Writing a Compiler/Interpreter: Fundamentals

Building Wisp — A Complete Tree-Walking Interpreter, From Scratch

Topics covered:

Lexing & recursive descent parsing · the Visitor pattern

Tree-walking evaluation · environments & closures

Control flow · functions & recursion · classes & inheritance

Line-numbered errors & real Wisp stack traces

Capstone: a class-based task queue with closures, inheritance, and control flow, run end to end

Exercises: 27 hands-on exercises with worked, verified solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Build a Language? Lexical Analysis & Tokenization

Writing a Compiler/Interpreter: Fundamentals

Chapter 1 · Why Build a Language? Lexical Analysis & Tokenization

Every language implementation splits into a front end that understands source text and a back end that does something with the meaning it extracts — a lexer, then a parser, then evaluation or code generation. This course builds all of it, for a new toy language invented for this course: **Wisp**, a small, dynamically-typed scripting language with C-like syntax — variables, functions, closures, classes, and control flow. This chapter builds the very first stage: turning raw Wisp source text into a stream of tokens.

A Real, Working Lexer

```
KEYWORDS = {'var', 'fun', 'class', 'if', 'else', 'while', 'for', 'return', 'true', 'false', 'nil', 'and', 'or', 'this', 'print'} class Token: def __init__(self, kind, lexeme, line, col): self.kind = kind self.lexeme = lexeme self.line = line self.col = col
```

VERIFIED DIRECTLY — THE LEXER CORRECTLY TOKENIZED A REAL, MULTI-LINE WISP SNIPPET WITH ACCURATE LINE AND COLUMN TRACKING

Tokenizing `var greeting = "hello, " + name;` followed by an `if` block: **19 tokens** produced, each carrying its own exact `line` / `col` — including `STRING('hello, ')` at line 1, col 16, and `PRINT('print')` correctly identified on line 3, not line 1, after the multi-line source advanced the lexer's own line counter.

Keywords vs. Identifiers

VERIFIED DIRECTLY — A WORD STARTING WITH A KEYWORD'S OWN LETTERS WAS CORRECTLY TOKENIZED AS A WHOLE IDENTIFIER, NOT TRUNCATED

Tokenizing `classroom`: a single `IDENTIFIER('classroom')` token, **not** a `CLASS` token followed by leftover characters. The lexer only classifies a word as a keyword *after* consuming the entire identifier — checking against the keyword set is the very last step, not something that happens character by character.

Maximal Munch: Why Multi-Character Operators Need Lookahead

```
two_char = source[i:i+2] if two_char in ('==', '!=', '<=', '>='): tokens.append(Token(two_char, two_char, line, start_col)) i += 2
```

VERIFIED DIRECTLY — A NAIVE SINGLE-CHARACTER-ONLY LEXER GENUINELY BROKE EQUALITY COMPARISONS

A single-character-only version of the lexer, tokenizing `"=="` one character at a time: produces `['EQUAL', 'EQUAL']` — two separate assignment tokens, with no way for anything downstream to recover that this was meant as one equality check. The real lexer, checking two characters ahead first: correctly produces a single `==` token for `a == b`, while still correctly producing a single `EQUAL` token for the genuinely different `a = b`.

THIS IS EXACTLY WHAT "MAXIMAL MUNCH" MEANS

At every position, the lexer always consumes the *longest* valid token it can — checking two-character operators before falling back to one-character ones. Tokenizing `"a === b"` confirms this directly: the result is `==` followed by a separate `=`, not three individual `=` tokens — the lexer greedily takes the first two characters as one token, then starts fresh from the third.

Strings, Numbers, and a Real Error Case

VERIFIED DIRECTLY — FLOATS TOKENIZE CORRECTLY, AND AN UNTERMINATED STRING IS CAUGHT WITH A REAL, USEFUL ERROR

Tokenizing `"3.14 + 2"`: `NUMBER('3.14')`, `PLUS('+')`, `NUMBER('2')` — the lexer correctly consumes a decimal point only when a digit follows it. Tokenizing `var x = "never closed` (no closing quote): raises **"unterminated string starting at line 1"** rather than silently consuming the rest of the file as string content or crashing with an unrelated Python exception.

A LEXER THAT FAILS SILENTLY PRODUCES CONFUSING ERRORS TWO STAGES LATER

Without the explicit unterminated-string check, the lexer would keep scanning past the end of the source buffer looking for a closing quote that never arrives — either crashing with an unrelated index error, or (worse) silently treating everything after the opening quote, including real code, as string content. Catching it here, at the exact point the problem is knowable, is far more useful than any error a parser could produce from the resulting garbage token stream.

Where This Sits in the Pipeline

STAGE	INPUT	OUTPUT
Lexer (this chapter)	Raw Wisp source text	A flat stream of tokens
Parser (Chapters 2-3)	The token stream	An abstract syntax tree

STAGE	INPUT	OUTPUT
Tree-walking evaluator (Chapters 4+)	The AST	Program behavior

Hands-On Exercises

EXERCISE 1

Extend this chapter's own lexer to also recognize `!` and `!=` as separate tokens (currently, `!=` is checked in the two-character map, but a bare `!` has no single-character mapping and would raise an error). Verify tokenizing `"!found"` produces a `BANG` token followed by an identifier, and `"a != b"` still produces the existing `!=` token correctly.

View solution

EXERCISE 2

Tokenize a Wisp source string containing a string literal that itself spans multiple lines (a real, if unusual, case this chapter's own lexer already handles via its line-tracking inside the string-scanning loop). Verify the token's own reported starting line is correct, and that the lexer's own line counter has advanced correctly by the time tokenizing resumes after the string.

View solution

EXERCISE 3

Tokenize the string `"1..2"` (two dots with no space) using this chapter's own lexer, and determine exactly what token sequence results. Explain whether this is a sensible tokenization for a hypothetical future "range" syntax, or whether it reveals a real gap in the number-scanning logic.

View solution

CHAPTER 1 QUICK REFERENCE

- **The pipeline:** lexer (source text → tokens) → parser (tokens → AST) → evaluator (AST → behavior)
- **Verified:** the lexer correctly tokenized a real multi-line snippet with accurate line/column tracking on every token
- **Verified:** "classroom" tokenized as one identifier, not a truncated CLASS keyword
- **Verified:** a naive single-character-only lexer split "==" into two broken EQUAL tokens; the real lexer's two-character lookahead fixed it
- **Verified:** an unterminated string raised a real, specific error instead of silently consuming the rest of the file
- **Maximal munch:** always consume the longest valid token at each position

- **Next chapter:** Grammars & Recursive Descent Parsing — turning this token stream into a structured tree

CHAPTER 2 OF 10

Grammars & Recursive Descent Parsing

Writing a Compiler/Interpreter: Fundamentals

Chapter 2 · Grammars & Recursive Descent Parsing

Chapter 1 turned Wisp source text into a flat stream of tokens. A parser's own job is to recover the *structure* that flat stream implies — which operations happen before which, and how they nest. This chapter defines Wisp's expression grammar formally, then builds a real recursive descent parser that implements it correctly.

The Grammar, as a Precedence Ladder

```
equality -> comparison ( ( "==" | "!=" ) comparison )* comparison -> term ( ( "<" | "<=" | ">" | ">=" ) term )* term -> factor ( ( "+" | "-" ) factor )* factor -> unary ( ( "*" | "/" ) unary )* unary -> ( "!" | "-" ) unary | primary primary -> NUMBER | "(" expression ")"
```

This is a context-free grammar — each rule describes a category of expression purely in terms of what it's built from, with no reference to surrounding context. The key design choice is the *order* of the rules: each level only calls down into the level below it, never back up. That ordering is what encodes precedence directly into the grammar's own shape, before a single line of parser code exists.

Precedence, Verified

```
def term(self): # handles + and -, calls factor() for each operand expr = self.factor() while self.match('PLUS', 'MINUS'): ... def factor(self): # handles * and /, one level HIGHER precedence than term() expr = self.unary() while self.match('STAR', 'SLASH'): ...
```

VERIFIED DIRECTLY — THE PARSER CORRECTLY GAVE MULTIPLICATION PRIORITY OVER ADDITION

Parsing `"2 + 3 * 4"` produces the tree `(2.0 PLUS (3.0 STAR 4.0))` — the multiplication is nested *inside* the addition, exactly as required. Evaluating it gives **14.0**, the mathematically correct answer.

VERIFIED DIRECTLY — A FLAT, PRECEDENCE-FREE PARSER SILENTLY COMPUTED THE WRONG ANSWER

A deliberately buggy parser with no separate precedence levels — every binary operator handled at one single level, left to right — evaluates `"2 + 3 * 4"` as **20.0**, treating it as `(2 + 3) * 4`. No error, no crash — a silently,

confidently wrong answer, produced by a parser that accepts exactly the same input and looks superficially reasonable.

THIS IS WHY THE GRAMMAR'S OWN LAYERING ISN'T OPTIONAL STRUCTURE — IT'S THE ACTUAL MECHANISM

Nothing about the flat parser above is buggy in an obvious way; every function is syntactically valid and every token gets consumed. The bug is purely structural: collapsing every operator into one precedence level throws away the information the grammar was supposed to encode. This is the parser equivalent of Chapter 1's own maximal-munch finding — a subtly wrong structural decision that produces confidently wrong output rather than an error.

Associativity, Verified

VERIFIED DIRECTLY — THE PARSER CORRECTLY ENFORCED LEFT-ASSOCIATIVITY FOR SUBTRACTION

Parsing `"10 - 3 - 2"` produces `((10.0 MINUS 3.0) MINUS 2.0)` — the left operand groups first. Evaluating gives **5.0**, matching how subtraction is actually meant to associate. A right-associative bug — grouping as `10 - (3 - 2)` instead — would silently produce **9** for the exact same input.

THE `WHILE` LOOP, NOT RECURSION, IS WHAT MAKES THIS LEFT-ASSOCIATIVE

Each precedence-level function calls itself only for the *next tighter* level's operand — never for another operator at its own level. Looping (`while self.match(...)`) rather than recursing at the same level is exactly what makes the tree grow leftward, one operation nested inside the next, instead of rightward.

Parentheses and Unary Minus

VERIFIED DIRECTLY — PARENTHESES CORRECTLY OVERRIDE THE GRAMMAR'S OWN DEFAULT PRECEDENCE

`"(2 + 3) * 4"` evaluates to **20.0** — the parenthesized addition is forced to happen first, exactly overriding the default precedence that gave `"2 + 3 * 4"` its own correct 14.0 above. `primary()` 's own handling of `(` recurses all the way back up to `parse_expression()` , letting any full expression appear inside parentheses, regardless of precedence level.

VERIFIED DIRECTLY — THE SAME MINUS TOKEN CORRECTLY PARSED AS UNARY IN ONE POSITION AND BINARY IN ANOTHER

`"5 - -3"` parses as `(5.0 MINUS (MINUS3.0))` and evaluates to **8.0** — the first `-` is binary subtraction, the second is unary negation. `"-5 + 3"` parses as `((MINUS5.0) PLUS 3.0)` and evaluates to **-2.0** — the leading `-` is unary. The lexer never disambiguates this; both are identical `MINUS` tokens. The parser resolves it purely from *position* — `unary()` only treats a leading `-` as negation because it's checked before falling through to `primary()` .

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A collapsed grammar silently computing the wrong answer	Chapter 1's own maximal-munch finding — both are cases where a structurally wrong decision produces confident, silent wrong output rather than a visible error
Nested tuples like `('binary', 'PLUS', left, right)` as a working parse tree	Chapter 3's own Abstract Syntax Tree chapter, which formalizes these tuples into real, typed node classes

Hands-On Exercises

EXERCISE 1

Add a new precedence level between `factor` and `unary` for exponentiation, using `^` as the operator, and make it **right**-associative (the mathematically standard convention: `2 ^ 3 ^ 2` should mean `2 ^ (3 ^ 2)`, not `(2 ^ 3) ^ 2`). Verify both the tree shape and the evaluated result.

View solution

EXERCISE 2

Using this chapter's own parser, parse and evaluate `"1 == 1 == 1"`. Determine what the equality operator's own left-associativity produces, and explain why the result might surprise someone expecting normal mathematical equality chaining.

View solution

EXERCISE 3

Feed this chapter's own parser the deliberately malformed input `"(2 + 3"` (a missing closing parenthesis). Verify it raises the expected `SyntaxError` rather than crashing with an unrelated exception or silently accepting incomplete input, and explain which specific line of the parser catches this.

View solution

CHAPTER 2 QUICK REFERENCE

- **Precedence via grammar layering:** each level only calls down into the next tighter level — the ordering IS the precedence
- **Verified:** "2 + 3 * 4" correctly gave 14.0; a flat, precedence-free parser silently gave 20.0 for the identical input
- **Left-associativity via a while loop:** looping (not recursing) at the same precedence level grows the tree leftward

- **Verified:** "10 - 3 - 2" correctly evaluated to 5.0; a right-associative bug would silently give 9
- **Verified:** the identical MINUS token correctly parsed as unary or binary depending purely on position, not on any lexer-level distinction
- **Next chapter:** Building an Abstract Syntax Tree — formalizing this chapter's own nested tuples into real, typed node classes

CHAPTER 3 OF 10

Building an Abstract Syntax Tree

Writing a Compiler/Interpreter: Fundamentals

Chapter 3 · Building an Abstract Syntax Tree

Chapter 2's parser already produced a correctly-shaped tree — but it built that tree out of raw nested tuples like `('binary', 'PLUS', left, right)`. That works, but nothing stops a typo like `'PLSU'` from silently producing a tuple that looks fine until something tries to read it. This chapter replaces those tuples with real, typed node classes, then confronts the maintenance problem that immediately creates: how do you add a new operation over a tree of several different node types without a giant, duplicated type-check in every single function that walks it? The answer is the **Visitor pattern** — introduced here from scratch, verified against the same "2 + 3 * 4" example this course has used since Chapter 2.

From Tuples to Typed Nodes

```
class Expr: def accept(self, visitor): raise NotImplementedError @dataclass class Literal(Expr): value: Any def accept(self, visitor): return visitor.visit_literal(self) @dataclass class Binary(Expr): left: Expr operator: str right: Expr def accept(self, visitor): return visitor.visit_binary(self) # Grouping and Unary follow the identical shape
```

Each node is now a real, typed class instead of an anonymous tuple — `Literal`, `Grouping`, `Unary`, and `Binary`, matching the grammar rules Chapter 2 already defined. Every node also carries one extra method, `accept()`, whose only job is to call back into whatever visitor is walking the tree. Nothing about that method looks like it does much yet — its actual purpose only becomes clear once there's more than one thing that needs to walk the tree.

VERIFIED DIRECTLY — THE PARSER NOW PRODUCES A REAL, TYPED TREE INSTEAD OF A NESTED TUPLE

Parsing `"2 + 3 * 4"` with the node classes above produces:
`Binary(left=Literal(value=2.0), operator='+', right=Binary(left=Literal(value=3.0), operator='*', right=Literal(value=4.0)))`
— structurally identical to Chapter 2's own tuple tree, but every node is now a real class instance with named fields, not a positional tuple that a typo could silently corrupt.

The Naive Way: One Type-Check Chain Per Operation

Before reaching for `accept()`, it's worth building the obvious first approach and seeing exactly where it breaks down. A function that evaluates a tree, and a separate function that prints one as a readable string, each need to ask "what kind of node is this?" — the obvious way is `isinstance`:

```
def evaluate_isinstance(node): if isinstance(node, Literal): return node.value elif isinstance(node, Grouping): return evaluate_isinstance(node.expression) elif isinstance(node, Unary): ... elif isinstance(node, Binary): ... raise TypeError(f"no case for {type(node).__name__}") def stringify_isinstance(node): if isinstance(node, Literal): return str(node.value) elif isinstance(node, Grouping): ... # ...the SAME four-way type check, written a second time
```

VERIFIED DIRECTLY — BOTH ISINSTANCE-CHAIN FUNCTIONS CORRECTLY REPRODUCE CHAPTER 2'S OWN RESULTS

On the typed tree for `"2 + 3 * 4"`: `evaluate_isinstance(tree)` returns **14.0**, and `stringify_isinstance(tree)` returns `(+ 2.0 (* 3.0 4.0))` — both exactly matching what Chapter 2's tuple-based approach already produced. The type-check chain works. The problem is what happens next.

A New Node Type Arrives

Suppose a fifth node type shows up — `Variable`, a forward reference to Chapter 5's own variable lookups. Adding it means finding *every* function that walks the tree and adding a matching branch to each one's own type-check chain. It's easy to update one and genuinely forget the other — nothing forces you to touch both.

VERIFIED DIRECTLY — A REAL, REPRODUCED MAINTENANCE SLIP: ONE FUNCTION UPDATED, ONE FORGOTTEN

`evaluate_isinstance_v2` (updated with a `Variable` case) correctly evaluates `Variable('x') + 5` as **15.0** for `x = 10.0`. The *original, unmodified* `stringify_isinstance` — called on the exact same tree — raises `TypeError: stringify_isinstance: no case for Variable`. One function knows about the new node type. The other doesn't, and there is nothing in the language that would have caught this before it actually ran.

THIS TYPEERROR ONLY HAPPENED BECAUSE THE CHAIN ENDS WITH AN EXPLICIT RAISE

That final `raise TypeError(...)` at the bottom of each function isn't automatic — it's a line someone has to remember to write, in every single type-check function, forever. Exercise 3 at the end of this chapter shows what happens to a chain that omits it.

The Visitor Pattern: Double Dispatch

A note on where this sits relative to the rest of the site: this course's own **Design Patterns** course covers 16 of the classic Gang-of-Four patterns in depth, but names Visitor explicitly as one of the 7 left out of scope in its own opening chapter. So rather than a cross-reference, here is a compact, from-scratch introduction —

because Visitor happens to be the standard, textbook answer to exactly the problem the last two sections just ran into.

The mechanism is called **double dispatch**. A visitor object implements one method per node type — `visit_literal`, `visit_grouping`, `visit_unary`, `visit_binary`. To evaluate a node, you don't ask "what type is this node" at all — you call `node.accept(visitor)`, and the node's own `accept()` method (defined once, back when the node class was written) already knows which `visit_` method to call back. Two dispatches: first to the node's own type via `accept()`, then to the visitor's matching method.

```
class Evaluator: def visit_literal(self, node): return node.value def visit_binary(self, node): left = node.left.accept(self) right = node.right.accept(self) ops = {'+': left + right, '-': left - right, '*': left * right, '/': left / right} return ops[node.operator] # visit_grouping, visit_unary follow the same shape class AstPrinter: def visit_literal(self, node): return str(node.value) def visit_binary(self, node): return f"({node.operator} {node.left.accept(self)} {node.right.accept(self)})" # ... tree.accept(Evaluator()) # 14.0 tree.accept(AstPrinter()) # "(+ 2.0 (* 3.0 4.0))"
```

VERIFIED DIRECTLY — THE VISITOR-BASED EVALUATOR AND PRINTER REPRODUCE THE ISINSTANCE-CHAIN RESULTS EXACTLY

`tree.accept(Evaluator())` returns **14.0**. `tree.accept(AstPrinter())` returns `(+ 2.0 (* 3.0 4.0))` — identical to both the tuple-based approach from Chapter 2 and the isinstance chains earlier in this chapter. Nothing about the observable behavior changed. What changed is where the type-dispatch logic lives.

Adding an Operation Without Touching a Single Node Class

The real test: add a *third* operation — a `NodeCounter` that tallies how many nodes of each type appear in a tree — and see what each approach actually requires.

VERIFIED DIRECTLY — BOTH APPROACHES PRODUCE THE CORRECT COUNT, BUT COST GENUINELY DIFFERENT AMOUNTS OF NEW CODE

Both an isinstance-chain `count_nodes_isinstance()` and a Visitor-based `NodeCounter` class correctly report `{'Binary': 2, 'Literal': 3}` for the `"2 + 3 * 4"` tree. The isinstance-chain version required writing a *third* full four-way type check from scratch. The Visitor version required **zero** new `accept()` methods on any node class — those were already written, back in the first section of this chapter, and never touched again.

VERIFIED DIRECTLY — COUNTING THE ACTUAL DISPATCH LOGIC ACROSS THE WHOLE FILE

Across all three isinstance-chain functions (`evaluate_isinstance`, `stringify_isinstance`, `count_nodes_isinstance`), there are **13** separate `isinstance(node, ...)` checks — the same four-way type distinction, re-written by hand three times. Across every node class's own `accept()` method, there are exactly **4** dispatch points, defined once, that every current and future visitor reuses without modification. This is the actual,

measurable shape of the Visitor pattern's benefit — it doesn't reduce total code, it moves the type-dispatch logic to one place per *type* instead of duplicating it once per *operation*.

THIS IS THE CLASSIC GANG-OF-FOUR FRAMING FOR WHEN VISITOR IS THE RIGHT TOOL

Visitor trades one axis of flexibility for another. Adding a new *operation* (a new visitor) becomes cheap — write one class, touch nothing else. Adding a new *node type* becomes more expensive — every existing visitor now needs a new method, or it fails the moment it meets that type. For an AST, that tradeoff is usually the right one: this course will add plenty of new operations over these same four-ish node types (an evaluator in Chapter 4, a resolver, a compiler in Course 2's own bytecode chapters) — but the node types themselves change rarely.

Failing Loud vs. Failing Silent

One more property falls out of the Visitor pattern for free, without anyone writing a single extra line to get it. Build an `EvaluatorWithEnv` that correctly implements `visit_variable`, and an `AstPrinter` that — realistically — hasn't been updated yet:

VERIFIED DIRECTLY — THE MISSING VISIT METHOD FAILS IMMEDIATELY AND SPECIFICALLY, WITH NO FALLBACK CODE REQUIRED

`var_tree2.accept(EvaluatorWithEnv({'x': 10.0}))` correctly returns **15.0**. Calling `var_tree2.accept(AstPrinter())` — where `AstPrinter` has no `visit_variable` method — immediately raises `AttributeError: 'AstPrinter' object has no attribute 'visit_variable'`. Nobody wrote that error. It's just what happens when Python tries to look up a method that was never defined.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Nested tuples formalized into typed `Expr` subclasses with `accept()`	Chapter 2's own raw `('binary', 'PLUS', left, right)` tuples, now replaced with real node classes carrying the same shape
The `Evaluator` visitor built in this chapter	Chapter 4's own tree-walking evaluator, which extends exactly this class with statements, scope, and control flow
Visitor missing a method fails loudly with `AttributeError`	Chapter 9's own error-handling chapter, which distinguishes this kind of implementation bug from a genuine Wisp-program runtime error
Design Patterns' own Chapter 1 named Visitor as explicitly out of scope	This chapter fills that specific, honestly-flagged gap rather than falsely claiming a cross-reference that doesn't exist

Hands-On Exercises

EXERCISE 1

Add a new `Ternary` node type for a conditional expression (`cond ? then : else`), including its own `accept()` method. Implement matching `visit_ternary` methods on both `Evaluator` and `AstPrinter`, and verify a tree for `1 ? 2 : 3` evaluates to `2.0` and prints as `(ternary 1.0 2.0 3.0)`.

View solution

EXERCISE 2

Write a new visitor, `MaxDepth`, that computes the maximum nesting depth of a tree (a `Literal` alone has depth 1) without modifying `Literal`, `Grouping`, `Unary`, or `Binary` at all. Verify it against the `"2 + 3 * 4"` tree from this chapter, and against a deeper tree like `"((1 + 2) * (3 + 4))"`.

View solution

EXERCISE 3

Write an isinstance-chain function `contains_division(node)` that checks whether a tree contains a division operator — but deliberately omit the final `raise` for unhandled node types. Run it on a tree containing a `Variable` node and determine exactly what it returns. Explain, in terms of Python's own truthiness rules, why the result is wrong rather than an error — and how the equivalent Visitor-based version would have failed instead.

View solution

CHAPTER 3 QUICK REFERENCE

- **Typed AST nodes:** `Literal`, `Grouping`, `Unary`, `Binary` replace Chapter 2's raw tuples, each with an `accept(visitor)` method
- **Verified:** the typed tree for "2 + 3 * 4" is structurally identical to Chapter 2's tuple tree, just now a real, typed object
- **Verified:** a new node type updated in one isinstance-chain function but forgotten in another raised a real TypeError — a genuine, reproduced maintenance slip
- **Double dispatch:** `node.accept(visitor)` calls back `visitor.visit_X(node)` — no type-checking anywhere
- **Verified:** adding a third operation (NodeCounter) needed zero new `accept()` methods; the isinstance-chain equivalent needed a full new type check, bringing the total to 13 dispatch checks across 3 functions vs. 4 defined once, ever
- **Verified:** a visitor missing a method fails immediately with AttributeError — no fallback code required, unlike an isinstance chain's easy-to-omit final raise
- **Honest gap:** this course's own Design Patterns course names Visitor as out of scope — this chapter is the from-scratch introduction that fills it

- **Next chapter:** Tree-Walking Evaluation — extending this chapter's own `Evaluator` into a full interpreter for statements, not just expressions

CHAPTER 4 OF 10

Tree-Walking Evaluation: Expressions & Statements

Writing a Compiler/Interpreter: Fundamentals

Chapter 4 · Tree-Walking Evaluation: Expressions & Statements

Chapter 3's `Evaluator` could compute the value of a single expression — but a real Wisp program isn't one expression, it's a sequence of statements, some of which produce a value nobody reads and some of which exist purely to cause an effect, like printing something. This chapter turns that one-expression evaluator into a real interpreter: statement node classes with their own accept/visit dispatch, string and boolean literals the parser never handled before, comparison operators actually wired up to the evaluator, and two pieces of quiet, easy-to-get-wrong semantics — what counts as "true," and how a number should look when printed.

Statements Are Not Expressions

Every `Expr` node produces a value when evaluated. A **statement** doesn't — it's executed for its effect. Wisp gets two for now: an expression statement (evaluate something and throw the result away — this is what a bare `2 + 2;` is) and a print statement (evaluate something, then actually output it).

```
class Stmt: def accept(self, visitor): raise NotImplementedError @dataclass class
ExpressionStmt(Stmt): expression: Expr def accept(self, visitor): return
visitor.visit_expression_stmt(self) @dataclass class PrintStmt(Stmt): expression: Expr def
accept(self, visitor): return visitor.visit_print_stmt(self)
```

Same shape as every `Expr` node from Chapter 3 — a small dataclass, an `accept()` method, double dispatch. The parser grows a second grammar layer above expressions: a `statement()` function that checks for the `print` keyword first, and a top-level `parse_program()` that keeps calling `declaration()` until it runs out of tokens, building a flat list of statements — an actual *program*, not just one expression.

Literals the Parser Never Handled Before

Chapter 1's lexer already produced `STRING` tokens and recognized `true` / `false` / `nil` as keywords — but Chapter 2 and 3's `primary()` only ever consumed `NUMBER` tokens. Printing anything interesting needs strings and booleans too, so `primary()` grows four new branches for them, each producing an ordinary `Literal` node wrapping a real Python `str`, `bool`, or `None`.

VERIFIED DIRECTLY — A REAL 8-STATEMENT WISP PROGRAM PARSES AND RUNS END TO END

Running `print 2 + 3 * 4;`, `print "hello, " + "world";`, `print 5 < 3;`, `print !0;`, `print nil;`, `print 3.0;`, `print 3.5;`, and a bare `2 + 2;` through `parse_program()` then `Interpreter.interpret()` produces exactly **7** lines of output (the bare expression statement contributes none): `14`, `hello, world`, `false`, `false`, `nil`, `3`, `3.5`.

Truthiness: Not Python's

The unary `!` operator needs an answer to "is this value true-ish?" for *any* Wisp value — a number, a string, `nil`. The tempting shortcut is Python's own `bool()` / `not` — except Python treats `0`, `0.0`, and `""` as falsy, and Wisp deliberately doesn't.

```
def is_truthy(value): if value is None: return False if isinstance(value, bool): return value
return True # everything else -- including 0.0 and "" -- is truthy in Wisp
```

VERIFIED DIRECTLY — TRUSTING PYTHON'S OWN TRUTHINESS GIVES THE WRONG ANSWER FOR "!0"

`is_truthy(0.0)` correctly returns **True** in Wisp's own rules, even though Python's `bool(0.0)` is `False`. Evaluating `!0` with the correct `is_truthy()` check gives **false** — the right answer, since 0 is a truthy value in Wisp and negating a truthy value is false. A naive implementation using Python's own `not 0.0` directly would have silently returned **True** instead — confidently wrong, with no error anywhere to catch it.

ONLY NIL AND FALSE ARE FALSY — THAT'S THE ENTIRE RULE

It's a short list on purpose. Some languages treat empty strings, empty collections, or zero as falsy too (Python and JavaScript both do, in overlapping but not identical ways) — Wisp doesn't, matching the same two-value falsy set used by Lua and Ruby. The value of picking a short, memorizable rule is exactly what this section just demonstrated: it's easy to accidentally reach for the host language's own truthiness instead, and the bug that produces is silent, not a crash.

Comparisons and a Real Equality Rule

Chapter 2's grammar already had a full precedence ladder for `==`, `!=`, `<`, `<=`, `>`, `>=` — but Chapter 3's `Evaluator` only ever implemented the four arithmetic operators inside `visit_binary`. This chapter fills in the rest, and equality specifically needs its own helper rather than a bare Python `==`.

```
def _is_equal(self, left, right): if type(left) is not type(right): return False return left ==
right
```

VERIFIED DIRECTLY — THE TYPE CHECK IS DOING REAL WORK, NOT JUST DEFENSIVE BOILERPLATE

`print true == 1;` evaluates to **false** in Wisp. Without the explicit `type(left) is not type(right)` guard, a bare Python `left == right` would have said **True** — Python's own `bool` is a subclass of `int`, so `True == 1.0` is genuinely `True` at the Python level. Wisp treats booleans and numbers as different types with nothing in common, so the type check isn't defensive padding — it's the one line standing between "correct" and a real, silent bug inherited from the host language's own type system.

THE SAME TYPE CHECK CORRECTLY LETS EQUAL-LOOKING NUMBERS THROUGH

`print 3.0 == 3;` evaluates to **true** — both literals parse through the same `float(tok[1])` call in `primary()`, so `type(3.0) is type(3.0)` is trivially true, and the values themselves are equal. The type check isn't about literal spelling; it's about which Wisp value Wisp actually produced.

Runtime Errors Instead of Leaking Python's Own

Feeding the wrong types into an arithmetic or comparison operator shouldn't crash the interpreter with a Python traceback a Wisp programmer has no way to make sense of. Each operator branch in `visit_binary` checks its operand types first and raises a dedicated `WispRuntimeError` with a message about the actual Wisp operation, before ever letting Python's own operators run on mismatched types.

VERIFIED DIRECTLY — A TYPE MISMATCH IS CAUGHT CLEANLY, WITH THE RAW PYTHON ERROR VISIBLE FOR CONTRAST

Evaluating `"five" < 3;` raises `WispRuntimeError: operands of '<' must be numbers, got str and float`. The same comparison performed with Python's own unguarded `<` operator instead raises `TypeError: '<' not supported between instances of 'str' and 'float'` — technically accurate, but describing Python's own type system to someone who has never heard of Python and is just trying to run a Wisp script.

Printing a Number Without Its Python Accent

Every Wisp number is a Python `float` under the hood, even literals that look like whole numbers — `3.0`, not a separate integer type. Printed with Python's own `str()`, that's `"3.0"`. A language with no separate int type shouldn't visibly leak that implementation detail every time a whole number gets printed.

```
def stringify(value): if value is None: return "nil" if isinstance(value, bool): return "true" if value else "false" if isinstance(value, float): if value == int(value): return str(int(value)) # 3.0 -> "3", not "3.0" return str(value) return str(value)
```

VERIFIED DIRECTLY — WHOLE AND FRACTIONAL NUMBERS PRINT DIFFERENTLY, AND CORRECTLY

`stringify(3.0)` returns `"3"`; `stringify(3.5)` returns `"3.5"`. Both are confirmed in the 8-statement program run earlier in this chapter: `print 3.0;` and `print 3.5;` printed `3` and `3.5` respectively, not `3.0` and `3.5`.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`Stmt` classes, parallel to Chapter 3's own `Expr` classes, with the same `accept()` /double-dispatch shape	Chapter 5's own `var` declarations and Chapter 6's own `if` / `while` statements will be new `Stmt` subclasses following this exact pattern
The `Interpreter` class built here, extending Chapter 3's `Evaluator` concept with statement handling	Chapter 5 extends this same class again with an `Environment` for variable storage — the object doesn't get replaced, it keeps growing
`WispRuntimeError`, raised for type mismatches	Chapter 9's own error-handling chapter, which distinguishes this from a parse-time `SyntaxError` and gives it a real line number to report
Wisp's own truthiness rule (only nil/false are falsy)	Chapter 6's own `if` statements and `and` / `or` short-circuit logic will call `is_truthy()` directly, unchanged from this chapter

Hands-On Exercises

EXERCISE 1

Add a modulo operator (`%`) at the same precedence level as `*` and `/` in `factor()`, including lexer support and a type-checked `visit_binary` case that raises `WispRuntimeError` for non-number operands, matching the existing arithmetic operators. Verify `"10 % 3;"` evaluates to `1`.

View solution

EXERCISE 2

Evaluate `-"hello";` using this chapter's own interpreter. Determine whether unary minus's existing type check catches it as a clean `WispRuntimeError`, or whether it's actually a raw Python exception leaking through undetected — and explain exactly which line of `visit_unary` is responsible either way.

View solution

EXERCISE 3

This chapter showed `print 3.0 == 3;` evaluating to `true` while `print true == 1;` evaluates to `false`. Trace both through `_is_equal`'s own type check step by step, and explain precisely why one passes the `type(left) is not type(right)` guard and the other doesn't, given that both pairs of literals "look like" they could be considered equal.

View solution

CHAPTER 4 QUICK REFERENCE

- **Statements vs. expressions:** `Stmt` nodes execute for effect; `Expr` nodes evaluate to a value — `ExpressionStmt` discards the value, `PrintStmt` uses it
- **Verified:** an 8-statement program parsed and ran end to end, producing exactly 7 lines of output — the bare expression statement contributed none
- **Truthiness:** only `nil` and `false` are falsy in Wisp — **verified** that trusting Python's own truthiness would have silently miscomputed `!0`
- **Equality:** `_is_equal` checks `type(left) is not type(right)` first — **verified** this is the one line preventing Python's own `True == 1.0` quirk from leaking into Wisp
- **Runtime errors:** type-mismatched operators raise a clean `WispRuntimeError` instead of a raw Python `TypeError` — **verified** side by side
- `stringify()`**:** whole-number floats print without a trailing `.0` — **verified** `3.0` prints as `3`, `3.5` prints as `3.5`
- **Next chapter:** Variables, Scope & Environments — giving the interpreter somewhere to actually store values between statements

CHAPTER 5 OF 10

Variables, Scope & Environments

Writing a Compiler/Interpreter: Fundamentals

Chapter 5 · Variables, Scope & Environments

Every program so far in this course has been stateless — each statement ran with no memory of the ones before it. This chapter gives the interpreter somewhere to actually store values: an **Environment**. Declaring a variable, reading it back, reassigning it, and scoping it to a block all come down to one design question — how does an environment find a name that wasn't declared in its own immediate scope? Get that wrong, and the bug it produces is exactly the kind that looks fine until you nest two blocks.

Four New Node Types

Variables touch both halves of the AST built so far. Reading one is an *expression* — `x` needs a value the same way `2 + 3` does. Declaring and reassigning are *statements* and an *expression*, respectively — `var x = 5;` is a new kind of statement, while `x = 5` is a new kind of expression that happens to also cause an effect. Blocks are statements too — a way to group statements and, critically, introduce a new scope.

```
@dataclass class Variable(Expr): # reading a variable's value name: str def accept(self, visitor): return visitor.visit_variable(self) @dataclass class Assign(Expr): # x = value -- an expression, not a statement name: str value: Expr def accept(self, visitor): return visitor.visit_assign(self) @dataclass class VarStmt(Stmt): # var x = value; (initializer may be None) name: str initializer: Expr def accept(self, visitor): return visitor.visit_var_stmt(self) @dataclass class BlockStmt(Stmt): # { ...statements... } statements: list def accept(self, visitor): return visitor.visit_block_stmt(self)
```

Assignment is parsed at the very top of the expression grammar, above equality — `parse_expression()` now calls a new `assignment()` level first, which parses an ordinary expression, then checks whether it's followed by `=`. If the left side wasn't a `Variable` node, that's a parse error — `2 + 2 = 5;` is nonsense, and this is where it gets caught.

VERIFIED DIRECTLY — DECLARING AND READING VARIABLES WORKS END TO END

Running `var x = 10; print x; var y = "hi"; print y;` produces exactly `['10', 'hi']`.

Undefined Variables Fail Loudly

Reading a name that was never declared, and assigning to one, both need to raise a real, specific error — not a raw Python `KeyError` from a dict lookup, and not a silently-returned `nil` that would mask a typo as a legitimate empty value.

VERIFIED DIRECTLY — READING AND ASSIGNING TO AN UNDECLARED NAME BOTH FAIL CLEANLY

`print y;` (with no prior `var y`) raises `WispRuntimeError: undefined variable 'y'`. `y = 5;` — assignment, not declaration — raises the exact same error. Assignment does not implicitly create a variable.

THIS IS A DELIBERATE DESIGN CHOICE, NOT AN OBVIOUS DEFAULT

Some real languages let a bare assignment to an undeclared name silently create one — pre-`"use strict"` JavaScript famously does, and it's a well-known source of accidental global variables from a single missing `var`/`let`. Wisp requires `var` for every new binding on purpose, so a typo in an assignment (`totl = totl + 1;` instead of `total`) is a loud runtime error instead of a silently-created, always-`nil`-until-now variable that quietly produces wrong output three lines later.

The Naive Environment: A Snapshot, Not a Chain

Blocks need their own scope — a variable declared inside `{ ... }` shouldn't leak out. The obvious-looking first attempt: when entering a block, make a new environment by *copying* whatever the enclosing scope currently holds.

```
class NaiveEnvironment: def __init__(self, enclosing=None): # BUG: copies the parent's CURRENT values into a new flat dict self.values = dict(enclosing.values) if enclosing else {} def assign(self, name, value): if name in self.values: self.values[name] = value return raise WispRuntimeError(f"undefined variable '{name}'")
```

This looks reasonable, and even runs without error. The bug only shows up when a block *reassigns* a variable that belongs to an outer scope, then that outer scope is read again afterward.

VERIFIED DIRECTLY — A REAL, REPRODUCED SCOPING BUG

Running `var x = 10; { x = x + 5; } print x;` using `NaiveEnvironment` prints **10** — the assignment inside the block never reached the outer `x` at all. The block's own environment is a *copy* of the global environment's values at the moment the block was entered; assigning inside the block updates that copy, which is discarded the instant the block ends.

The Fix: A Live Chain of Environments

The fix isn't to copy values at all — it's to keep a live reference to the enclosing environment, and walk up that reference chain whenever a name isn't found locally.

```
class Environment: def __init__(self, enclosing=None): self.values = {} self.enclosing = enclosing
# a LIVE reference, never copied def define(self, name, value): self.values[name] = value # always
defines in THIS environment def get(self, name): if name in self.values: return self.values[name]
if self.enclosing is not None: return self.enclosing.get(name) # walk UP the chain raise
WispRuntimeError(f"undefined variable '{name}'") def assign(self, name, value): if name in
self.values: self.values[name] = value return if self.enclosing is not None:
self.enclosing.assign(name, value) # walk UP, don't create locally return raise
WispRuntimeError(f"undefined variable '{name}'")
```

A block statement now executes its own statements against a brand-new `Environment(self.env)` — a child whose `enclosing` points straight at whatever environment was active before the block started — then restores the previous environment when the block ends. `get` and `assign` both fall through to `self.enclosing` when a name isn't local, recursing outward until either the name is found or the chain runs out.

> **VERIFIED DIRECTLY — THE EXACT SAME PROGRAM NOW GIVES THE CORRECT ANSWER**
>
> Running the identical `var x = 10; { x = x + 5; } print x;` using `Environment` instead of `NaiveEnvironment` prints **15**. Nothing about the AST or the parser changed — only how the environment looks up a name that isn't its own.

Shadowing

A block declaring its own variable with the same name as an outer one shouldn't reassign the outer variable — it should create a brand-new binding that temporarily hides it. `define()` always writes into *this* environment's own `values` dict, never walking the chain the way `assign()` does — that asymmetry is exactly what makes shadowing work.

> **VERIFIED DIRECTLY — A BLOCK'S OWN "VAR X" SHADOWS, AN ASSIGNMENT INSIDE IT WOULDN'T HAVE**
>
> Running `var x = "outer"; { var x = "inner"; print x; } print x;` produces `['inner', 'outer']`. Had the block instead written `x = "inner";` (assignment, no `var`), it would have walked the chain and overwritten the *outer* `x` instead of creating a new one — the same distinction the last section's bug hinged on.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`Environment`'s live `enclosing` chain	Chapter 7's own closures depend on exactly this mechanism — a function capturing "its own defining environment" means capturing a live reference to one of these chains, not a snapshot
A block gets a fresh child `Environment`, restored afterward	Chapter 6's own `if`/`while` bodies reuse this identical block-execution mechanism — a loop body is just a block executed repeatedly
`define()` never walks the chain; `assign()`/`get()` always do	This one-line asymmetry is the entire mechanism behind both shadowing and the naive-environment bug this chapter demonstrated — worth re-reading if either ever looks surprising later
Assignment does not implicitly declare	Chapter 9's own error-handling chapter gives `WispRuntimeError` a real source line, making "undefined variable" errors as easy to locate as a parse error already is

Hands-On Exercises

EXERCISE 1

Run `var x = 1; var x = 2; print x;` — redeclaring `x` in the exact same scope — through this chapter's own interpreter. Determine what gets printed, and explain whether `define()`'s own implementation permits or blocks this, and why that's a genuinely different case from the shadowing example in this chapter (which redeclares across a block boundary, not within one scope).

View solution

EXERCISE 2

Write a 3-level-deep nested program: a global variable, a block declaring a second variable, and an innermost block declaring a third — with the innermost block reading all three and then reassigning the global. Verify every read resolves correctly, and trace exactly how many `enclosing` hops `get()` needs to reach the global from the innermost block.

View solution

EXERCISE 3

Run `var x = 1; { var x = 2; { x = 99; } print x; } print x;` under both `NaiveEnvironment` and `Environment`. The two disagree on one of the two printed values but agree on the other — determine which value differs, what each implementation actually prints for it, and explain precisely why the naive version's bug applies to the middle block's `x` but not the outer one in this particular program.

View solution

CHAPTER 5 QUICK REFERENCE

- **Four new nodes:** `Variable` / `Assign` (expressions), `VarStmt` / `BlockStmt` (statements)
- **Verified:** declaring and reading variables works end to end — `var x=10; print x;` → `10`
- **Verified:** both undefined reads and undefined assignments raise `WispRuntimeError` — assignment never implicitly declares
- **Verified — the core bug:** a snapshot-copy `NaiveEnvironment` makes a block's assignment to an outer variable silently vanish (printed 10, not 15)
- **Verified — the fix:** a live `enclosing` chain, walked by `get()` / `assign()`, makes the identical program correctly print 15
- **Shadowing:** `define()` always writes locally; `get()` / `assign()` always walk the chain — that asymmetry is the whole mechanism
- **Next chapter:** Control Flow — `if` / `while` / `for`, built on this chapter's own block-execution machinery

CHAPTER 6 OF 10

Control Flow: Conditionals & Loops

Writing a Compiler/Interpreter: Fundamentals

Chapter 6 · Control Flow: Conditionals & Loops

Every program run through this interpreter so far has executed every statement exactly once, top to bottom. This chapter gives Wisp the ability to skip statements (`if` / `else`) and repeat them (`while` , `for`) — and along the way, a genuinely distinct kind of expression node, `Logical` , whose entire reason to exist is that it must sometimes decline to evaluate its own right-hand side at all. That turns out to matter for more than just `and` / `or` — this chapter closes with a real bug, found while writing it, in code this course has been running since Chapter 4.

if/else: A Straightforward New Statement

```
@dataclass class IfStmt(Stmt): condition: Expr then_branch: Stmt else_branch: Stmt # or None def accept(self, visitor): return visitor.visit_if_stmt(self) def visit_if_stmt(self, stmt): if is_truthy(stmt.condition.accept(self)): stmt.then_branch.accept(self) elif stmt.else_branch is not None: stmt.else_branch.accept(self)
```

Nothing surprising here — the condition is evaluated once, and Chapter 4's own `is_truthy()` decides which branch (if either) actually runs. The interesting part of `if` isn't the statement itself; it's that **the branch that isn't taken is never evaluated at all** — `stmt.then_branch.accept(self)` is a real Python call that either happens or doesn't, not a value that gets computed either way and discarded. That's the same idea this chapter's real subject, `Logical` , needs to get right at the expression level.

VERIFIED DIRECTLY

`if (5 > 3) { print "yes"; } else { print "no"; }` prints `yes` — the else branch's own `print "no";` never executes.

while: The Same Idea, Repeated

```
def visit_while_stmt(self, stmt): while is_truthy(stmt.condition.accept(self)): stmt.body.accept(self)
```

The condition is re-evaluated fresh before every single iteration — not cached, not assumed. This is what makes a Wisp `while` loop able to see its own body's side effects: the body reassigns a variable via `Assign` (Chapter 5), and the very next condition check reads that new value straight out of the `Environment`.

> **VERIFIED DIRECTLY — SUMMING 1 THROUGH 5 WITH A WHILE LOOP**
>
> `var i = 1; var sum = 0; while (i <= 5) { sum = sum + i; i = i + 1; } print sum;` prints **15**.

for: Desugared Into while, Zero New Node Types

A C-style `for (initializer; condition; increment) body` doesn't need its own AST node or its own interpreter method at all. The parser can build it entirely out of pieces this chapter and Chapter 5 already have: an optional `var` declaration (or a bare expression statement) before a `WhileStmt`, whose own body is the original body followed by the increment, all wrapped in a `BlockStmt` so the loop variable stays scoped to the loop.

```
def for_statement(self): self.expect('(') initializer = None if self.match(';') else (
self.var_declaration() if self.match('VAR') else self.expression_statement()) condition = None if
self.peek()[0] == ';' else self.parse_expression() self.expect(';') increment = None if
self.peek()[0] == ')' else self.parse_expression() self.expect(')') body = self.statement() if
increment is not None: body = BlockStmt([body, ExpressionStmt(increment)]) body =
WhileStmt(condition if condition is not None else Literal(True), body) if initializer is not None:
body = BlockStmt([initializer, body]) return body # the parser hands the interpreter a
BlockStmt/WhileStmt -- nothing "for"-shaped at all
```

By the time this reaches the interpreter, it isn't a `for` loop anymore — it's a block containing a declaration and a `while` loop. `Interpreter` needs no new method whatsoever to run it; it was already capable of running exactly this shape.

> **VERIFIED DIRECTLY — THE DESUGARED FOR LOOP IS BYTE-FOR-BYTE IDENTICAL IN OUTPUT TO THE EQUIVALENT HAND-WRITTEN WHILE LOOP**
>
> `for (var i = 0; i < 5; i = i + 1) { print i; }` produces `['0', '1', '2', '3', '4']`. A hand-written `{ var i = 0; while (i < 5) { print i; i = i + 1; } }` — written out exactly the way the desugaring above constructs it — produces the exact same list. There is no test that could distinguish the two at runtime, because after parsing, they aren't two different things.

Logical: A New Node, Because Binary Can't Do This

`and` and `or` look like they belong in `Binary` alongside `+` and `<` — but `Binary`'s own `visit_binary` always evaluates both `node.left.accept(self)` and `node.right.accept(self)` before deciding anything.

That's correct for arithmetic (you need both operands to add them) and wrong for logical operators, which are defined specifically so that the second operand is sometimes never even examined.

```
def visit_logical(self, node): left = node.left.accept(self) if node.operator == 'or': if
is_truthy(left): return left # right side NEVER evaluated return node.right.accept(self) else: #
'and' if not is_truthy(left): return left # right side NEVER evaluated return
node.right.accept(self)
```

Notice what these branches return: not always `true` or `false`, but one of the two *operand values themselves*. That's deliberate — it's what makes a common default-value idiom possible.

VERIFIED DIRECTLY — AND/OR RETURN AN OPERAND, NOT A COERCED BOOLEAN

`print nil or "default";` prints **default** — the string itself, not `true`. `print "hi" or 2;` prints **hi** — the left side won, because a non-empty string is truthy in Wisp, so the right side was never touched.

THIS IS EXACTLY THE SAME EVALUATION-ORDER QUESTION CHAPTER 4'S TRUTHINESS RULE ALREADY ANSWERED ONCE

`print 0 and "reached";` prints **reached**, not `0` — which can look wrong at first if you're used to a language where `0` is falsy. It isn't a new rule; it's Chapter 4's own truthiness table applied here: `0` is truthy in Wisp, so `and`'s left side doesn't stop evaluation, and the right side runs and wins.

Proving the Short-Circuit, Not Just Asserting It

"Short-circuit" is a claim about what *doesn't* run — which means the way to actually verify it is to put something that would visibly fail on the side that's supposed to be skipped, and confirm it never fails.

VERIFIED DIRECTLY — A NAIVE, EAGER VERSION OF LOGICAL RELIABLY CRASHES; THE REAL ONE DOESN'T

`var flag = false; print flag and (1 / 0);`, run through this chapter's own short-circuiting `visit_logical`, prints **false** with no error — `(1 / 0)` is never evaluated, because `flag` already determined the answer. The same program, run through a deliberately naive version that evaluates `node.right.accept(self)` unconditionally before checking anything (the same shape `visit_binary` already uses), raises a real `ZeroDivisionError: division by zero` instead. Swapping `flag` to `true` and the operator to `or` reproduces the identical contrast on the other branch: the correct version prints `true` cleanly, the naive version crashes the same way.

An Aside: The Same Mistake, Found Living in Chapter 4's Own Code

Writing the naive comparison above meant deliberately building code that evaluates something it doesn't need. That's worth checking for elsewhere in this course — and it turns up in a place with no relation to `and` / `or` at all: the arithmetic dispatch shown since Chapter 4.

```
# The shape used in Chapters 4-5's own code samples: return {'-': left - right, '*': left * right, '/': left / right}[op]
```

VERIFIED DIRECTLY — THIS LINE EVALUATES ALL THREE OPERATIONS, EVEN WHEN ONLY ONE WAS REQUESTED

Python builds a dict literal by evaluating *every value* before the dict exists at all — `left / right` gets computed even when `op` is `'-'`. Running `print 5 - 0;` through this exact dict-based dispatch raises `ZeroDivisionError: division by zero` — on a subtraction, because building the dict silently attempted the unrelated division first. Rewriting it as a plain `if`/`elif` chain (checking `op` before computing anything) fixes it: `print 5 - 0;` then correctly prints **5**. This is the same underlying mistake as an eager `Logical` — evaluating something "just in case" instead of only when it's actually needed — just found in a spot with no logical operator anywhere nearby. The code samples from this chapter onward use the `if`/`elif` form.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`IfStmt`/`WhileStmt` execute a branch by calling `.accept()` conditionally, never unconditionally	Chapter 7's own function calls will use this same "only run what's actually reached" discipline for early `return` statements
`for` desugars into existing nodes with zero new interpreter methods	A general technique worth remembering for Course 2's own bytecode compiler — fewer distinct node types to compile means fewer opcodes to design
`Logical` is a genuinely separate node type from `Binary`, not a special-cased operator inside it	Chapter 3's own Visitor pattern chapter predicted exactly this — a new node type needs a new `visit_` method on every existing visitor, which is precisely what happened here
The eager-dict bug, found and fixed in this chapter	A genuine, previously-unflagged bug in the arithmetic dispatch code shown in Chapters 4 and 5 — worth knowing about if you built along with those chapters directly

Hands-On Exercises

EXERCISE 1

C-style `for` loops allow omitting the initializer, the condition, or the increment (only the two semicolons are mandatory). Run `var i = 0; for (; i < 3; i = i + 1) { print i; }` (no initializer clause) and `for (var j = 0; j < 3;) { print j; j = j + 1; }` (no increment clause) through this chapter's own `for_statement()`. Verify both still produce correct output, and trace through the desugaring code to explain exactly which `if` branch each omitted clause takes.

View solution

EXERCISE 2

Run `var i = 10; while (i < 5) { print 1 / 0; } print "after";` — a `while` loop whose condition is false from the very first check. Verify the program completes without error despite its own body containing a guaranteed division by zero, and explain this in terms of the same "never evaluated" principle this chapter used for `if`'s untaken branch and `Logical`'s skipped operand.

View solution

EXERCISE 3

Run `print nil or false or 0 or "found it";` — a chain of three `or`s. Determine exactly which value gets printed, and explain why, tracing through `visit_logical` and Chapter 4's own `is_truthy()` rule for each operand in order. Is the result the one you'd expect coming from a language where `0` is falsy?

View solution

CHAPTER 6 QUICK REFERENCE

- **if/else, while:** new `Stmt` nodes; a branch/body only runs via a conditional `.accept()` call, never unconditionally
- **for:** desugared entirely into `BlockStmt` + `WhileStmt` at parse time — **verified** byte-for-byte identical output to the hand-written equivalent
- `Logical`**:** a genuinely new node type, distinct from `Binary`, because it must sometimes skip evaluating its own right side
- **Verified:** short-circuit `and`/`or` avoid a real `ZeroDivisionError` that a naive, eager version reliably triggers
- `and`/`or` **return an operand value**, not a coerced boolean — enables a `nil or "default"` idiom
- **Real bug found and fixed:** the dict-literal arithmetic dispatch shown since Chapter 4 evaluates all three operators before selecting one — **verified** crashing a plain subtraction; fixed with `if`/`elif`
- **Next chapter:** Functions & Closures — giving Wisp something for `if`/`while` bodies to actually call

CHAPTER 7 OF 10

Functions & Closures

Writing a Compiler/Interpreter: Fundamentals

Chapter 7 · Functions & Closures

Everything built so far runs top to bottom, once. This chapter adds the ability to package up a sequence of statements, give it a name and parameters, and run it — possibly many times, possibly with different arguments each time, possibly calling itself. Functions turn out to need surprisingly little new machinery: one new callable class, one new exception type for `return`, and a careful answer to a question that's easy to get subtly wrong — when a function is declared inside another function, exactly *which* environment does it remember?

Three New Node Types

```
@dataclass class FunctionStmt(Stmt): # fun name(params) { body } name: str params: list body: list
def accept(self, visitor): return visitor.visit_function_stmt(self) @dataclass class Call(Expr): #
callee(arguments) callee: Expr arguments: list def accept(self, visitor): return
visitor.visit_call(self) @dataclass class ReturnStmt(Stmt): # return expr; (expr may be omitted)
value: Expr def accept(self, visitor): return visitor.visit_return_stmt(self)
```

`Call` is parsed at a new precedence level between `unary` and `primary` — after parsing a primary expression, the parser checks for a following `(` in a loop, so `f()`, and even a hypothetical `f()()`, both parse correctly without any special-casing.

WispFunction: A Real Callable

A Wisp function value isn't the `FunctionStmt` node itself — it's a wrapper object pairing that declaration with the environment that was active *when the function was declared*. That second piece is the closure, and it's the part this chapter spends most of its time on.

```
class WispFunction: def __init__(self, declaration, closure): self.declaration = declaration
self.closure = closure # environment active AT DECLARATION time def call(self, interpreter,
arguments): env = Environment(self.closure) # a FRESH environment, every single call for param,
arg in zip(self.declaration.params, arguments): env.define(param, arg) try:
interpreter.execute_block(self.declaration.body, env) except ReturnException as r: return r.value
return None # fell off the end with no return -- implicit nil
```

`visit_function_stmt` builds one of these when the `fun` statement executes: `WispFunction(stmt, self.env)` — `self.env` at that exact moment, not the global environment, not a copy. That single argument is the entire closure mechanism.

> **VERIFIED DIRECTLY — DECLARING, CALLING, AND RETURNING ALL WORK END TO END**
>
> `fun greet(name) { print "hello, " + name; } greet("wisp");` prints **hello, wisp**. A function that falls off its own end with no `return` statement produces `nil`: `fun noReturn() { print "ran"; } var result = noReturn(); print result;` prints `['ran', 'nil']`.

return: An Exception, Not a Value

A `return` can appear anywhere inside a function body — nested three `if`s and a `while` loop deep, and it still needs to immediately stop everything and hand a value back to whoever called the function. A normal Python return from `visit_return_stmt` can't do that; it would only stop the current statement, not unwind out of however many nested blocks and loops are currently executing. Raising a Python exception can, because Python's own exception handling already unwinds through exactly that kind of nesting.

```
class ReturnException(Exception): def __init__(self, value): self.value = value def visit_return_stmt(self, stmt): value = stmt.value.accept(self) if stmt.value is not None else None raise ReturnException(value)
```

> **VERIFIED DIRECTLY — A RETURN STATEMENT CORRECTLY UNWINDS THROUGH NESTED CONTROL FLOW**
>
> `fun findFirstOver(limit) { var i = 0; while (true) { if (i > limit) { return i; } i = i + 1; } print "never reached"; } print findFirstOver(5);` prints **6**. The `return` sits two levels deep — inside an `if`, inside a `while` — and the `ReturnException` passes cleanly through both `visit_if_stmt` and `visit_while_stmt` (neither one catches it) all the way up to `WispFunction.call()`'s own `try`/`except`. The trailing `print "never reached";` is exactly that — never reached.

Recursion works the same way it does in Python, because a Wisp function call really is a Python function call. `fun factorial(n) { if (n <= 1) { return 1; } return n * factorial(n - 1); }` computing `factorial(5)` makes a real, recursive Python call to `WispFunction.call()` five levels deep.

> **VERIFIED DIRECTLY — RECURSION WORKS, COMPUTING GENUINELY CORRECT VALUES**
>
> `print factorial(5); print factorial(10);` prints `['120', '3628800']`.

Closures Capture a Live Environment, Not a Copy

"Closure" means the function remembers the environment it was declared in, and keeps seeing that environment's own live updates — not a frozen snapshot of what it contained at declaration time.

```
fun makeAdder(n) { fun adder(x) { return x + n; // 'n' isn't adder's own parameter -- it's captured } return adder; } var addFive = makeAdder(5); var addTen = makeAdder(10);
```

Each call to `makeAdder` gets its own fresh call environment (that's Chapter 5's own `Environment`, unchanged). `adder` is declared inside that call, so its closure is *that specific call's* environment — `self.env` at the moment `fun adder(x) {...}` executes, which is genuinely different for the `makeAdder(5)` call than for the `makeAdder(10)` call.

VERIFIED DIRECTLY — TWO CLOSURES FROM TWO SEPARATE CALLS STAY INDEPENDENT

`print addFive(3); print addTen(3);` prints `['8', '13']` — each `adder` correctly remembers its own `n`, even though both were declared from the exact same source line inside `makeAdder`.

The Over-Capture Bug: One Environment, Reused

The bug isn't in what gets captured — it's in *when* the call environment gets created. `WispFunction.call()` above creates `Environment(self.closure)` fresh, every single call. A tempting, subtly wrong "optimization" is to create it once and reuse it:

```
class OverCapturingWispFunction: def __init__(self, declaration, closure): self.declaration = declaration self.closure = closure self._shared_call_env = None # BUG: created once def call(self, interpreter, arguments): if self._shared_call_env is None: self._shared_call_env = Environment(self.closure) env = self._shared_call_env # BUG: reused every call for param, arg in zip(self.declaration.params, arguments): env.define(param, arg) ...
```

This looks harmless in isolation — every individual call to a single `WispFunction` still runs correctly. The bug only appears when a function that itself *declares and returns closures* — like `makeAdder` — is called more than once, because now every one of those returned closures shares the exact same underlying call environment object.

VERIFIED DIRECTLY — A REAL, REPRODUCED BUG: TWO "INDEPENDENT" CLOSURES BLEEDING INTO EACH OTHER

Running the exact same `makeAdder` program from above, but with `makeAdder` itself built using `OverCapturingWispFunction`, `print addFive(3); print addTen(3);` prints `['13', '13']` — not `['8', '13']`. Calling `makeAdder(10)` second reused and overwrote the same shared environment's `n` from `5` to `10`. `addFive`'s own closure points at that identical shared object, so by the time `addFive(3)` actually runs, the `n` it sees has already been silently changed out from under it by an entirely separate, later call to `makeAdder`.

THIS IS WHY "FRESH ENVIRONMENT PER CALL" ISN'T OPTIONAL, EVEN THOUGH IT LOOKS LIKE WASTED ALLOCATION

Every call to *any* function gets its own new `Environment` in the correct implementation — including calls that don't return a closure at all, where reusing one might genuinely seem safe. The moment a function's own body can declare and return something that keeps a live reference to that call's environment, reuse becomes a correctness bug, not just a performance shortcut. There's no cheap way to tell in advance which functions will do that, so every call gets a fresh one, unconditionally.

The same live-capture mechanism also means a closure sees mutations made to a variable *after* the closure was declared, as long as they happen before the closure is called — closures over a mutable local, like a counter, work for exactly this reason.

VERIFIED DIRECTLY — A CLOSURE OVER A REPEATEDLY-MUTATED LOCAL VARIABLE

`fun makeCounter() { var count = 0; fun increment() { count = count + 1; print count; } return increment; } var counter = makeCounter(); counter(); counter(); counter();` prints `['1', '2', '3']` — the same `count` binding, correctly persisting and incrementing across three separate calls to the returned closure.

The Call Stack Is Python's Own

This interpreter never built anything resembling a call stack — no explicit list of active calls, no frame objects. Every Wisp function call is a real, nested Python call (`accept` → `visit_call` → `WispFunction.call` → `execute_block` → `accept` → ...), so Wisp's own call depth is bounded by whatever bounds Python's.

VERIFIED DIRECTLY — A REAL, MEASURED RECURSION LIMIT, NOT A THEORETICAL ONE

A recursive `countDown(n)` function, run at Python's actual default recursion limit (1000), correctly completes up to **163** levels of Wisp recursion before raising a genuine Python `RecursionError` — *not* 1000, because each single Wisp call costs several real Python stack frames (`accept`, `visit_call`, `call`, `execute_block`, another `accept`...) rather than one. Raising Python's own limit to 3000 pushed the measured working depth to **497**. There is no Wisp-specific "stack overflow" check anywhere in this interpreter — the error is entirely Python's own, borrowed for free.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`WispFunction.closure`, captured as `self.env` at declaration time	Chapter 5's own `Environment.enclosing` chain — a closure is nothing more than one more link that chain remembers, stored on a function value instead of discarded when a block ends
`ReturnException` unwinding through nested `if`/`while`	Chapter 6's own control-flow statements needed no changes at all to support this — they simply never catch an exception that isn't theirs

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The over-capture bug: one call environment reused across calls	Chapter 5's own `NaiveEnvironment` bug (a copy instead of a live reference) — a different mistake with the same root cause: sharing state between things that were supposed to be independent
Recursion depth bounded by Python's own stack	Course 2's own bytecode VM (Chapter 2 onward) explicitly manages its own call-frame stack instead of borrowing the host language's — a direct architectural consequence of not tree-walking anymore

Hands-On Exercises

EXERCISE 1

Write a recursive Fibonacci function, `fun fib(n) { ... }`, using the same base-case-then-recursive-case shape as this chapter's own `factorial`. Verify `fib(10)` against the well-known Fibonacci sequence, and count how many total recursive calls to `fib` happen for `fib(10)` — is it closer to 10, or dramatically more?

View solution

EXERCISE 2

Run `var x = "before"; fun show() { print x; } x = "after"; show();`. Determine whether it prints `before` or `after`, and use it to explain precisely what "closures capture a live environment, not a snapshot of values" means for a case this chapter didn't directly cover — a variable reassigned *after* the function was declared but *before* it was ever called.

View solution

EXERCISE 3

This chapter measured a maximum working recursion depth of 163 at Python's default limit and 497 at a limit of 3000. Using `sys.setrecursionlimit()` and this chapter's own `countDown` function, find the maximum working depth at a limit of 2000, and use the two known data points (1000→163, 3000→497) to predict it before checking. Is the relationship between Python's limit and Wisp's own usable depth linear?

View solution

CHAPTER 7 QUICK REFERENCE

- **Three new nodes:** `FunctionStmt`, `Call`, `ReturnStmt`
- `WispFunction`: pairs a declaration with a closure environment; `call()` creates a fresh `Environment(closure)` every invocation

- `return`: implemented as a raised `ReturnException`, caught only in `WispFunction.call()` — **verified** unwinding cleanly through nested `if`/`while`
- **Verified:** recursion works correctly — `factorial(10)` → 3628800
- **Verified — closures:** two closures from two separate calls to the same outer function stay independent (8 and 13, not both 13)
- **Verified — the over-capture bug:** reusing one call environment across calls instead of creating a fresh one makes independent closures bleed into each other
- **Verified:** Wisp's own recursion limit is Python's, inherited for free — measured at 163 levels (default) and 497 levels (limit raised to 3000)
- **Next chapter:** Classes & Object-Oriented Features — building on this exact closure mechanism for how `this` gets bound inside a method

CHAPTER 8 OF 10

Classes & Object-Oriented Features

Writing a Compiler/Interpreter: Fundamentals

Chapter 8 · Classes & Object-Oriented Features

A class in Wisp is, structurally, not far from what Chapter 7 already built: a callable value (calling it constructs an instance, the same way calling a function runs its body) that owns a dictionary of methods, each of which is itself a `WispFunction`. The genuinely new problem this chapter has to solve is `this` — a method needs to know which specific instance it's currently running against, and getting that binding wrong is easy to do in a way that looks correct until two instances exist at once.

Three New Pieces: Classes, Instances, Properties

```
class WispClass: def __init__(self, name, superclass, methods): self.name = name self.superclass = superclass # WispClass or None self.methods = methods # dict: name -> WispFunction def find_method(self, name): if name in self.methods: return self.methods[name] if self.superclass is not None: return self.superclass.find_method(name) # walk UP the chain return None def call(self, interpreter, arguments): # calling the CLASS constructs an instance instance = WispInstance(self) initializer = self.find_method("init") if initializer is not None: initializer.bind(instance).call(interpreter, arguments) return instance class WispInstance: def __init__(self, klass): self.klass = klass self.fields = {} # per-instance data, set via 'this.x = ...' def get(self, name): if name in self.fields: return self.fields[name] method = self.klass.find_method(name) if method is not None: return method.bind(self) # the key line -- covered below raise WispRuntimeError(f"undefined property '{name}'")
```

Fields aren't declared anywhere — `WispInstance.fields` starts empty, and a property is created the first time something is assigned to it via `this.x = ...` inside a method. `Get` (`instance.field`) and `Set` (`instance.field = value`) are two new expression nodes, parsed at the same precedence level as function calls — after a primary expression, the parser loops checking for either a following `(` (a call) or a following `.` (a property access), so `a.b.c()` parses correctly without any special grammar rule for chaining.

VERIFIED DIRECTLY — A CLASS, AN INSTANCE, A FIELD SET IN INIT, AND A METHOD READING IT

`class Greeter { init(name) { this.name = name; } greet() { return "hello, " + this.name; } } var g = Greeter("wisp"); print g.greet();` prints **hello, wisp**.

this: Bound Fresh, Every Single Access

`this` isn't a keyword with special evaluation rules — it's parsed straight into `Variable("this")` in `primary()`, and resolved through the ordinary `Environment` chain like any other name. The entire mechanism is `WispFunction.bind()`, called every time a method is looked up on an instance via `WispInstance.get()`:

```
def bind(self, instance): env = Environment(self.closure) # a FRESH environment env.define("this", instance) return WispFunction(self.declaration, env, self.is_initializer) # a FRESH function
```

Every property access that resolves to a method produces a brand-new `WispFunction` object, wrapping a brand-new environment, whose `enclosing` points back at the method's own original closure (the environment active when the class itself was declared) — with exactly one new binding, `this`, layered on top for this specific instance.

VERIFIED DIRECTLY — TWO INSTANCES OF THE SAME CLASS KEEP INDEPENDENT STATE

`class Counter { init(startAt) { this.count = startAt; } increment() { this.count = this.count + 1; return this.count; } } var a = Counter(0); var b = Counter(100); var incA = a.increment; var incB = b.increment; print incA(); print incB();` prints `['1', '101']` — `incA` and `incB` are two genuinely different `WispFunction` objects, each with its own bound `this`, even though both came from the exact same `increment` method declaration.

The Over-Mutation Bug: Binding in Place Instead of Creating Fresh

`bind()` looks like it's doing more allocation than necessary — a new environment and a new function object, every single time a method is touched. The tempting shortcut: define `this` directly into the method's own existing closure, and return the same function object unchanged.

```
def bind(self, instance): # BUGGY self.closure.define("this", instance) # mutates the SHARED closure in place return self # returns the SAME function object every time
```

`self.closure` here is the environment that was active when the class itself was declared — the *same* environment every instance's `increment` method shares, since there's only one `increment` declaration for the whole `Counter` class. Mutating it in place means every instance's own "binding" is actually the exact same shared slot, overwritten by whichever instance accessed the method most recently.

VERIFIED DIRECTLY — A REAL, REPRODUCED BUG: TWO "INDEPENDENT" STORED METHODS COLLAPSING INTO ONE

Running the identical `Counter` program above with this buggy `bind()`, `print incA(); print incB();` prints `['101', '102']` — not `['1', '101']`. `incB = b.increment;` ran *after* `incA = a.increment;`, and its own call to `bind()` overwrote the shared closure's `this` from `a` to `b`. By the time `incA()` actually executes, `incA` and

`incB` are — despite having come from two different instances — literally the same Python object, and both now operate on `b`'s own `count`. `a`'s data is never touched at all; the entire program silently forgets it exists.

THIS IS THE SAME SHAPE AS CHAPTER 7'S OWN OVER-CAPTURE BUG, ONE LAYER UP

Chapter 7 showed reusing one *call* environment across multiple calls to the same function breaking independence between calls. This is the identical mistake at the method-binding layer: reusing one *bound-method* object across multiple instances breaks independence between instances. Both bugs come from the same root cause — treating something that needs to be created fresh, per logical "instance" of a concept, as safe to create once and share.

Single Inheritance: A Chain of find_method() Calls

`class Dog < Animal { ... }` stores `Animal`'s own `WispClass` as `Dog.superclass`. Nothing about method lookup changes structurally — `find_method()` just doesn't stop at `self.methods` if the name isn't there; it asks the superclass to look too, recursively, for as many levels as the inheritance chain goes.

VERIFIED DIRECTLY — AN OVERRIDDEN METHOD WINS LOCALLY; AN UNOVERRIDDEN ONE IS FOUND UP THE CHAIN

`class Animal { init(name) { this.name = name; } speak() { return this.name + " makes a sound."; } } class Dog < Animal { speak() { return this.name + " barks."; } } class Cat < Animal { }` — `Dog` overrides `speak`; `Cat` doesn't. `var d = Dog("Rex"); var c = Cat("Whiskers"); print d.speak(); print c.speak();` prints `['Rex barks.', 'Whiskers makes a sound.']`. `Cat` also never defines its own `init` — `Cat("Whiskers")` works anyway, because `WispClass.call()`'s own `find_method("init")` walks the exact same chain and finds `Animal`'s.

init Always Returns the Instance

A constructor's job is to set up an instance, not to compute a return value — so `init` is treated specially: even if its own body contains an explicit `return` statement, calling a class always yields the instance, never whatever `init` tried to return.

VERIFIED DIRECTLY — AN EXPLICIT RETURN INSIDE INIT IS IGNORED

`class Weird { init(x) { this.x = x; return "ignored"; } } var w = Weird(42); print w.x;` prints **42** — `Weird(42)` itself evaluates to the instance, not the string `"ignored"`, because `WispFunction.call()` checks `self.is_initializer` and substitutes `self.closure.get("this")` for whatever the `ReturnException` actually carried.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`bind()` creates a fresh `Environment(self.closure)`, wrapping the method's own closure	Chapter 7's own closure mechanism, reused directly — a bound method is a closure, just one whose captured environment happens to contain `this`
The over-mutation bug: one shared bound-method object across instances	Chapter 7's own over-capture bug (one shared call environment across function calls) and Chapter 5's `NaiveEnvironment` (a copy instead of a live reference) — three chapters, the same underlying mistake at three different layers
`find_method()`'s recursive walk up `superclass`	Chapter 5's own `Environment.get()` walking up `enclosing` — structurally the identical pattern, applied to class hierarchies instead of lexical scopes
No `super.method()` syntax in this chapter	An honest scope boundary — this chapter covers single inheritance and method resolution as outlined, not explicit superclass-method calls from inside an override; a real Wisp implementation would add a `Super` expression node following the same closure-binding pattern as `this`

Hands-On Exercises

EXERCISE 1

Call `g.greet(1)` against a class whose `greet(a, b)` method expects two parameters. Verify this raises the same kind of arity error Chapter 7 established for plain function calls, and trace through `visit_call` to confirm it applies identically to a bound method as it does to an ordinary function — is there any special-casing for methods anywhere in the arity check?

View solution

EXERCISE 2

This chapter showed `init` ignoring an explicit `return "ignored";` when the constructor completes normally. Trace through `WispFunction.call()`'s own `is_initializer` handling for *both* code paths — the `except ReturnException` branch and the "fell off the end with no return at all" branch — and explain why both need their own explicit substitution of `self.closure.get("this")` rather than one shared check being sufficient.

View solution

EXERCISE 3

Build a three-level inheritance chain, `class A { whoAmI() { return "A"; } } class B < A {} class C < B {}`, and verify `C().whoAmI()` correctly finds `A`'s method two levels up. Then verify what happens calling a method that exists nowhere in the entire chain, and explain how many recursive `find_method()` calls each scenario needs.

View solution

CHAPTER 8 QUICK REFERENCE

- **New pieces:** `ClassStmt`, `Get`/`Set` expressions, `WispClass` (callable → constructs), `WispInstance` (a `fields` dict)
- **Verified:** a class, instance, `init`-set field, and method read all work end to end
- `this`: just `Variable("this")`, bound fresh via `WispFunction.bind()` on every single property access — a new environment, a new function object, every time
- **Verified — the over-mutation bug:** binding `this` by mutating a method's shared closure in place, instead of creating a fresh copy, collapses two stored, independent bound methods into one (['1','101'] becoming ['101','102'])
- **Inheritance:** `find_method()` walks the `superclass` chain recursively — **verified** override, inheritance, and inherited `init` all working correctly
- `init`: always returns the instance, regardless of any explicit `return` inside it — **verified**
- **Next chapter:** Error Handling & Runtime Diagnostics — giving every `WispRuntimeError` raised since Chapter 4 a real source line to report

CHAPTER 9 OF 10

Error Handling & Runtime Diagnostics

Writing a Compiler/Interpreter: Fundamentals

Chapter 9 · Error Handling & Runtime Diagnostics

Every chapter since Chapter 1 has raised an error of one kind or another — an unterminated string, a missing parenthesis, an undefined variable, a type mismatch — but each used whatever exception felt convenient at the time, with no shared shape and no line number attached to most of them. This chapter fixes that: one small error hierarchy, real line numbers threaded through the lexer and parser and onto every AST node that needs one, and a genuine Wisp-level stack trace for runtime errors — built by hand, since Chapter 7 already established that this interpreter has no call stack of its own to read one from.

Three Categories, One Shared Shape

```
class WispError(Exception): def __init__(self, message, line=None): super().__init__(message)
self.message = message self.line = line def report(self): where = f"[line {self.line}] " if
self.line is not None else "" return where + type(self).__name__ + ": " + self.message class
LexError(WispError): pass # Chapter 1: unterminated strings, bad characters class
ParseError(WispError): pass # Chapter 2: unexpected tokens, missing punctuation class
WispRuntimeError(WispError): pass # Chapter 4 onward: type errors, undefined names
```

Nothing about *when* each of these three fires has changed since the chapters that introduced them — the lexer still raises during tokenizing, the parser still raises during parsing, the interpreter still raises during evaluation. What's new is that all three now share one `report()` method and one consistent `[line N] ErrorType: message` format, instead of three unrelated ad-hoc exception shapes accumulated one chapter at a time.

VERIFIED DIRECTLY — ONE CONSISTENT REPORT FORMAT ACROSS ALL THREE CATEGORIES

`var x = "never closed;` (no closing quote) reports `[line 1] LexError: unterminated string`. `if (5 > 3 { print "no paren"; }` (a missing `)`) reports `[line 1] ParseError: expected ), got { ('{')`. `print 5 + "text";` reports `[line 1] WispRuntimeError: operands of '+' must be two numbers or two strings, got float and str`.

Threading Line Numbers Through the Whole Pipeline

A line number has to originate somewhere real — the lexer's own scan position — and then survive being carried through every stage after that. Tokens grow a third field: `(kind, lexeme, line)`, computed once per token as `1 + source.count('\n', 0, match_start)`. The parser then stamps that line onto every AST node it builds — `Binary`, `Unary`, `Call`, `Variable`, `Get`, `Set`, and `FunctionStmt` each grow a `.line` field, taken from whichever token triggered that node's construction (the operator for `Binary`, the opening `(` for `Call`, the identifier itself for `Variable`).

Every place in the interpreter that already raised a `WispRuntimeError` — Chapter 4's type checks, Chapter 5's undefined-variable checks, Chapter 7's arity checks, Chapter 8's undefined-property checks — now passes `node.line` through, since the node being evaluated when the error is detected is always sitting right there in the same method.

Reachability: The Real Difference Between the Three Categories

The categories aren't just cosmetic labels — they genuinely behave differently, and the difference is about *when* the whole program has to be examined versus only the parts that actually run. Lexing and parsing both process the entire source file before a single statement executes. Interpretation only ever reaches the statements the program's own control flow actually visits.

VERIFIED DIRECTLY — A PARSE ERROR INSIDE A BRANCH THAT NEVER RUNS STILL FAILS IMMEDIATELY

`if (false) { print 5 } print "fine";` — a missing semicolon inside an `if (false)` branch that can never execute — still reports `[line 1] ParseError: expected ;, got } ('}')`, and `"fine"` is never printed. The whole program has to parse successfully before the interpreter runs *any* of it, so the branch's own unreachability is irrelevant — the broken syntax is found regardless.

VERIFIED DIRECTLY — THE EXACT SAME UNREACHABLE BRANCH, WITH A RUNTIME ERROR INSTEAD, CAUSES NO FAILURE AT ALL

`if (false) { print 5 + "oops"; } print "fine";` — syntactically valid, but a genuine type error if that branch ever ran — completes successfully and prints only `['fine']`, no error whatsoever. `5 + "oops"` is never evaluated, because the `if`'s own condition is false, so `visit_binary` never runs against it and never gets the chance to detect the mismatch.

THIS IS ALSO TRUE OF LEXICAL ERRORS, NOT JUST PARSE ERRORS

`if (false) { var x = "unterminated; } print "fine";` raises the exact same `LexError` it would raise if that line were the very first thing in the file — tokenizing happens once, for the entire source text, before parsing even begins, so an unreachable branch's own broken string literal is found just as reliably as broken syntax is.

A Real Wisp Stack Trace

Chapter 7 established that Wisp's own call "stack" is just Python's — there's no explicit list of active calls anywhere. That's fine for making recursion work, but it means there's nothing to read a Wisp-level stack trace *from* when something goes wrong three function calls deep. This chapter adds exactly that: an explicit `call_stack` list on the interpreter, pushed and popped around every function call.

```
def call(self, interpreter, arguments, call_line=None): env = Environment(self.closure) for param, arg in zip(self.declaration.params, arguments): env.define(param, arg) interpreter.call_stack.append((self.declaration.name, call_line)) try: interpreter.execute_block(self.declaration.body, env) except ReturnException as r: return r.value except WispRuntimeError as e: if not hasattr(e, "wisp_stack"): e.wisp_stack = list(interpreter.call_stack) # snapshot BEFORE any frame pops raise finally: interpreter.call_stack.pop() return None
```

The snapshot has to happen in the *innermost* `call()` frame that sees the error — the first one to run its own `except WispRuntimeError` block — because that's the only point where every frame from the outermost call down to the one that actually failed is still present in `call_stack`. The `finally` block pops this frame regardless, but only *after* the `except` block above it has already run and taken its snapshot; the `if not hasattr(...)` guard stops an outer frame from overwriting that snapshot with its own, now-truncated view as the exception continues propagating upward.

VERIFIED DIRECTLY — A REAL, MULTI-FRAME TRACE THROUGH A THREE-LEVEL CALL CHAIN

`fun c(x) { return x + "oops"; } fun b(x) { return c(x); } fun a(x) { return b(x); } print a(5);` reports:

```
[line 3] WispRuntimeError: operands of '+' must be two numbers or two strings, got float and str
  at c() (called from line 6)
  at b() (called from line 9)
  at a() (called from line 11)
  at <script>
```

Three real, correctly-ordered frames — innermost first — each showing the line where *that specific call* happened, not where the function was declared.

A Real Bug, Found While Writing This Chapter's Own Verification

Testing the stack trace against a recursive function surfaced a genuine gap: `print 1 / 0;` doesn't raise a `WispRuntimeError` at all.

VERIFIED DIRECTLY — DIVISION BY ZERO LEAKS A RAW PYTHON EXCEPTION, UNCAUGHT SINCE CHAPTER 4

Running `print 1 / 0;` through the top-level driver raises a bare `ZeroDivisionError: division by zero` — Python's own exception, with no line number, no `[line N]` prefix, and no chance for this chapter's own error-reporting machinery to touch it at all, because `visit_binary`'s `/` case has never once checked its divisor. The fix follows the same pattern as every other operand check since Chapter 4: `if right == 0.0: raise WispRuntimeError("division by zero", node.line)`, checked before the division itself runs.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
One `WispError` hierarchy for lex/parse/runtime errors	Chapters 1, 2, and 4-8's own scattered, ad-hoc exceptions — unified here, not replaced; every earlier chapter's own error condition still fires at exactly the same point
Reachability: lex/parse errors are whole-program, runtime errors are execution-dependent	Chapter 6's own short-circuit evaluation and Exercise 2 (a while loop whose body never runs) — both are instances of the same general fact, that Wisp only evaluates what control flow actually visits
An explicit `call_stack`, pushed/popped around every `WispFunction.call()`	Chapter 7's own finding that Wisp's call depth is bounded by Python's real stack — this chapter adds a Wisp-level *view* of that stack without changing the underlying mechanism at all
The division-by-zero gap, found and fixed	Chapter 6's own eager-dict arithmetic bug — the second real, previously-unflagged bug this course has found by actually testing edge cases rather than assuming coverage

Hands-On Exercises

EXERCISE 1

Run `if (false) { var x = "unterminated; } print "fine";` — a lexical error (not a parse error) sitting inside a branch that can never execute. Determine whether the program prints `fine` or fails, and explain why this chapter's own "whole-program, before-any-execution" reasoning applies identically to lexing as it does to parsing, even though they're two separate stages.

View solution

EXERCISE 2

Write a recursive `countDown(n)` that calls itself until `n <= 0`, at which point it evaluates `n + "boom"` (a genuine type error). Call it with an initial argument of 3, and inspect the resulting stack trace. Explain why the function name `countDown` appears multiple times, why three of those entries share one "called from" line while a fourth doesn't, and what that difference reveals about where each specific call in the trace actually originated.

View solution

EXERCISE 3

Run three nested function calls (`a` calls `b` calls `c`, no errors anywhere) to completion, then inspect `interpreter.call_stack` immediately afterward. Determine whether any frames are left behind, and explain specifically which line of `WispFunction.call()` is responsible for the answer — would the same guarantee hold if that line were inside the `try` block instead of where it actually is?

View solution

CHAPTER 9 QUICK REFERENCE

- **One hierarchy:** `WispError` → `LexError` / `ParseError` / `WispRuntimeError`, each with `.line` and a shared `report()`
- **Line numbers threaded through:** tokens carry a line; the parser stamps it onto `Binary` / `Unary` / `Call` / `Variable` / `Get` / `Set` / `FunctionStmt`
- **Verified — reachability:** lex/parse errors fire regardless of whether the broken code would ever run; runtime errors only fire if the line actually executes
- `call_stack`: an explicit list, pushed/popped around every `WispFunction.call()` — **verified** producing a correct, multi-frame, correctly-ordered trace
- **Real bug found and fixed:** division by zero was leaking a raw Python `ZeroDivisionError` since Chapter 4, uncaught by any of this chapter's own reporting
- **Next chapter:** Capstone — assembling every chapter's own component into one complete, working Wisp interpreter

CHAPTER 10 OF 10

Capstone — A Complete Tree-Walking Interpreter for Wisp

Writing a Compiler/Interpreter: Fundamentals

Chapter 10 · Capstone: A Complete Tree-Walking Interpreter for Wisp

Nine chapters, each adding one working piece: a lexer, a parser, a tree of typed nodes, a tree-walking evaluator, an environment chain, control flow, functions and closures, classes and inheritance, and a real error-reporting layer with line numbers and stack traces. Nothing from any of those chapters gets replaced here — this capstone is one continuous, moderately complex Wisp program, run through the exact interpreter those nine chapters built, exercising closures, classes, and control flow together in a single working system rather than in nine separate, disconnected demonstrations.

The Program: A Task Queue, Built Without Arrays

Wisp never gained a native array or list type across this course — so the capstone's own data structure is a linked list, built entirely out of classes, the same way a real language without a builtin collection type would have to. This is deliberate: it's a genuine, useful demonstration of what classes alone can build, not a workaround apologized for.

```
class Task { init(description, priority) { this.description = description; this.priority = priority; } describe() { return this.description; } } class UrgentTask < Task { // Chapter 8: single inheritance, method override describe() { return "URGENT: " + this.description; } } class TaskNode { // a linked-list node, built from an ordinary class init(task, next) { this.task = task; this.next = next; } } class TaskQueue { init() { this.head = nil; this.count = 0; } add(task) { this.head = TaskNode(task, this.head); // prepend this.count = this.count + 1; } forEach(callback) { // Chapter 7: functions are ordinary values var node = this.head; while (node != nil) { // Chapter 6: control flow callback(node.task); node = node.next; } } }
```

`UrgentTask` overrides `describe()` exactly the way Chapter 8's own `Dog < Animal` example did. `TaskQueue.forEach` takes a *function* as its own argument — `callback` is called like any other value, because Chapter 7 established that functions in Wisp are ordinary, first-class values, not a special kind of name.

A Closure Factory, Reused Directly From Chapter 7

```
fun makeUrgencyChecker(threshold) { // the exact shape of Ch.7's own makeAdder fun isUrgent(task)
{ return task.priority >= threshold; // 'threshold' is CAPTURED, not a parameter } return
isUrgent; } fun countMatching(queue, checker) { var count = 0; fun tally(task) { if
(checker(task)) { // calling a function PASSED IN as a value count = count + 1; // mutating a
captured local -- Ch.5 + Ch.7's own mechanism } } queue.forEach(tally); return count; }
```

`tally` is a closure over two different things at once: `checker`, a parameter of `countMatching`, and `count`, a local variable it mutates on every matching task. It's also passed into `queue.forEach` as a plain value — three chapters' worth of machinery (Chapter 5's live environment chain, Chapter 7's closures, and Chapter 7's first-class functions) cooperating in four lines, with nothing here that any single earlier chapter didn't already establish on its own.

Running It

```
var queue = TaskQueue(); queue.add(Task("Water the plants", 2)); queue.add(UrgentTask("Submit tax
filing", 9)); queue.add(Task("Read a book", 1)); queue.add(UrgentTask("Fix the leak", 8)); print
"All tasks:"; print queue.count; fun printTask(task) { print " - " + task.describe(); }
queue.forEach(printTask); var isCritical = makeUrgencyChecker(8); print "Critical tasks:"; print
countMatching(queue, isCritical); var isLowPriority = makeUrgencyChecker(2); print "Priority >=
2:"; print countMatching(queue, isLowPriority);
```

VERIFIED DIRECTLY — THE COMPLETE PROGRAM, RUN END TO END THROUGH THE REAL INTERPRETER

```
All tasks:
4
  - URGENT: Fix the leak
  - Read a book
  - URGENT: Submit tax filing
  - Water the plants
Critical tasks:
2
Priority >= 2:
3
```

Every task prints in reverse of its own insertion order, because `add()` prepends — the linked list's own head always points at whichever task was added most recently, so walking it in `forEach` naturally visits `"Fix the leak"` (added last) first. `Critical tasks` correctly counts the two tasks with priority 8 or higher; `Priority >= 2` correctly counts three, excluding only `"Read a book"` at priority 1. Two calls to `makeUrgencyChecker` with two different thresholds produced two genuinely independent closures — `isCritical` and `isLowPriority` — the same guarantee Chapter 7 verified with `addFive` / `addTen`, reused here for a real purpose instead of a synthetic example.

Breaking It on Purpose

A capstone that only ever shows a program succeeding doesn't exercise Chapter 9's own contribution at all. Appending one deliberately broken line — passing `nil` where `forEach` expects a callable — triggers the real error-reporting pipeline this course spent an entire chapter building.

```
queue.forEach(nil);
```

VERIFIED DIRECTLY — A CORRECTLY-ATTRIBUTED, TWO-FRAME STACK TRACE

```
RUNTIME ERROR: [line 37] WispRuntimeError: can only call functions and classes
  at forEach() (called from line 82)
  at <script>
```

Line 37 is `callback(node.task);` — inside `TaskQueue.forEach`'s own body, exactly where `nil` was actually called. Line 82 is `queue.forEach(nil);` — the real call site, one frame further out. Chapter 9's own `call_stack` mechanism gets this exactly right on a program it has never seen before, built entirely out of pieces (a class method, a first-class function argument, a runtime type check) that were each verified independently, in isolation, back in the chapters that introduced them.

Where Each Piece Came From

CAPSTONE COMPONENT	CHAPTER
Tokenizing the whole program, with real line numbers	Chapter 1 (lexer) + Chapter 9 (line tracking)
Parsing classes, functions, control flow, and expressions into one AST	Chapter 2 (recursive descent), Chapter 6 (control flow grammar), Chapter 7 (call/function grammar), Chapter 8 (class grammar)
Typed nodes with `accept()`, walked by a Visitor-based interpreter	Chapter 3
Statements, truthiness, string/number/boolean literals	Chapter 4
`var`, `TaskQueue`'s own fields, the live `Environment` chain	Chapter 5
The `while` loop walking the linked list	Chapter 6
Functions as values, `makeUrgencyChecker`'s own closures	Chapter 7
`Task` / `UrgentTask` / `TaskNode` / `TaskQueue`, inheritance, `this`	Chapter 8
The line-numbered error and correctly-attributed stack trace	Chapter 9

What This Course Doesn't Cover

Wisp, as built across these ten chapters, has no array or list type (the capstone's own linked list is the workaround), no string formatting or number-to-string conversion (every `print` in this chapter avoided concatenating a number into a string, because Chapter 4's own `+` operator deliberately refuses to), no `super.method()` syntax for calling an overridden method's own parent implementation, no `break` / `continue`, no static methods or class-level fields, and no standard library beyond `print`. None of these are oversights discovered too late to fix — they're honest scope boundaries, each traceable to a specific chapter that could have covered them but chose a narrower, more teachable slice instead.

Where This Connects

Every one of these nine chapters' own components — the lexer, the AST, the Visitor pattern, environments, closures, classes, error handling — carries forward unchanged into **Course 2: Writing a Compiler/Interpreter: Advanced**. That course doesn't discard this interpreter; it rebuilds Wisp's own *runtime* as a bytecode compiler and a stack-based virtual machine, closing with a direct, measured performance comparison against the exact tree-walking interpreter finished here. The language stays the same. How it runs changes completely.

COURSE 1 COMPLETE — FUNDAMENTALS QUICK REFERENCE

- **Ch.1-2:** lexer (maximal munch, line tracking) → recursive descent parser (precedence, associativity)
- **Ch.3:** typed AST nodes + the Visitor pattern (13 duplicated isinstance checks vs. 4 `accept()` methods, verified)
- **Ch.4-5:** statements, truthiness, a live `Environment` chain (a naive copy-based version verified broken, then fixed)
- **Ch.6:** `if` / `while` / `for` (desugared), short-circuit `Logical` (verified avoiding a real crash)
- **Ch.7:** `WispFunction`, closures (a real over-capture bug verified and fixed), recursion bounded by Python's own stack (measured: 163 levels at the default limit)
- **Ch.8:** `WispClass` / `WispInstance`, `this` bound fresh per access (a real over-mutation bug verified and fixed), single inheritance via `find_method()`
- **Ch.9:** a unified error hierarchy, real line numbers, and a genuine multi-frame Wisp stack trace
- **Ch.10:** everything above, assembled into one working program — verified correct, then verified failing correctly on purpose
- **Next:** Writing a Compiler/Interpreter: Advanced — the same language, a bytecode VM instead of a tree walk