reusing the same path as the CLI `format` and `parse-and-unparse` commands [Microsoft, 2023, HOOX Project, 2026]. Because annotations survive the builder post-pass, version headers remain at the top of formatted buffers.

7 Static Analysis

7.1 Linter rules

`PineLinter` combines a full parse with lightweight regex heuristics. Rule codes use prefixes E (errors), W (warnings), and C (style). Table 1 lists the built-in rules (nine-plus checks spanning syntax, version, deprecation, naming, and style).

The linter is intentionally not a full type checker: it provides fast editor feedback and CI gates, while deeper type modeling lives in `type_system.py`.

7.2 Type system overview

The type model represents:

Qualifiers

`TypeQualifier`: `const`, `simple`, `series`, `input`.

Builtin scalars

`int`, `float`, `bool`, `string`, `color`, `na`, and `v6 enum`.

Collections

`ArrayType`, `MatrixType`, and map-like structures with element (and key/value) parameters.

UDTs

`UserDefinedType` with fields and method resolution via `MethodResolver`.

Registry

`TypeRegistry` for name lookup during evaluation and tooling.

Algorithm 2 SanitizeCorpus

Require: Raw text R (possibly mixed markdown / UI chrome)

Ensure: Sanitized source S or minimal stub

- 1: $L \leftarrow \text{splitlines}(R)$; track open quote state across lines
 - 2: Drop fence lines, expand stubs, HR, URL-only, footer labels, HTML noise
 - 3: Drop leading prose until a Pine-like line (`//@version, indicator(, strategy(, assignments, ...)`)
 - 4: Repair mild truncation artifacts when safe
 - 5: **if** L has no usable Pine **then**
 - 6: **return** minimal stub `//@version=5 / indicator / plot`
 - 7: **end if**
 - 8: **return** $\text{join}(L)$
-

Inference is partial and runtime-assisted: the interpreter uses the model for compatibility checks and UDT instantiation, while the LSP consumes related metadata inventories (hundreds of builtins—on the order of 482–800+ surface entries depending on inventory generation) for completion and signatures [HOOX Project, 2026]. Full bidirectional checking of every Pine qualifier lattice edge remains future work (Section 11).

8 Corpus Engineering

8.1 Sanitization of scraped sources

Community scripts scraped from publications, forums, or documentation pages rarely arrive as pure Pine. `corpus_sanitize.py` implements `sanitize_corpus_source`, which strips page chrome while preserving in-comment markdown used for `//@function` documentation.

Typical removals include: markdown fences (`"'pine)`; blockquote chrome; “Expand (N lines)” UI stubs; horizontal rules; bare URLs; HTML comments; publication footers (author/license/tags); leading prose before the first script-like line; and mis-collected shell/Python/HTML fragments. If no usable Pine remains, a minimal parseable stub is returned so callers fail softly.

The compiler and multi-run hosts invoke sanitization on cache miss paths so warm hits skip the cost [HOOX Project, 2026].

8.2 Parse-fail buckets and regeneration workflow

Corpus tests parametrize over directories of `.pine` files. Failures are bucketed by root cause (lexer indent, missing construct, sanitize residue, version-specific syntax). The engineering loop is:

1. Reproduce with `parse/unparse` on the failing file.
2. Prefer grammar or lexer-base fixes in `resource/*.g4 / LexerBase.py`; never hand-edit `generated/`.
3. Regenerate parser/lexer; align builder `visit_*` names with new rule contexts.
4. Extend ASDL only when node shape must change; regenerate node classes.
5. Add a focused unit test, then re-run corpus and linter suites.

This discipline keeps the approximate grammar evolving with community patterns without forking generated code.

Algorithm 3 RoundTripCheck

Require: Source s , optional sanitize flag

- 1: $s_0 \leftarrow \text{sanitize}(s)$ if enabled else s
 - 2: $T \leftarrow \text{parse}(s_0)$
 - 3: $s' \leftarrow \text{unparse}(T)$
 - 4: $T' \leftarrow \text{parse}(s')$
 - 5: Assert $\text{dump}(T)$ compatible with $\text{dump}(T')$ (structure; attributes per test policy)
 - 6: Optionally assert selected annotation strings preserved
-

Table 2: Parse performance (representative agent results).

Workload	Before / LL-only	After / SLL-first	Speedup
Mean per script (12-file mix)	$\sim 80\text{--}140$ ms	$\sim 14.5\text{--}15$ ms	$\sim 5.4\times$
Ops/s (mix)	$\sim 7\text{--}13$	~ 69	$\sim 5\text{--}10\times$
Complex ~ 16 KB script	~ 1917 ms	~ 254 ms	$\sim 7.5\times$

9 Evaluation

9.1 Correctness: coverage and round-trip

- **Parser coverage.** High success rates on 138+ real scripts in parse-and-unparse suites (e.g., 138 passed on a builtin-script directory configuration reported in performance agents). Broader project automation exceeds 1100 tests across front-end, runtime, and compiler concerns [HOOX Project, 2026].
- **SLL $\equiv$ LL.** On 30 size-stratified scripts, AST dumps with attributes matched between SLL-first and forced pure-LL parses (0 mismatches in that sample).
- **Annotations.** Scripts with `//@` retain `script.annotations` and declaration annotations through parse; unparse re-emits them as lines.
- **Modes.** `mode="eval"` parses isolated expressions; `mode="exec"` parses full scripts.

Algorithm 3 states the structural check used in CI-oriented workflows.

9.2 Parse performance

Profiling showed ANTLR adaptive prediction dominating wall time, with the builder a smaller fraction. Table 2 summarizes reported measurements from the project’s parse performance agents (Python 3.14, in-process microbenchmarks).

Fair same-process A/B (12 iterations $\times$ 12 scripts) reported approximately 175.7 ms vs. 953.7 ms per pass ($5.43\times$, 81.6% improvement) for SLL-first vs. LL-only, with identical trees on success. Additional micro-optimizations (annotation skip without `@`, recursion-limit guard, location fast path, lexer deque pending tokens) further reduced constant factors after the SLL win.

Figure 5 visualizes the headline SLL vs. LL comparison.

9.3 Unparse performance

On 99 scripts from an indicators/strategies set, thread-local unparser reuse and dispatch optimizations improved median throughput from ~ 1266 to ~ 2469 unparses/s ($\approx 1.95\times$), with large-script calls improving by $\approx 2.12\times$. Outputs remained byte-identical to the baseline unparser across successfully parsed ASTs.

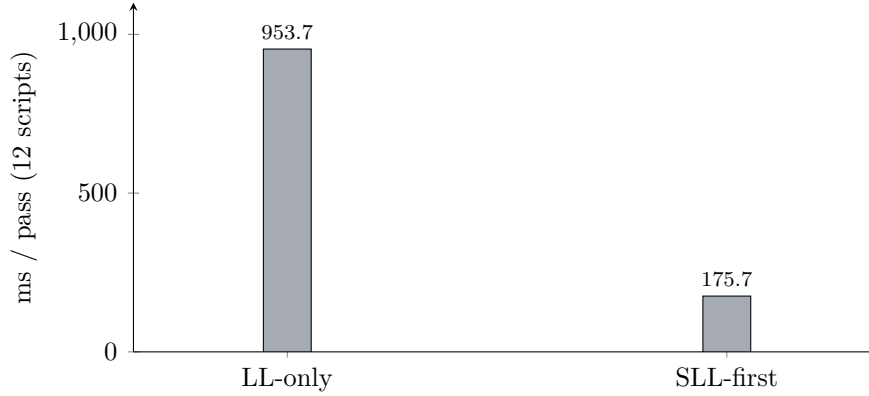


Figure 5: Parse performance: pure LL vs. SLL-first with LL fallback on a 12-script stratified mix (fair in-process A/B). Trees matched on the measured sample.

9.4 Parse cache

A process-local LRU keyed by $(\text{SHA256}(\text{source}), \text{mode})$ (default max 128, `threading.RLock`, env toggles `PYNE_PARSE_CACHE` / `PYNE_PARSE_CACHE_MAX`) returns shared AST identity on hit. On hits, evaluator-bound `_pine_call_site` attributes are scrubbed so multi-run hosts do not reuse another evaluator’s bound methods. Failed parses are not cached. Multi-parse warm paths report order-of-magnitude gains approaching $1000\times$ vs. cold ANTLR parse when the cache hits repeatedly on identical sources (host benchmarks; dominated by hash + dict lookup).

10 Related Work

Parser generators and ANTLR. ANTLR 4 popularized adaptive $\text{LL}(\ast)$ parsing with automatic left-recursion elimination and continuous lookahead [Parr, 2013, Parr et al., 2014, Parr and Quong, 1995]. The SLL-first/LL-fallback pattern is recommended practice for throughput without sacrificing generality; PYNE confirms large wins on a charting DSL with indent-sensitive lexing layered outside pure grammar rules.

ASDL and language IRs. Zephyr ASDL [Wang et al., 1997] and Python’s `ast` module [Python Software Foundation, 2024] demonstrate the value of declarative node schemas for compilers and tools. PYNE reuses this philosophy for a DSL, including location attributes and visitor/transformer APIs familiar to Python tooling authors.

DSL engineering. Fowler [Fowler, 2010] and Hudak [Hudak, 1996] survey external vs. embedded DSLs. Pine Script™ is an external, product-hosted language; PYNE builds an external toolchain around an approximate grammar rather than embedding Pine in Python syntax. Classic compiler texts [Aho et al., 2006, Cooper and Torczon, 2012] and Nystrom’s interpreter construction [Nystrom, 2021] inform the pipeline staging (lex, parse, AST, analysis, unparse).

Language servers and formatters. LSP [Microsoft, 2023] standardizes editor integration; formatters based on parse-print pipelines (gofmt style) motivate lossless-enough unparsing. PYNE’s format path is intentionally the same as the test round-trip.

Financial DSLs. Vendor documentation [TradingView, Inc., 2024] specifies user-facing semantics but does not publish an official grammar. Independent implementations must reverse-engineer surface syntax from documents and corpora—a central motivation for this paper’s grammar-engineering narrative.

11 Limitations

1. **Approximate, unofficial grammar.** The `.g4` sources are reverse-engineered and corpus-hardened. They are not an official TradingView grammar and may accept or reject edge cases differently from the proprietary compiler.
2. **Semantic version gaps.** Some v6 semantic rules (strict booleans, division typing) are enforced in later stages, not fully in the parser.
3. **Comment fidelity.** Only `//@`-style annotations are first-class on the AST; arbitrary comments are not a full comment-preserving CST.
4. **Linter depth.** Rules are heuristic; the type model is not a complete static type checker for the Pine qualifier lattice.
5. **Cache mutability.** Shared AST identity requires read-only treatment or explicit `clear_parse_cache` after transformations.
6. **Corpus bias.** Evaluation scripts under-represent some rare library and brokerage-specific patterns; sanitize stubs can mask total scrape failure.

12 Conclusion

We described the PYNE language front-end for Pine Script™: an ANTLR4 grammar with Python indent tracking, a Zephyr-ASDL IR, a visitor builder with annotation attachment, a precedence-aware unparser, a structural linter, and a qualifier-aware type model, complemented by corpus sanitization and regeneration discipline. The pipeline enables formatters, LSP features, interpreters, and compilers to share one tree shape. Measured optimizations—SLL-first parsing ($\approx 5.4\times$), unparser reuse ($\approx 1.95\times$), and SHA-256 AST caching (orders of magnitude on multi-parse hits)—show that careful front-end engineering pays off even when ANTLR already dominates profiles.

Future work includes tighter CST/comment preservation, richer static checking of qualifiers and UDTs, grammar differential testing against larger public corpora, and continued v6+ surface tracking as the language evolves. PYNE remains an independent open implementation; we hope the documented grammar-engineering practices help other charting-DSL and financial-DSL tooling efforts [HOOX Project, 2026].

Acknowledgments. We thank contributors to the PYNE / HOOX open-source effort and authors of ANTLR, ASDL, and the Python `ast` ecosystem.

Trademark notice. Pine Script™ and TradingView® are trademarks of TradingView, Inc. PYNE is an independent, unofficial implementation and is not affiliated with, authorized by, sponsored by, or endorsed by TradingView, Inc.

References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, Boston, 2nd edition, 2006.
- Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2012.
- Martin Fowler. *Domain-specific languages*. Addison-Wesley, 2010.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1994.
- HOOX Project. PYNE: Independent open toolchain for the Pine Script language. <https://github.com/hoox-sh/pyne>, 2026. Version 0.3.0; PyPI package `hoox-pyne`.
- Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es): 196, 1996.
- Microsoft. Language server protocol specification. <https://microsoft.github.io/language-server-protocol/>, 2023. Version 3.17.
- Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.
- Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2013.
- Terence Parr. ANTLR 4 documentation. <https://www.antlr.org/>, 2022.
- Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *OOPSLA*. ACM, 2014.
- Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
- Python Software Foundation. Python ASDL and the `ast` module. <https://docs.python.org/3/library/ast.html>, 2024.
- TradingView, Inc. Pine Script language reference manual. <https://www.tradingview.com/pine-script-docs/>, 2024. Accessed 2026.
- Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–228. USENIX, 1997.