LANGUAGE IMPLEMENTERS · CONTRIBUTORS

LANGUAGE CORE MANUAL

ANTLR4 grammar, ASDL AST, builder, unparser, type system, linter

Open-source Pine Script evaluation framework

CONTENTS

OVERVIEW

title: "Language Core" description: "ANTLR4 grammar, ASDL AST, builder, visitors, unparser, type system, linter, and error model for PYNE."

LANGUAGE CORE

The language core is the front half of the evaluate contract: source text becomes a typed tree; the tree can be walked, rewritten, dumped, and unparsed without ever entering the bar loop.

Abstract

PYNE treats Pine Script as a **formal pipeline**, not a host plugin. Lexing and parsing are ANTLR4-driven. The concrete syntax tree is lowered by a hand-written visitor (`PinescriptASTBuilder`) into ASDL-generated dataclasses. Public helpers in `helper.py` mirror Python's `ast` module: `parse` , `unparse` , `dump` , `walk` ,

`literal_eval`. Static tools (linter, type registry, LSP) consume the same tree. Evaluation is out of scope here — see Runtime.

Hard rule: edit grammar only under `resource/`. Paths under `generated/` are regenerated artifacts and must not be hand-edited.

Conceptual model

```
flowchart TB
  SRC["Source string / file"]
  LEX["PinescriptLexer

+ LexerBase INDENT/DEDENT"]
  TOK["CommonTokenStream"]
  PAR["PinescriptParser"]
  PT["Parse tree contexts"]
  BLD["PinescriptASTBuilder"]
  ANN["Comment → annotations"]
  AST["ASDL AST

Script | Expression"]
  VIS["NodeVisitor / Transformer"]
  UNP["NodeUnparser"]
  LIN["PineLinter"]
  TYP["TypeRegistry"]
  EVAL["Evaluator / LSP / API"]

  SRC --> LEX --> TOK --> PAR --> PT --> BLD --> AST
  TOK -.->|COMMENT tokens| ANN
  BLD --> ANN
  ANN --> AST
  AST --> VIS
  AST --> UNP
  AST --> LIN
  AST --> TYP
  AST --> EVAL
```

Stage	Role	Primary path
Grammar	Concrete syntax	`src/pynescript/ast/grammar/antlr4/resource/*.g4`
ASDL	Algebraic node shape	`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl`
Builder	CST → AST	`src/pynescript/ast/builder.py`
Helper	Public parse/unparse API	`src/pynescript/ast/helper.py`
Nodes	Re-export generated types	`src/pynescript/ast/node.py`

Interface surface

Consumers almost always enter through `pynescript.ast.helper`:

```
from pynescript.ast.helper import parse, unparse, dump, walk, literal_eval

tree = parse('//@version=5\nindicator("x")\nplot(close)')
print(dump(tree, indent=2))
print(unparse(tree))
```

Modes:

`mode`	Entry rule	Root node
`"exec"` (default)	`start_script`	`Script`
`"eval"`	`start_expression`	`Expression`

Full page inventory:

Page	Topic
Grammar (ANTLR4)	Resource vs generated; regen; v6 triple strings
ASDL schema	Algebraic AST; codegen
Builder	CST visitor → ASDL nodes
Helper API	`parse`, `dump`, `walk`, locations
Unparser	Precedence-aware source regen
Visitor / transformer	Read vs rewrite
Type system	Qualifiers, UDT registry
Linter	Static rules, codes
Error model	`SyntaxError`, indent, listener

Internals

```
src/pynescript/ast/
  helper.py              # public pipeline orchestration
  builder.py             # PinescriptASTBuilder
  unparser.py            # NodeUnparser + Precedence
  visitor.py             # NodeVisitor
  transformer.py         # NodeTransformer
  collector.py           # StatementCollector (annotation pairing)
  linter.py              # PineLinter
  type_system.py         # Type*, TypeRegistry, MethodResolver
  error.py               # SyntaxError / IndentationError
  node.py                # from grammar.asdl.generated import *
  grammar/
    antlr4/
      resource/          # EDIT HERE: .g4 + *Base.py
      generated/         # NEVER HAND-EDIT
      error_listener.py
      tool/generate.py
    asdl/
      resource/          # EDIT HERE: Pinescript.asdl
      generated/         # NEVER HAND-EDIT
      tool/generate.py
```

Thin wrappers `grammar/antlr4/lexer.py`, `parser.py`, `visitor.py` re-export generated classes so application code imports stable paths.

Invariants

1. **Resource is source of truth.** Lexer/parser `.g4` and `Pinescript.asdl` define syntax and node shape; generated Python is a build product (committed so install does not require Java).
2. **Round-trip is a test oracle.** For the supported surface, `unparse(parse(src))` should re-parse and preserve semantics (not byte-identical formatting).
3. **Locations are first-class.** Statements and expressions carry `lineno`, `col_offset`, optional `end_*` for diagnostics and LSP.
4. **Annotations are not free-floating.** Comment tokens on `COMMENT_CHANNEL` are reified as `Comment` nodes and attached to `Script` / `FunctionDef` / `TypeDef` / `Assign` via kind suffixes (`S`, `F`, `T`, `V`).
5. **Recursion budget.** Deep ternary chains raise the Python recursion limit temporarily during parse (cap ≥ 5000).

Worked examples

Minimal script:

```
from pynescript.ast.helper import parse, dump

src = """
//@version=5
indicator("demo")
x = ta.sma(close, 14)
plot(x)
"""
tree = parse(src)
assert tree.__class__.__name__ == "Script"
assert any("version" in a for a in (tree.annotations or []))
print(dump(tree, include_attributes=True, indent=2))
```

Expression-only eval mode:

```
from pynescript.ast.helper import parse, literal_eval

expr = parse("1 + 2 * 3", mode="eval")
assert literal_eval(expr) == 7
```

Failure modes

Symptom	Likely cause	Where to look
`SyntaxError` with caret	Lexer/parser reject	Error model
`IndentationError`	Bad INDENT/DEDENT from LexerBase	`PinescriptLexerBase.py`
Missing `visit_*` after grammar change	Builder not updated for new rule	Builder
Hand-edit of `generated/` lost on regen	Violated resource-only rule	Grammar
Full regen breaks `builder.py`	Parser context accessors changed	Prefer targeted lexer refresh (v6 case study)

See also

- Grammar (ANTLR4)
- Helper API
- Runtime
- LSP
- Compatibility

GRAMMAR ANTLR4

title: "Grammar (ANTLR4)" description: "Resource .g4 grammars, generated artifacts, INDENT/DEDENT, regeneration, and the never-edit-generated rule."

GRAMMAR (ANTLR4)

Concrete syntax for Pine Script in PYNE is defined by a split lexer/parser ANTLR4 grammar plus hand-written base classes that inject Python-like indentation tokens.

Abstract

Two grammars live under `src/pynescript/ast/grammar/antlr4/resource/`:

- `PinescriptLexer.g4` — keywords, operators, literals (including v6 triple-quoted strings), comments, hidden whitespace.

- `PinescriptParser.g4` — statements, expressions, type/enum/function/method declarations, control structures.

Python runtime classes under `generated/` are **outputs** of `antlr4 -Dlanguage=Python3`. They are committed so end users never need a JDK. **Never edit files under `generated/`** — changes evaporate on the next regeneration and fight the build pipeline.

Hand-written bases (copied into `generated/` by the generate tool):

- `resource/PinescriptLexerBase.py` — INDENT/DEDENT, newline elision, multiline string handling.
- `resource/PinescriptParserBase.py` — parser superClass hooks.

Conceptual model

```
flowchart LR
  subgraph resource ["resource/ — edit here"]
    LG4["PinescriptLexer.g4"]
    PG4["PinescriptParser.g4"]
    LB["PinescriptLexerBase.py"]
    PB["PinescriptParserBase.py"]
  end

  GEN["antlr4 tool"]
  subgraph generated ["generated/ — do not hand-edit"]
    LPY["PinescriptLexer.py"]
    PPY["PinescriptParser.py"]
    VIS["PinescriptParserVisitor.py"]
    LIS["PinescriptParserListener.py"]
    LBgen["*Base.py copies"]
  end

  WRAP["lexer.py / parser.py / visitor.py wrappers"]
  BLD["builder.py visit_*"]

  LG4 --> GEN
  PG4 --> GEN
  GEN --> LPY
  GEN --> PPY
  GEN --> VIS
  GEN --> LIS
  LB --> LBgen
  PB --> LBgen
  LPY --> WRAP
  PPY --> WRAP
  VIS --> BLD
```

Interface surface

Application code should not import `generated` symbols directly when a wrapper exists:

```
from pynescript.ast.grammar.antlr4.lexer import PinescriptLexer
from pynescript.ast.grammar.antlr4.parser import PinescriptParser
from pynescript.ast.grammar.antlr4.visitor import PinescriptParserVisitor
from pynescript.ast.grammar.antlr4.error_listener import PinescriptErrorListener
```

The public parse path is higher level — Helper API — which constructs lexer, token stream, parser, and strips default console error listeners in favor of `PinescriptErrorListener`.

Entry rules (`PinescriptParser.g4`)

Rule	Purpose
`start` → `start_script`	Full script (`statements? EOF`)
`start_expression`	Single expression + optional NEWLINE + EOF
`start_comments`	Comment stream only

Lexer structural facts

Token / channel	Behavior
`INDENT` / `DEDENT`	Imaginary tokens emitted by `PinescriptLexerBase` (declared in `tokens { }`)
`COMMENT`	`//` line comments → `COMMENT_CHANNEL` (not the default channel)
`WS`	Spaces/tabs/form-feed → `HIDDEN`
`ERROR_TOKEN`	Catch-all `.` for recovery/reporting
`TRIPLE_DQ_STRING` / `TRIPLE_SQ_STRING`	v6 multiline strings, typed as `STRING`

Keywords include `var`, `varip`, `method`, `export`, `enum`, `type`, qualifiers `const` / `input` / `simple` / `series`, and control forms `if` / `for` / `while` / `switch`.

Parser structural facts

- **Compound vs simple statements.** Functions, methods, types, enums, and structure-valued assignments are compound; imports, break/continue, expression statements, and simple assignments terminate with `NEWLINE`.
- **Structures are dual-use.** `if` / `for` / `while` / `switch` appear both as statements and as **expressions** (`structure_expression`) — the classic Pine "if returns a value" form.
- **Local blocks.** Either indented (`NEWLINE INDENT statements DEDENT`) or inline single statement.
- **Library export.** `EXPORT?` prefixes function/method/type/enum and June-2025 `export const T name = …` compound initializers.

Internals

Paths

Path	Edit?	Role
`src/pynescript/ast/grammar/antlr4/resource/PinescriptLexer.g4`	**Yes**	Lexer grammar
`src/pynescript/ast/grammar/antlr4/resource/PinescriptParser.g4`	**Yes**	Parser grammar
`src/pynescript/ast/grammar/antlr4/resource/PinescriptLexerBase.py`	**Yes**	Indentation machine
`src/pynescript/ast/grammar/antlr4/resource/PinescriptParserBase.py`	**Yes**	Parser base
`src/pynescript/ast/grammar/antlr4/generated/**`	**No**	ANTLR output + copied bases
`src/pynescript/ast/grammar/antlr4/tool/generate.py`	Tool	Runs `antlr4`, copies `*.py` bases
`src/pynescript/ast/grammar/antlr4/error_listener.py`	Hand	Maps ANTLR errors → `ast.error.SyntaxError`

LexerBase responsibilities

`PinescriptLexerBase` is not optional sugar; it is part of the language definition:

- Ignore leading / collapse consecutive newlines; ensure trailing newline.
- Suppress newlines inside open `()` / `[]`, after operators, and for soft wrap lines whose indent width is not a multiple of four.
- Push `INDENT` / `DEDENT` with tab length and indent length = 4.
- Special-case multiline string tokens so wrap indentation inside strings is not mis-tokenized as structure.

Indent mistakes surface as `pynescript.ast.error.IndentationError` (subclass of the project `SyntaxError`).

Regeneration

```
# Preferred hatch env (pulls antlr4-cli)
hatch run lint:gen-parser

# Or: python -m pynescript.ast.grammar.antlr4.tool.generate
# which invokes: $(python)/antlr4 -o generated -lib resource -listener -visitor -Dlanguage=Python3
# then copies resource/*.py into generated/
```

Requires a working `antlr4` CLI (often `antlr4-cli` on `PATH`, sometimes under `~/.local/bin` or next to `sys.executable`). Generated modules are excluded from mypy/ruff strictness in `pyproject.toml` — do not “lint-fix” them by hand.

Downstream after grammar change

1. Regenerate (or selectively refresh lexer — see case study).
2. Add/adjust `visit_*` methods in `src/pynescript/ast/builder.py` for new or renamed rules.
3. If the **shape** of the language changed (new statement/expression kind), update `Pinescript.asdl` and regenerate ASDL nodes — ASDL schema.
4. Smoke-test with `parse` / `unparse` on a minimal snippet **before** the full `tests/data/builtin_scripts/` corpus.

Invariants

1. **Never hand-edit `generated/`.** Patches belong in `resource/*.g4` or `resource/*Base.py`.
2. **Visitor contract.** Generated `PinescriptParserVisitor` method names follow ANTLR rule names; the builder subclass must track renames.
3. **Comments are off the default channel.** Annotation logic in `helper._collect_comment_nodes` iterates `token_stream.tokens` for `COMMENT` type after `fill()`.
4. **Indent unit is four spaces.** Soft-wrap lines that are not multiples of four are newline-elided, not nested blocks.

Worked examples

Minimal grammar-level mental model

```
//@version=5
indicator("x")
if close > open
    plot(1)
else
    plot(0)
```

Lexer emits NEWLINE, INDENT, DEDENT around the `if` bodies. Parser builds `if_structure` → builder yields `ast.If` with `body` / `orelse` statement lists.

v6 triple-quoted strings

Resource lexer rules (illustrative — see live `PinescriptLexer.g4`):

```
TRIPLE_DQ_STRING : '"""' ( ~'"' | '"' ~'"' | '""' ~'"' )* '"""' -> type(STRING) ;
TRIPLE_SQ_STRING : '\'\'\'' ( ~'\'' | '\'' ~'\'' | '\'\'' ~'\'' )* '\'\'\'' -> type(STRING) ;
```

ANTLR quoting pitfall: putting `"'''"` or `'"""'` as fragment literals can produce tool errors (`quote came as a complete surprise`). Factor starters when needed:

```
fragment TRIPLE_SQ_START: '\'' '\'' '\'';
```

Content and source indentation inside the triple quotes are preserved literally (no automatic dedent).

Case study: targeted lexer refresh (2026-07)

Full regeneration once produced a `PinescriptParser.py` whose context accessors diverged from what the hand-written builder expected (e.g. missing `template_spec_suffix()`), breaking even trivial parses. The practical recovery:

1. Edit **only** `resource/` grammar.
2. Generate the lexer in a clean temp directory (avoids nested `src/` mirror paths).
3. Copy **only** `PinescriptLexer.py` (+ refreshed `LexerBase.py`) into `generated/`.
4. Leave committed `PinescriptParser.py` and visitors untouched unless prepared to patch the builder in the same change.
5. Verify immediately:

```
from pynescript.ast.helper import parse, unparse
ast = parse('indicator("x")\ns = """\nfoo\n  bar\n"""\n')
assert "foo" in unparse(ast)
```

Failure modes

Failure	Cause	Mitigation
`mismatched input` after `"""`	Old lexer ATN without triple rules	Refresh generated lexer from resource
Full regen breaks builder	Parser context API drift	Selective copy; or update all `visit_*` together
`antlr4` not found	CLI not on PATH	Use hatch env or full path to `antlr4-cli`
Nested `generated/src/pynescript/...` junk	Running antlr with wrong `-o` cwd	Generate from clean temp + explicit `-o`
IndentationError mid-file	Mixed tabs/spaces or wrong nest	Align to 4-space blocks; check soft-wrap rules
Silent loss of hand patches	Edited `generated/`	Revert; re-apply in `resource/`

See also

- ASDL schema
- Builder
- Error model
- Helper API
- Missing features

ASDL SCHEMA

title: "ASDL Schema" description: "Algebraic AST definition in Pinescript.asdl, generated dataclasses, and how schema changes flow into the pipeline."

ASDL SCHEMA

The abstract syntax tree is not free-form dicts. It is an **algebraic data type** declared in ASDL and materialized as Python dataclasses.

Abstract

`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl` defines the `Pinescript` module: roots (`Script`, `Expression`), statements, expressions (including control structures that are also expressions), operators, params, args, cases, and comments. Codegen writes `grammar/asdl/generated/PinescriptASTNode.py`.

`src/pynescript/ast/node.py` re-exports that module so the rest of the codebase imports `from pynescript.ast import node as ast` (or `from pynescript.ast.node import Script`).

ASDL is the contract between **builder** (produces nodes), **evaluator/LSP** (consume nodes), and **unparser** (serializes nodes). Changing a field without regenerating and updating all three is a schema break.

Conceptual model

```
flowchart TB
  ASDL["Pinescript.asdl

resource"]
  GEN["asdl / pyasdl generator"]
  PY["PinescriptASTNode.py

generated dataclasses"]
  NODE["node.py re-export"]
  BLD["builder.py"]
  UNP["unparser.py"]
  VIS["visitor / transformer"]
  EV["evaluator / LSP"]

  ASDL --> GEN --> PY --> NODE
  NODE --> BLD
  NODE --> UNP
  NODE --> VIS
  NODE --> EV
```

Each product sum type becomes a Python class hierarchy under `AST` with:

- `_fields` — logical children (walked by `iter_fields` / visitors)
- `_attributes` — location metadata (`lineno`, `col_offset`, `end_lineno`, `end_col_offset`) where declared

Interface surface

Module roots (`mod`)

Constructor	Fields	Use
`Script`	`stmt* body`, `string* annotations`	Full file (`mode="exec"`)
`Expression`	`expr body`	Single expression (`mode="eval"`)

Statements (`stmt`)

Constructor	Notable fields
`FunctionDef`	`name` , `param* args` , `body` , `method?` , `export?` , `annotations`
`TypeDef`	UDT `type Name` body (fields + methods as stmts)
`EnumDef`	`enum Name` members
`Assign`	`target` , optional `value` / `type` / `mode` (`Var` / `VarIp`), `export?` , `annotations`
`ReAssign`	`:=` reassignment
`AugAssign`	`+=` … ops
`Import`	`namespace/name/version` , optional `alias`
`Expr`	Expression statement
`Break` / `Continue`	Loop control

All `stmt` and most other attributed products carry location attributes.

Expressions (`expr`)

Operators and primaries:

- `BoolOp` , `BinOp` , `UnaryOp` , `Conditional` (`? :`), `Compare`
- `Call` , `Constant` , `Attribute` , `Subscript` , `Name` , `Tuple`
- **Structures as expressions:** `ForTo` , `ForIn` , `While` , `If` , `Switch` — same shapes can appear as values (Pine's expression-oriented control flow)
- `Qualify` — `const` / `input` / `simple` / `series` applied to a type/expression
- `Specialize` — generic specialization form (`value` + `args`)

Auxiliaries

Product	Role
`decl_mode`	`Var` | `VarIp`
`type_qual`	`Const` | `Input` | `Simple` | `Series`
`expr_context`	`Load` | `Store`
`param`	Function/method parameter
`arg`	Call argument (optional keyword `name`)
`case`	Switch arm (`pattern?` , `body`)
`cmnt` / `Comment`	Comment with `value` and `kind` (annotation classification)

Operators are empty product constructors (`Add` , `And` , `Eq` , …) used as tags — the unparser maps class names back to tokens.

Importing nodes

```
from pynescript.ast import node as ast

script = ast.Script(body=[
    ast.Expr(value=ast.Call(
        func=ast.Name(id="plot", ctx=ast.Load()),
        args=[ast.Arg(value=ast.Name(id="close", ctx=ast.Load()))],
    ))
])
```

Prefer constructing via `parse()` unless synthesizing trees for tests.

Internals

Paths

Path	Edit?
`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl`	**Yes** — schema source of truth
`src/pynescript/ast/grammar/asdl/generated/PinescriptASTNode.py`	**No** — regenerated
`src/pynescript/ast/grammar/asdl/tool/generate.py`	Generator entry
`src/pynescript/ast/node.py`	Thin re-export (`from .grammar.asdl.generated import *`)

Generated class shape

Excerpt of generated form (do not edit this file; shown for orientation):

```
@_dataclasses.dataclass
class Script(mod):
    body: list[stmt] = field(default_factory=list)
    annotations: list[string] = field(default_factory=list)
    _fields: ClassVar[list[str]] = ["body", "annotations"]
```

`stmt` base injects location `_attributes`. Hashability is forced to `object.__hash__` so nodes can live in sets/maps by identity when needed.

Regeneration

When the schema changes:

```
# Project convention (pyasdl / asdl tool — see asdl/tool/generate.py)
pyasdl src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl \
  -o src/pynescript/ast/grammar/asdl/generated
```

Then update, in the same change set:

1. `builder.py` construction sites
2. `unparser.py` `visit_*` methods
3. Evaluator expression/statement handlers

4. Any isinstance checks in LSP features

Dual nature of structures

ASDL places `If`, `ForTo`, `ForIn`, `While`, `Switch` under `expr`, while the builder often wraps structure-as-statement under `Expr(value=…)`. Annotation collection (`StatementCollector`) treats assign/reassign/augassign/expr that hold structure values specially so nested statements remain visible for `//@…` pairing.

Invariants

1. **Schema first.** New language constructs need an ASDL constructor (or a clear encoding into an existing one) before evaluator semantics.
2. **Fields vs attributes.** Visitors walk `_fields` only. Location is metadata; missing locations are fillable via `fix_missing_locations`.
3. **Optional flags as `int?`.** `method` and `export` are integer flags (truthy when set), not booleans — match builder assignments (`export = 1`).
4. **Generated file is not a style playground.** Formatting/lint of `PinescriptASTNode.py` is overwritten on regen.

Worked examples

Map ASDL to a short script

```
//@version=5
f(x) => x + 1
a = f(close)
```

Approximate tree:

```
Script(
  annotations=["//@version=5", ...],
  body=[
    FunctionDef(name="f", args=[Param(name="x")], body=[Expr(BinOp(...))]),
    Assign(target=Name("a", Store), value=Call(...)),
  ],
)
```

Dump fields programmatically

```
from pynescript.ast.helper import parse, iter_fields

tree = parse("x = 1")
for name, value in iter_fields(tree):
    print(name, type(value).__name__)
```

Failure modes

Failure	Cause
`AttributeError` on new field	Schema regenerated but consumer not updated
Builder constructs wrong product	ASDL and grammar out of sync
Unparser omits new node	Missing `visit_NewNode` → `generic_visit` may emit nothing useful
Hand-edit of generated nodes lost	Edited `generated/` instead of `.asdl`

See also

- Builder
- Visitor / transformer
- Unparser
- Grammar (ANTLR4)

BUILDER

title: "AST Builder" description: "PinescriptASTBuilder: ANTLR parse-tree visitor that constructs ASDL nodes, locations, and comment kinds."

AST BUILDER

The builder is the semantic boundary between ANTLR's concrete parse tree and PYNE's ASDL abstract tree.

Abstract

`src/pynescript/ast/builder.py` defines three cooperating mixins:

1. **`PinescriptASTLocator`** — maps `ParserRuleContext.start/stop` tokens to `lineno` / `col_offset` / `end_*`.
2. **`PinescriptCommentParser`** — classifies `//…` text into annotation kinds (`@=S` , `@0F` , `#` , plain `//` , …).

3. **`PinescriptASTBuilder`** — subclasses generated `PinescriptParserVisitor`, implements `visit_*` for every meaningful rule, returns ASDL nodes.

`helper._parse` constructs a builder, runs `builder.visit(rule_context)`, then (in exec mode) uses `StatementCollector` + comment tokens to attach annotations. The builder itself does not walk the default token channel for comments; that is a post-pass.

Conceptual model

```
sequenceDiagram
  participant H as helper._parse
  participant L as PinescriptLexer
  participant P as PinescriptParser
  participant B as PinescriptASTBuilder
  participant C as StatementCollector

  H->>L: tokenize stream
  H->>P: start_script / start_expression
  P-->>H: parse tree root ctx
  H->>B: visit(root)
  B-->>H: Script | Expression
  H->>C: visit(Script) → stmts
  H->>B: _parseComment on COMMENT tokens
  H->>H: _add_annotations(script, stmts, comments)
```

Interface surface

You rarely instantiate the builder directly; `parse()` does. For tools that already hold a parse tree:

```
from pynescript.ast.builder import PinescriptASTBuilder
from pynescript.ast.grammar.antlr4.parser import PinescriptParser
# ... construct parser, parse to ctx ...
node = PinescriptASTBuilder().visit(ctx)
```

Locator API

Method	Role
`_getLocations(ctx)`	Dict of four location keys
`_setLocations(node, ctx)`	Mutates node attributes from `ctx.start` / `ctx.stop`

End column calculation accounts for embedded newlines in the stop token text.

Comment kinds

`_parseComment` returns `(kind, parts)`:

Pattern	Kind prefix	Suffix	Example
`//@version = 5`	`@=`	`S` (script)	version assignment
`//@description …`	`@0`	`S`	script description
`//@function` / `@returns`	`@0`	`F`	function docs
`//@type`	`@0`	`T`	type docs
`//@variable`	`@0`	`V`	variable docs
`//@param name …`	`@1`	`F`	param docs
`//@field name …`	`@1`	`T`	field docs
`//# region` / `endregion`	`#`	—	editor regions
other `//`	`//`	—	ordinary comment

`helper._add_annotations` filters by suffix when attaching to nodes.

Store context

`_set_store_ctx` walks assignment targets (`Name`, nested `Tuple`) and sets `ctx = Store()` so load/store distinction is available to later passes (Python-ast style).

Representative visit methods

Rule visitor	Emits
`visitStart_script`	`Script(body)`
`visitStart_expression`	`Expression(body)`
`visitFunction_declaration` / `visitMethod_declaration`	`FunctionDef` (`method=1` for methods)
`visitType_declaration`	`TypeDef`
`visitEnum_declaration`	`EnumDef`
`visitCompound_*` / `visitSimple_*` assignment family	`Assign` / `ReAssign` / `AugAssign`
`visitIf_structure` / `visitFor_*` / `visitWhile_*` / `visitSwitch_*`	Structure expressions
`visit*_*_expression` (disjunction → … → primary)	Operator trees with correct associativity
`visitLiteral_*`	`Constant` (numbers via `ast.literal_eval`, strings, bools, colors)
`visitPrimary_expression_call`	`Call` + `Arg` list (positional/keyword)

Expression climbing follows the grammar's layered rules (`conditional` → `disjunction` → `conjunction` → equality → inequality → additive → multiplicative → unary → primary).

Internals

Path

- Implementation: `src/pynescript/ast/builder.py`

- Visitor base: `src/pynescript/ast/grammar/antlr4/visitor.py` → generated `PinescriptParserVisitor`
- Node types: `src/pynescript/ast/node.py`
- Annotation pairing: `src/pynescript/ast/collector.py` + `helper._add_annotations`

Statement list flattening

`visitStatements` flattens each statement visitor's result: some rules return a **list** (simple multi-statement lines joined by commas), so the body is always a flat `list[stmt]`.

Export flag

When `EXPORT` is present on library declarations or compound name initialization, the builder sets `export = 1` on the corresponding node (integer flag per ASDL `int?`).

Field / enum members

Field definitions become `Assign`-like statements inside `TypeDef.body` (with optional `VarIp` mode). Enum members become assignment-shaped stmts under `EnumDef.body`. The unparser special-cases method members of types when pretty-printing.

Invariants

1. **Every constructed stmt/expr that has a source span should receive `_setLocations`** so LSP diagnostics and `get_source_segment` work.
2. **Visitor method names must track generated rule names.** Renaming a parser rule without updating the builder is a silent `generic_visit` fallback (wrong or empty trees).
3. **Do not edit `builder.py.bak`.** Live builder is `builder.py` only (stale backups are forbidden under project policy).
4. **Grammar and builder co-evolve.** A resource-only grammar change that adds rules is incomplete until `visit_*` exists.

Worked examples

What `x = close + 1` becomes

Rough structure after build:

```
Assign(
  target=Name(id="x", ctx=Store()),
  value=BinOp(
    left=Name(id="close", ctx=Load()),
    op=Add(),
    right=Constant(value=1),
  ),
)
```

Export const initialization

Grammar: `EXPORT? variable_declaration EQUAL structure_expression`

Builder sets `assign.export = 1` when `ctx.EXPORT()` is present — required for library export of typed constants.

Debugging a missing visitor

```
from pynescript.ast.helper import parse, dump
print(dump(parse(your_snippet), indent=2))
```

If a subtree is `None` or a raw unexpected type, find the rule in `PinescriptParser.g4` and the matching `visitRule_name` in `builder.py`.

Failure modes

Symptom	Cause
`AttributeError: '…Context' object has no attribute '…'`	Full parser regen changed accessors; builder outdated
Locations all zero / missing	Forgot `_setLocations` on new visit method
Annotations not attached	Comment kind suffix wrong, or statement not yielded by `StatementCollector`
Tuples not assignable	Store context not propagated through `Tuple.elts`
Structure expression lost	Wrong rule branch between statement vs expression path

See also

- Grammar (ANTLR4)
- ASDL schema
- Helper API
- Error model

HELPER API

title: "Helper API" description: "parse, unparse, dump, walk, literal_eval, and location utilities — the public AST surface."

HELPER API

`src/pynescript/ast/helper.py` is the stable library face of the language core. It orchestrates ANTLR, the builder, annotations, and the unparser behind functions deliberately similar to CPython's `ast` module.

Abstract

Call `parse` to obtain an ASDL tree; `dump` / `walk` / `iter_*` to inspect it; `unparse` to regenerate source; `literal_eval` for constant-folding-safe evaluation of literal expressions. Location helpers (`copy_location`, `fix_missing_locations`, `get_source_segment`, `increment_lineno`) support tooling that rewrites or reports on trees.

Everything higher in the stack (CLI, LSP, Pro API, workers) ultimately routes through this module for front-end work.

Conceptual model

```
flowchart TB
  subgraph public ["Public API"]
    parse
    unparse
    dump
    walk
    literal_eval
    locs["location helpers"]
  end

  subgraph private ["Private pipeline"]
    _pis["_parse_inputstream / _parse_filestream"]
    _parse
    LEX["Lexer + TokenStream"]
    PAR["Parser + ErrorListener"]
    BLD["ASTBuilder.visit"]
    COL["StatementCollector"]
    ANN["_add_annotations"]
  end

  parse --> _pis --> _parse
  _parse --> LEX --> PAR --> BLD
  BLD --> COL --> ANN
  unparse --> UP["NodeUnparser"]
  literal_eval --> NLE["NodeLiteralEvaluator"]
```

Interface surface

`parse(source, filename="<unknown>", mode="exec") -> AST`

Parameter	Meaning
`source`	Full script or expression text
`filename`	Error reporting name; resolved to absolute path if the file exists
`mode`	`"exec"` → `start_script` → `Script`; `"eval"` → `start_expression` → `Expression`

Raises:

- `ValueError` — invalid mode
- `pynescript.ast.error.SyntaxError` — lexer/parser failure (via `PinescriptErrorListener`)

Side effects during exec mode:

1. Temporarily raises `sys.getrecursionlimit` to at least 5000.
2. Collects COMMENT tokens and attaches annotations to eligible statements.

`unparse(node) -> str`

Delegates to `NodeUnparser().visit(node)`. See Unparser.

`dump(node, *, annotate_fields=True, include_attributes=False, indent=None) -> str`

Recursive pretty-print of node trees. With `indent` (int spaces or string), multi-line layout is used for non-trivial nodes.

`literal_eval(node_or_string, context=None, data_feed=None, data_provider=None) -> Any`

- Strings are `parse(..., mode="eval")` first.
- Unwraps `Expression.body`.
- Uses `NodeLiteralEvaluator` — **not** full script evaluation. Non-literal graphs raise.

Tree navigation

Function	Behavior
`iter_fields(node)`	Yields `(name, value)` for set `_fields`
`iter_child_nodes(node)`	Direct AST children (incl. list items)
`walk(node)`	BFS over the full subtree

Location utilities

Function	Behavior
`copy_location(new, old)`	Copy lineno/col/end_* when both define the attribute
`fix_missing_locations(node)`	Fill gaps from parent defaults starting at `(1, 0)`
`increment_lineno(node, n=1)`	Shift all line numbers
`get_source_segment(source, node, *, padded=False)`	Slice original source by node span; `None` if incomplete ends

`__all__` export list

`copy_location`, `dump`, `fix_missing_locations`, `get_source_segment`, `increment_lineno`, `iter_child_nodes`, `iter_fields`, `literal_eval`, `parse`, `unparse`, `walk`.

Internals

Path

`src/pynescript/ast/helper.py` — all public helpers above.

Related:

- `src/pynescript/ast/collector.py` — `StatementCollector` for annotation pairing
- `src/pynescript/ast/unparser.py` — unparse backend
- `src/pynescript/ast/evaluator` — `NodeLiteralEvaluator` (literal_eval only)

Annotation algorithm (`_add_annotations`)

1. Merge comments and statements; sort by `(lineno, col_offset)`.
2. Group consecutive comments vs statements.
3. Keep only kinds starting with `@`.

4. Script-level: first group members with kind ending in `S` → `script.annotations`.
5. Pair remaining comment groups with following statements; attach `F` / `T` / `V` filters to `FunctionDef` / `TypeDef` / `Assign`.

Comment **nodes** are not left in `Script.body`; only string annotation lists are stored on targets.

Stream helpers

Internal	Role
`_parse_inputstream`	`InputStream(source)` + `stream.name = filename`
`_parse_filestream`	`FileStream` for on-disk scripts
`_get_absolute_path`	Resolve existing paths; leave `<unknown>` alone

Invariants

1. **`mode` is a closed set.** Only `exec` and `eval`.
2. **Default listeners are removed.** Console ANTLR spam is replaced by raising project `SyntaxError`.
3. **`dump` requires an `AST` instance.** Non-nodes raise `TypeError`.
4. **Round-trip tests should use `parse` + `unparse`**, not raw builder access, so annotation behavior matches production.

Worked examples

Inspect a tree

```
from pynescript.ast.helper import parse, dump, walk

tree = parse("""
//@version=5
indicator("demo")
plot(close)
""")
print(dump(tree, indent=2))
print(sum(1 for _ in walk(tree)), "nodes")
```

Source segment

```
from pynescript.ast.helper import parse, get_source_segment

src = "a = 1\nb = a + 2\n"
tree = parse(src)
assign_b = tree.body[1]
print(get_source_segment(src, assign_b))  # "b = a + 2\n" or similar span
```

Literal evaluation

```
from pynescript.ast.helper import literal_eval

assert literal_eval("2 * 3 + 4") == 10
assert literal_eval("'hi'") == "hi"
```

Failure modes

Failure	Notes
Deeply nested ternaries hit recursion	Mitigated by temporary limit ≥ 5000; pathological depth can still fail
`get_source_segment` returns `None`	Missing `end_lineno` / `end_col_offset`
`literal_eval` raises	Non-literal AST (names, calls beyond allowed set)
Filename `<unknown>` in errors	Expected when parsing pure strings without a path
Annotations missing	Comment not `@…` form, or not immediately preceding eligible stmt

See also

- Builder
- Unparser
- Visitor / transformer
- Error model

UNPARSER

title: "Unparser" description: "NodeUnparser: precedence-aware AST → Pine Script source regeneration."

UNPARSER

The unparser is the inverse of the builder: ASDL nodes become source text with correct operator parenthesization and Pine idioms (`=>`, `:=`, `var`/`varip`, `export`, method vs function).

Abstract

`src/pynescript/ast/unparser.py` implements `NodeUnparser(NodeVisitor)` and a `Precedence` enum. `helper.unparse(node)` constructs a fresh unparser and returns `"".join` of the internal buffer. Output is **semantically faithful**, not a pretty-printer of original trivia: whitespace is normalized; comments that never became annotations are not reconstructed; annotation strings that *were* attached are re-emitted.

Round-trip tests (`parse → unparse → parse`) are the primary oracle for grammar/builder/unparser coherence.

Conceptual model

```
flowchart LR
  AST["ASDL tree"]
  V["visit / traverse"]
  PR["Precedence table"]
  BUF["_source list"]
  OUT["source string"]

  AST --> V
  PR --> V
  V --> BUF --> OUT
```

Parenthesization rule of thumb: if a child expression's stored precedence is **weaker** (higher binding needed) than the parent operator, wrap the child in `()` .

Interface surface

```
from pynescript.ast.helper import parse, unparse

src = """
//@version=5
f(x) => x * 2
plot(f(close))
"""
print(unparse(parse(src)))
```

`Precedence` (low → high binding)

Level	Operators / forms
`TEST`	ternary `? :` (weakest)
`OR`	`or`
`AND`	`and`
`EQ`	`==` `!=`
`INEQ` / `CMP`	`<` `>` `<=` `>=`
`ARITH`	`+` `-`
`TERM`	`*` `/` `%`
`FACTOR` / `NOT`	unary `+` `-` `not`
`ATOM`	names, literals, calls, subscripts

`Precedence.next()` steps one level tighter for right-hand associativity control.

Buffer helpers (internal but useful when subclassing)

Method	Role
`write(*text)`	Append fragments
`fill(text="")`	Newline + indent + text
`block()`	Indent context manager
`delimit(start, end)`	Wrap sub-output
`require_parens(prec, node)`	Conditional parentheses
`items_view` / `interleave`	Comma-separated lists
`buffered()`	Capture substring generation

Node coverage (high level)

Node	Emission sketch
`Script`	annotations then body
`FunctionDef`	optional `export` / `method`, `name(args) => body` (inline single `Expr` or indented block)
`TypeDef`	`type Name` with fields then methods
`EnumDef`	`enum Name` body
`Assign`	annotations, `export?`, `var` / `varip?`, type?, target `= value`
`ReAssign`	`target := value`
`AugAssign`	`target op= value`
`ForTo` / `ForIn` / `While` / `If` / `Switch`	Pine control syntax; `else if` chain collapse
`Import`	`import ns/name/version as alias?`
`BinOp` / `BoolOp` / `UnaryOp` / `Compare` / `Conditional`	precedence-aware
`Call` / `Attribute` / `Subscript` / `Name` / `Constant` / `Tuple`	primaries

Internals

Path

- `src/pynescript/ast/unparser.py` — `Precedence`, `NodeUnparser`
- Entry: `helper.unparse` → `NodeUnparser().visit(node)`

Visit vs traverse

`visit` resets `_source` and calls `traverse`. `traverse` accepts lists (statement bodies) or single nodes, then dispatches via the visitor cache. This split lets statement lists and expression trees share code without double-buffering.

If / else if chains

`visit_If` detects `orelse == [Expr(If(...))]` and emits `else if` rather than nested `else` + `if` blocks — matching idiomatic Pine.

Type body ordering

`visit_TypeDef` partitions body into field statements vs `FunctionDef` with `method` set, emitting fields first then methods.

Constants

String constants use JSON-style quoting helpers where needed; colors and numbers re-serialize from their Python values. Exact original quote style (single vs double) is not guaranteed.

Invariants

1. **Precedence tables must stay aligned with the parser's expression layers.** A mismatch causes either redundant parens (benign) or wrong binding after re-parse (severe).
2. **Annotations are part of the round-trip surface** when attached to Script/FunctionDef/TypeDef/Assign.
3. **Unparser does not type-check.** Ill-typed but well-shaped trees still print.
4. **Indent unit is four spaces** (`" " * self._indent`).

Worked examples

Precedence

```
from pynescript.ast.helper import parse, unparse

# Builder produces BinOp(Add, left=1, right=BinOp(Mult, 2, 3)) or inverse depending on parse
print(unparse(parse("1 + 2 * 3", mode="eval")))
# Expect something that re-parses to the same value: e.g. "1 + 2 * 3"
```

Function forms

```
# Single-expression body → f(x) => expr
# Multi-statement body → f(x) =>\n    stmt\n    stmt
```

Reassignment vs declaration

```
// Assign with mode → var float x = 1.0
// ReAssign → x := 2.0
// AugAssign → x += 1
```

Failure modes

Symptom	Cause
Missing `visit_*` output	Falls through `generic_visit` which only walks children — may emit empty string for leaf-like unhandled nodes
Extra parentheses everywhere	Over-conservative precedence
Re-parse differs in binding	Under-parenthesized output
Lost comments	Only annotation strings are stored; plain `//` comments are dropped after parse
Method printed as function	`FunctionDef.method` flag not set by builder

See also

- Helper API
- Builder
- ASDL schema
- Visitor / transformer

VISITOR TRANSFORMER

title: "Visitor & Transformer" description: "NodeVisitor and NodeTransformer: read-only traversal vs in-place AST rewrite with caching dispatch."

VISITOR & TRANSFORMER

Tree walks in PYNE follow the same dual pattern as CPython's `ast` module: a visitor for analysis, a transformer for structural rewrite.

Abstract

- **`NodeVisitor`** (`src/pynescript/ast/visitor.py`) — dispatch `visit_<ClassName>`, default `generic_visit` walks children via `iter_fields`. Method lookups are cached per visitor instance.

- **`NodeTransformer`** (`src/pynescript/ast/transformer.py`) — same dispatch, but `generic_visit` **replaces** list items and fields according to return values (`None` removes, list splices, AST replaces).

Specialized visitors elsewhere:

Class	Path	Role
`PinescriptASTBuilder`	`builder.py`	CST visitor (ANTLR), not `NodeVisitor`
`NodeUnparser`	`unparser.py`	Source emission
`StatementCollector`	`collector.py`	Yield statements for annotations
Evaluator classes	`evaluator/*.py`	Runtime interpretation

Conceptual model

```
flowchart TB
  N["AST node"]
  D["visit(node)"]
  CACHE["_visitor_cache"]
  M["visit_ClassName or generic_visit"]

  N --> D --> CACHE
  CACHE --> M

  subgraph visitor ["NodeVisitor.generic_visit"]
    V1["for field in iter_fields"]
    V2["visit children"]
    V3["discard return values"]
  end

  subgraph transformer ["NodeTransformer.generic_visit"]
    T1["for field in iter_fields"]
    T2["visit child"]
    T3["None → drop"]
    T4["list → extend"]
    T5["AST → replace"]
  end

  M --> visitor
  M --> transformer
```

Interface surface

`NodeVisitor`

```
from pynescript.ast.visitor import NodeVisitor
from pynescript.ast.helper import parse

class CallCounter(NodeVisitor):
    def __init__(self):
        super().__init__()
        self.count = 0

    def visit_Call(self, node):
        self.count += 1
        self.generic_visit(node)

tree = parse("plot(ta.sma(close, 14))")
c = CallCounter()
c.visit(tree)
assert c.count >= 2
```

Method	Contract
`visit(node)`	Resolve `visit_<type(node).__name__>` with cache; fallback `generic_visit`
`generic_visit(node)`	Recurse into AST fields and lists of AST

Returns are whatever the override returns (analysis often returns `None`).

`NodeTransformer`

```
from pynescript.ast import node as ast
from pynescript.ast.transformer import NodeTransformer
from pynescript.ast.helper import parse, unparse

class RenameClose(NodeTransformer):
    def visit_Name(self, node: ast.Name):
        if node.id == "close":
            return ast.Name(id="open", ctx=node.ctx)
        return node

tree = parse("plot(close)")
tree = RenameClose().visit(tree)
assert "open" in unparse(tree)
```

Return value semantics when transforming a **list parent** (e.g. `body`):

Return	Effect
`None`	Remove this element
`AST`	Replace element
non-AST iterable	Splice multiple nodes in place of one
same node	Keep

For a **single AST field**, `None` deletes the attribute (`delattr`); otherwise `setattr` replaces.

`StatementCollector`

Generator-style visitor used only for annotation attachment:

- Yields `FunctionDef` , `TypeDef` , `EnumDef` , `Assign` , `ReAssign` , `AugAssign` , `Import` , `Expr` , `Break` , `Continue` .
- Descends into nested structures and into structure-valued assignments.
- Does **not** yield the structure nodes themselves as statements when they appear only as values — it yields their inner bodies.

```
from pynescript.ast.collector import StatementCollector
from pynescript.ast.helper import parse

stmts = list(StatementCollector().visit(parse("f() => 1\nx = 2")))
```

Internals

Paths

File	Symbols
`src/pynescript/ast/visitor.py`	`NodeVisitor`
`src/pynescript/ast/transformer.py`	`NodeTransformer`
`src/pynescript/ast/collector.py`	`StatementCollector` , `Structure` tuple
`src/pynescript/ast/helper.py`	`iter_fields` , `iter_child_nodes` , `walk`

Dispatch cache

```
node_class = type(node).__name__
visitor = cache.get(node_class) or getattr(self, "visit_" + node_class, generic_visit)
```

Subclass instances that dynamically add methods after first visit of a type will not see them unless the cache is cleared — prefer defining methods on the class body.

In-place lists

Transformers mutate list fields with `old_value[:] = new_values` rather than rebinding, so aliased references to the same list see updates.

Invariants

1. **Always call** `super().__init__()` on subclasses so `_visitor_cache` exists.
2. **Transformers should return the node** from overrides that use `generic_visit` (the default returns `node` after rewriting children).
3. **Do not assume identity stability** after a transformer pass if you replaced roots.
4. `walk` **is not a visitor** — it is a pure BFS helper without dispatch.

Worked examples

Strip all `Expr` statements that call `plot`

```
from pynescript.ast import node as ast
from pynescript.ast.transformer import NodeTransformer
from pynescript.ast.helper import parse, unparse

class DropPlots(NodeTransformer):
    def visit_Expr(self, node):
        if isinstance(node.value, ast.Call):
            func = node.value.func
            if isinstance(func, ast.Name) and func.id == "plot":
                return None
        return self.generic_visit(node)

tree = DropPlots().visit(parse("x = 1\nplot(x)\n"))
print(unparse(tree))
```

Collect all string constants

```
from pynescript.ast.visitor import NodeVisitor
from pynescript.ast.helper import parse

class Strings(NodeVisitor):
    def __init__(self):
        super().__init__()
        self.found = []

    def visit_Constant(self, node):
        if isinstance(node.value, str):
            self.found.append(node.value)

s = Strings()
s.visit(parse('indicator("hello")'))
```

Failure modes

Failure	Cause
`AttributeError: _visitor_cache`	Forgot `super().__init__()`
Transformer silently no-ops	Override returns nothing (`None`) unintentionally → node deleted
Infinite recursion	`visit_X` calls `self.visit(node)` on same node without change
Partial rewrite	Overrode `visit_*` without `generic_visit` — children untouched
Collector misses nested assign	Structure not in `Structure` tuple and not assigned as value

See also

- Helper API
- Unparser

- Builder
- ASDL schema

TYPE SYSTEM

title: "Type System" description: "Pine v6 type model: qualifiers, builtins, collections, UDTs, TypeRegistry, and MethodResolver."

TYPE SYSTEM

PYNE's type layer models Pine's *qualified* types and user-defined types (UDTs) as Python objects usable by analysis and evaluation — not as a full independent typechecker IR (yet), but as the shared vocabulary for "what kind of value is this?"

Abstract

`src/pynescript/ast/type_system.py` defines:

- **Qualifiers** — `const`, `simple`, `series`, `input` (`TypeQualifier`)

- **Builtin kinds** — `int`, `float`, `bool`, `string`, `color`, `na`, `enum`
- **Composite types** — `array<T>`, `matrix<T>`, `map<K,V>`
- **UDTs** — `UserDefinedType` with `Field`s and `MethodSignature`s
- **Runtime instances** — `ObjectInstance` for UDT values
- **Registry / resolution** — `TypeRegistry`, `MethodResolver` (`.new`, `.copy`, user methods)

ASDL already has `Qualify` / `Specialize` expression forms and `type_qual` constructors; the type system module is the **semantic** counterpart used when evaluating UDT construction and when tools reason about types.

Conceptual model

```
classDiagram
  class Type {
    +name: str
    +qualifier: TypeQualifier?
    +is_compatible_with(other)
  }
  class BuiltinType {
    +kind: BuiltinTypeKind
  }
  class ArrayType {
    +element_type: Type
  }
  class MatrixType {
    +element_type: Type
  }
  class MapType {
    +key_type: Type
    +value_type: Type
  }
  class UserDefinedType {
    +fields: dict
    +methods: dict
    +is_exported: bool
  }
  Type <|-- BuiltinType
  Type <|-- ArrayType
  Type <|-- MatrixType
  Type <|-- MapType
  Type <|-- UserDefinedType
  UserDefinedType --> Field
  UserDefinedType --> MethodSignature
  ObjectInstance --> UserDefinedType
  TypeRegistry --> UserDefinedType
  TypeRegistry --> BuiltinType
  MethodResolver --> TypeRegistry
```

Interface surface

Qualifiers

```
from pynescript.ast.type_system import TypeQualifier

TypeQualifier.CONST   # compile-time constant
TypeQualifier.SIMPLE  # bar-invariant
TypeQualifier.SERIES  # per-bar series
TypeQualifier.INPUT   # input.* style parameter
```

`Type.__str__` renders `"series float"` when a qualifier is set.

Builtins and factories

```
from pynescript.ast.type_system import (
    BuiltinTypeKind,
    int_type,
    float_type,
    bool_type,
    string_type,
    color_type,
    TypeQualifier,
)

t = float_type(TypeQualifier.SERIES)  # "series float"
```

Factory	Kind
`int_type`	`INT`
`float_type`	`FLOAT`
`bool_type`	`BOOL`
`string_type`	`STRING`
`color_type`	`COLOR`

Collections

```
from pynescript.ast.type_system import ArrayType, MatrixType, MapType, int_type, string_type

ArrayType(int_type())                # array<int>
MatrixType(float_type())             # matrix<float>
MapType(string_type(), int_type())   # map<string, int>
```

UDTs

```
from pynescript.ast.type_system import UserDefinedType, Field, MethodSignature, float_type, int_typ

point = UserDefinedType("point")
point.add_field(Field("x", float_type(), default_value=0.0))
point.add_field(Field("y", float_type(), default_value=0.0, varip=False))
point.add_method(MethodSignature("length", [], return_type=float_type()))
```

`Field` supports optional `default_value` and `varip` (mirrors `varip` fields in type declarations).

`TypeRegistry`

```
from pynescript.ast.type_system import TypeRegistry

reg = TypeRegistry()
reg.register_type(point)
assert reg.is_builtin_type("float")
assert reg.is_user_defined_type("point")
assert reg.get_type("float") is not None
```

Builtins are seeded in `_init_builtin_types` (`int`, `float`, `bool`, `string`, `color`, `na`, `enum`).

`ObjectInstance` + `MethodResolver`

```
from pynescript.ast.type_system import ObjectInstance, MethodResolver

inst = ObjectInstance(point)
inst.set_field("x", 1.5)
resolver = MethodResolver(reg)
# .new / .copy handled specially
copy = resolver.resolve_method(inst, "copy", [])
```

Method name	Behavior
`new`	Construct instance; positional args fill fields in declaration order
`copy`	Shallow copy of field dict
other	Lookup `UserDefinedType.methods`; missing → `AttributeError`

Unknown field get/set on `ObjectInstance` raises `AttributeError` with type name context.

Relation to ASDL `type_qual`

ASDL	`TypeQualifier`
`Const`	`CONST`
`Input`	`INPUT`
`Simple`	`SIMPLE`
`Series`	`SERIES`

Builder produces `Qualify(qualifier=…, value=…)` for annotated types in the AST; the type system module is used when those need runtime/type-registry meaning (especially UDTs in builtins/evaluator).

Internals

Path

`src/pynescript/ast/type_system.py` — sole definition site for the classes above.

Consumers (outside this doc's page set, for orientation):

- Evaluator UDT / collection code (`evaluator/builtins/`, `matrix_evaluator`, etc.)
- Future/static analysis hooks (LSP hover may grow toward this model)

Compatibility

`Type.is_compatible_with` is currently shallow: `BuiltinType` compares by equality; the base method returns `False` for other pairs. Treat richer subtyping (series/simple lattice, numeric promotion) as **not fully encoded** here — bar-loop semantics still own much of the coercion behavior.

`na` and `enum`

Registered as builtin kinds for name lookup. Enum *definitions* in scripts appear as AST `EnumDef`; linking enum members into `BuiltinTypeKind.ENUM` instances is evaluator-side work.

Invariants

1. **Registry lookup prefers builtins** over UDTs of the same name.
2. **Field presence is schema-true.** Setting an undeclared field is an error (no open bags).
3. **`.new` does not type-check arguments** against field types in the resolver — it assigns positionally.
4. **`varip` is metadata on `Field`**, not a separate type qualifier enum member.

Worked examples

Model a simple UDT from Pine

```
type candle
    float o
    float h
    float l
    float c
```

```
from pynescript.ast.type_system import UserDefinedType, Field, float_type, TypeRegistry, MethodResc

candle = UserDefinedType("candle")
for name in "o h l c".split():
    candle.add_field(Field(name, float_type()))
reg = TypeRegistry()
reg.register_type(candle)
c = MethodResolver(reg).resolve_method(ObjectInstance(candle), "new", [1, 2, 0.5, 1.5])
assert c.get_field("h") == 2
```

Qualified series float

```
from pynescript.ast.type_system import float_type, TypeQualifier
assert str(float_type(TypeQualifier.SERIES)) == "series float"
```

Failure modes

Failure	Cause
`AttributeError: Field '…' not found`	Typo or outdated UDT schema
`AttributeError: Method '…' not found`	Method never `add_method`'d; not `new` / `copy`
Silent type confusion	`is_compatible_with` too weak — do not rely on it alone for safety
Name shadowing	UDT registered with a builtin name is unreachable via `get_type`

See also

- ASDL schema — `Qualify`, `TypeDef`, `EnumDef`
- Runtime — bar-loop use of values
- Builder — parsing type declarations
- Linter — style-level checks (not full typing)

LINTER

title: "Linter" description: "PineLinter static rules: syntax via parse, version, deprecations, naming, style codes."

LINTER

The core linter is a lightweight static checker that sits on the parse pipeline plus regex rules. It is not a full dataflow analyzer; it is the first automated gate for “does this even look like modern Pine?”

Abstract

`src/pynescript/ast/linter.py` exports:

- `LintWarning` — dataclass (`code` , `message` , `line` , `column` , `severity`)
- `PineLinter` — stateful runner accumulating warnings

- `lint_script(source, filename)` / `lint_file(filepath)` — convenience entry points

Rules run in fixed order: **syntax** → **version** → **deprecated patterns** → **naming** → **style**. Syntax failures become severity `"error"` with code `E001`; other rules are mostly `"warning"`.

Conceptual model

```
flowchart TB
  SRC["source string"]
  L["PineLinter.lint"]
  S["_check_syntax → parse()"]
  V["_check_version"]
  D["_check_deprecated"]
  N["_check_naming"]
  STY["_check_style"]
  OUT["list of LintWarning"]

  SRC --> L --> S --> V --> D --> N --> STY --> OUT
```

Interface surface

```
from pynescript.ast.linter import lint_script, PineLinter, LintWarning

warnings = lint_script("""
indicator("x")
plot(close)
""")
for w in warnings:
    print(w)  # warning: [W001] Missing @version ... at line 1
```

`LintWarning`

Field	Type	Meaning
`code`	`str`	Stable id (`E001`, `W001`, `C002`, …)
`message`	`str`	Human-readable explanation
`line`	`int | None`	1-based when known
`column`	`int | None`	Reserved / optional
`severity`	`str`	`"warning"` (default) or `"error"`

`__str__` format: `{severity}: [{code}] {message} at {location}`.

`PineLinter.lint(source, filename="<input>") -> list[LintWarning]`

Resets `self.warnings`, runs all checks, returns the list (also stored on the instance).

File helper

```
from pynescript.ast.linter import lint_file

issues = lint_file("strategies/mean_reversion.pine")
```

Reads UTF-8 text then delegates to `lint_script`.

Rule catalog

Syntax

Code	Severity	Condition
`E001`	error	`parse(source, filename)` raises any exception; message includes exception text

Does not attempt recovery; one syntax error check per run.

Version

Code	Condition
`W001`	No `//@version = N` (flexible whitespace) match
`W002`	Version integer `< 5` (deprecated; suggest v5/v6)

Pattern: `//\s*@version\s*=\s*(\d+)`.

Deprecated patterns (`_check_deprecated`)

Code	Pattern (simplified)	Advice
`W101`	`security('EXCHANGE:SYM'…)` style	Prefer `request.security()` with explicit params
`W102`	`plot(… style=plot.style_histogram`	Consider `plotcandle`
`W103`	`var int name = na`	Prefer `0` for type safety

Matches set `line` from prefix newline count. Case-insensitive search.

Naming (`_check_naming`)

Code	Condition
`C001`	LHS of `name = ta.…` starts with lowercase — message suggests camelCase via `_to_camel`

Heuristic only: line-local regex `(\w+)\s*=\s*ta\.`.

Style (`_check_style`)

Code	Condition
`C002`	Line length (rstrip) > 120
`C003`	Line matches `^\s+if\s+` — flagged as “single-line if without braces” heuristic
`C004`	File does not end with newline

Internals

Path

`src/pynescript/ast/linter.py`

Dependencies:

- `pynescript.ast.parse` (re-exported path via `from pynescript.ast import parse`)
- Standard library `re`, `dataclasses`

Design stance

The linter intentionally uses **regex over AST** for several rules so it still partially works when parse fails (version/deprecations/style still run after a failed syntax check — note: `_check_syntax` records `E001` but does not abort the pipeline). That means:

- False positives/negatives on fancy formatting are possible.
- Deep semantic issues (wrong series type, undefined names) belong to evaluator/LSP diagnostics, not these codes.

Integration points

CLI, editor save hooks, and CI can call `lint_script` without spinning up the full evaluator. LSP diagnostics may layer additional semantic checks beyond this module.

Invariants

1. **`lint()` always clears prior warnings** on that instance before running.
2. **Codes are stable public strings** — treat renames as breaking for tooling.
3. **Syntax errors are not fatal to the function** — you get `E001` plus any later regex hits.
4. **No mutation of source** — pure analysis.

Worked examples

Clean modern script

```
from pynescript.ast.linter import lint_script

src = """//@version=5
indicator("ok")
length = 14
basis = ta.sma(close, length)
plot(basis)
"""
assert lint_script(src) == []  # may still flag C001 if naming heuristic fires
```

Note: `basis = ta.sma(...)` triggers `C001` under the current rule (lowercase LHS). Prefer documenting that heuristic when teaching style.

Missing version

```
ws = lint_script('indicator("x")\nplot(close)\n')
assert any(w.code == "W001" for w in ws)
```

Programmatic filter

```
errors = [w for w in lint_script(src) if w.severity == "error"]
if errors:
    raise SystemExit(1)
```

Failure modes

Issue	Explanation
`C003` on legitimate indented `if` blocks	Regex is coarse; multi-line `if` with body still matches `^\s+if\s+`
`C001` noise	Many valid snake_case or lower identifiers
`E001` opaque message	Exception string from parse; see Error model for caret formatting when catching `SyntaxError` directly
Encoding errors in `lint_file`	Non-UTF-8 files raise at read time
False security deprecation	Pattern looks for quoted `EXCHANGE:SYM` form only

See also

- Helper API — underlying `parse`
- Error model
- LSP diagnostics
- End-user troubleshooting

ERROR MODEL

title: "Error Model" description: "SyntaxError, IndentationError, SyntaxErrorDetails, and PinescriptErrorListener bridge from ANTLR to diagnostics."

ERROR MODEL

Parse failures in PYNE are not bare ANTLR console messages. They are structured exceptions with file, line, column, source excerpt, and a caret — suitable for CLI, tests, and LSP adapters.

Abstract

Two layers cooperate:

1. **`src/pynescript/ast/error.py`** — `SyntaxErrorDetails`, `SyntaxError`, `IndentationError`.

2. **`src/pynescript/ast/grammar/antlr4/error_listener.py`** — `PinescriptErrorListener` implements ANTLR's `ErrorListener.syntaxError`, builds details from the recognizer + offending token, and **raises** the project `SyntaxError`.

`helper._parse` installs this singleton listener on both lexer and parser after removing default listeners. Indentation problems raised from `PinescriptLexerBase` use the same exception hierarchy (`IndentationError` subclass).

Note: these names intentionally shadow Python builtins (`SyntaxError`, `IndentationError`) inside the package — import carefully:

```
from pynescript.ast.error import SyntaxError as PineSyntaxError
```

Conceptual model

```
sequenceDiagram
  participant U as User code / parse()
  participant L as Lexer / Parser
  participant EL as PinescriptErrorListener
  participant E as SyntaxError

  U->>L: tokenize / parse
  L->>EL: syntaxError(recog, symbol, line, col, msg, exc)
  EL->>EL: filename, line text, end span from token
  EL->>E: raise SyntaxError(msg, details)
  E-->>U: exception with caret __str__
```

Interface surface

`SyntaxErrorDetails` (NamedTuple)

Field	Type	Meaning
`filename`	`str`	Path or `<unknown>`
`lineno`	`int`	1-based line
`offset`	`int`	0-based column
`text`	`str`	Source line (or excerpt)
`end_lineno`	`int | None`	Multi-line span end
`end_offset`	`int | None`	End column

`SyntaxError(Exception)`

```
class SyntaxError(Exception):
    def __init__(self, message: str, *details): ...
```

`details` may be:

- a single `SyntaxErrorDetails` instance, or
- positional components matching the NamedTuple fields.

Attributes: `message`, `details`.

`__str__` renders:

```
{message}
  File "{filename}", line {lineno}
    {stripped_line}
    {spaces}^
```

Offset is adjusted when the displayed line is `lstrip()`'d so the caret still points at the logical column.

`IndentationError(SyntaxError)`

Empty subclass for indent-stack failures from the lexer base (wrong nest, inconsistent levels). Catch either specifically or via the base `SyntaxError`.

`PinescriptErrorListener`

```
from pynescript.ast.grammar.antlr4.error_listener import PinescriptErrorListener

listener = PinescriptErrorListener.INSTANCE  # singleton
lexer.removeErrorListeners()
lexer.addErrorListener(listener)
parser.removeErrorListeners()
parser.addErrorListener(listener)
```

Method	Role
`syntaxError(...)`	Build details; raise `SyntaxError`
`_getFilenameFrom(recognizer)`	Walk Parser → TokenStream → Lexer → InputStream/FileStream
`_getInputTextFrom(recognizer, lineno)`	Full buffer or single line
`_splitLines`	Portable newline split (`\r\n` / `\n` / `\r`)

If the ANTLR exception `e` is already a project `SyntaxError`, its `details` are updated; otherwise the new error's `__cause__` is set to `e`.

Internals

Paths

Path	Role
`src/pynescript/ast/error.py`	Exception types
`src/pynescript/ast/grammar/antlr4/error_listener.py`	ANTLR bridge
`src/pynescript/ast/grammar/antlr4/resource/PinescriptLexerBase.py`	May raise `IndentationError` / `SyntaxError` during tokenization
`src/pynescript/ast/helper.py`	Wires listener; maps filename onto streams

Filename resolution order (InputStream)

1. `getSourceName()` if present
2. `sourceName` attribute
3. `input_stream.name` (set by helper to the user filename)
4. `"<unknown>"`

`FileStream` uses `fileName`.

End span from offending token

```
symbol_len = stop - start + 1
end_lineno = token.line + newlines_in_text
end_offset = adjusted if multiline else column + symbol_len
```

Mirrors the builder's location math so diagnostics and AST spans speak the same coordinate system.

Relationship to linter `E001`

`PineLinter._check_syntax` catches **any** `Exception` from `parse` and wraps it as `LintWarning(code="E001", severity="error")`, losing structured details. For rich carets, catch `pynescript.ast.error.SyntaxError` at the call site instead of only reading linter output.

LSP / tooling

Adapters typically map:

Exception field	LSP `Diagnostic`
`lineno` / `offset`	`Range.start`
`end_lineno` / `end_offset`	`Range.end`
`message`	`message`
IndentationError	same, possibly different `code`

Invariants

1. **Default ANTLR listeners must stay removed** on the parse path — otherwise errors print twice and may not raise.
2. **Singleton listener is stateless** enough to share (`INSTANCE`); thread-local needs would require new instances.
3. **Raising aborts parse immediately** — no error recovery producing a partial tree through the public helper.
4. **Package `SyntaxError` ≠ builtin** — `except SyntaxError` without import may catch the wrong class.

Worked examples

Catching a parse error

```
from pynescript.ast.helper import parse
from pynescript.ast.error import SyntaxError as PineSyntaxError

try:
    parse("f( =>\n", filename="bad.pine")
except PineSyntaxError as e:
    print(e)  # caret form
    print(e.details.lineno, e.details.offset)
    print(e.details.filename)
```

Distinguishing indentation

```
from pynescript.ast.error import IndentationError as PineIndentError
from pynescript.ast.error import SyntaxError as PineSyntaxError

try:
    parse("if true\nplot(1)\n")  # missing indent under if — may indent-error depending on tokens
except PineIndentError as e:
    print("indent", e.message)
except PineSyntaxError as e:
    print("syntax", e.message)
```

Manual construction (tests)

```
from pynescript.ast.error import SyntaxError, SyntaxErrorDetails

details = SyntaxErrorDetails("t.pine", 3, 4, "    plot(\n", 3, 9)
err = SyntaxError("missing RPAR", details)
assert "t.pine" in str(err)
assert "^" in str(err)
```

Failure modes

Failure	Cause
`TypeError: unexpected type of input`	Listener received a recognizer/stream combo it does not support
Caret misaligned	Tabs vs spaces; display strips leading WS while offset counts raw
Filename always `<unknown>`	Stream name not set (bypassed helper)
Exception swallowed as generic `E001`	Only used linter path
`except SyntaxError` misses Pine errors	Caught builtin only
Partial trees	Not available via `parse()` after error — redesign would need recovery listeners

See also

- Helper API

- Grammar (ANTLR4)
- Linter
- LSP diagnostics
- Language core hub