

# The wick Programming Language

A small, typed scripting language for the lantern engine

The lantern engine authors

Version 0.3 • 2026

*The wick Programming Language*, version 0.3.

wick and the lantern engine are open source under the zlib license. The language implementation — lexer, one-pass typed compiler, bytecode, and stack virtual machine — is about two thousand lines of dependency-free C++, and lives with the engine at [github.com/alikatgh/lantern](https://github.com/alikatgh/lantern).

This book was typeset in Latin Modern with L<sup>A</sup>T<sub>E</sub>X. Every code sample is written for Wick 0.3; diagnostic examples intentionally demonstrate errors.

Copyright © 2026 The lantern engine authors. Distributed under the same zlib terms as the engine: you may use, copy, and modify this text, provided the origin is not misrepresented.

# Contents

|          |                                                   |           |
|----------|---------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>1</b>  |
| 1.1      | Why wick exists . . . . .                         | 1         |
| 1.2      | The Lua lessons . . . . .                         | 2         |
| 1.3      | Design principles . . . . .                       | 3         |
| 1.4      | How to read this book . . . . .                   | 4         |
| <br>     |                                                   |           |
| <b>2</b> | <b>A wick Tutorial</b>                            | <b>5</b>  |
| 2.1      | Installing and running . . . . .                  | 5         |
| 2.2      | A game is a folder . . . . .                      | 5         |
| 2.3      | Variables and the first motion . . . . .          | 6         |
| 2.4      | Functions . . . . .                               | 7         |
| 2.5      | An optional, and a saved score . . . . .          | 7         |
| 2.6      | A list: the lamps . . . . .                       | 9         |
| 2.7      | Putting a round together . . . . .                | 9         |
| 2.8      | The development loop . . . . .                    | 10        |
| <br>     |                                                   |           |
| <b>3</b> | <b>The Language</b>                               | <b>13</b> |
| 3.1      | Lexical structure . . . . .                       | 13        |
| 3.2      | Types . . . . .                                   | 14        |
| 3.3      | Variables and scope . . . . .                     | 15        |
| 3.4      | Expressions and precedence . . . . .              | 16        |
| 3.5      | Statements . . . . .                              | 18        |
| 3.6      | Control flow . . . . .                            | 19        |
| 3.7      | Functions . . . . .                               | 20        |
| 3.8      | Assignment compatibility . . . . .                | 21        |
| <br>     |                                                   |           |
| <b>4</b> | <b>Optionals</b>                                  | <b>23</b> |
| 4.1      | The problem, in Lua . . . . .                     | 23        |
| 4.2      | The type that makes it unwritable . . . . .       | 24        |
| 4.3      | Unwrapping with ?? . . . . .                      | 24        |
| 4.4      | Narrowing with <code>if x != nil</code> . . . . . | 25        |
| 4.5      | What returns an optional . . . . .                | 26        |
| 4.6      | Optionals in the type system . . . . .            | 27        |
| 4.7      | The discipline in practice . . . . .              | 27        |

|          |                                                 |           |
|----------|-------------------------------------------------|-----------|
| <b>5</b> | <b>Collections</b>                              | <b>29</b> |
| 5.1      | Lists . . . . .                                 | 29        |
| 5.2      | Maps . . . . .                                  | 30        |
| 5.3      | Records and lists of records . . . . .          | 31        |
| 5.4      | Iteration patterns . . . . .                    | 31        |
| 5.5      | Nested data without nested containers . . . . . | 32        |
| <b>6</b> | <b>The Engine Interface</b>                     | <b>35</b> |
| 6.1      | The frame contract . . . . .                    | 35        |
| 6.2      | Frame and screen . . . . .                      | 36        |
| 6.3      | The 3D scene . . . . .                          | 36        |
| 6.4      | Meshes . . . . .                                | 37        |
| 6.5      | 2D . . . . .                                    | 38        |
| 6.6      | Audio, input, and saves . . . . .               | 38        |
| 6.7      | Determinism . . . . .                           | 39        |
| 6.8      | A culture of verification . . . . .             | 40        |
| <b>7</b> | <b>Internals</b>                                | <b>41</b> |
| 7.1      | One-pass typed compilation . . . . .            | 41        |
| 7.2      | Specialised opcodes . . . . .                   | 42        |
| 7.3      | The stack VM . . . . .                          | 42        |
| 7.4      | Frame-boundary garbage collection . . . . .     | 43        |
| 7.5      | Embedding wick . . . . .                        | 43        |
| <b>8</b> | <b>Design and Trade-offs</b>                    | <b>47</b> |
| 8.1      | The same game, twice . . . . .                  | 47        |
| 8.2      | What you stop writing . . . . .                 | 48        |
| 8.3      | What stays the same . . . . .                   | 49        |
| 8.4      | The v0.3 limits . . . . .                       | 49        |
| 8.5      | Roadmap philosophy . . . . .                    | 50        |
| <b>9</b> | <b>Bits, Bytes, and Buses</b>                   | <b>51</b> |
| 9.1      | Write the bits you mean . . . . .               | 51        |
| 9.2      | Gates and masks . . . . .                       | 51        |
| 9.3      | A register has a width . . . . .                | 52        |
| 9.4      | Pairs and memory . . . . .                      | 52        |
| 9.5      | Make the state readable . . . . .               | 53        |
| 9.6      | The workbench and the next game . . . . .       | 53        |
| <b>A</b> | <b>Grammar</b>                                  | <b>55</b> |
| <b>B</b> | <b>Compile-Error Catalogue</b>                  | <b>57</b> |

|                                              |           |
|----------------------------------------------|-----------|
| <i>CONTENTS</i>                              | iii       |
| <b>C Quick Reference</b>                     | <b>61</b> |
| C.1 Bits and register widths (0.3) . . . . . | 63        |



## Chapter 1

# Introduction

*The wick is the part of the lantern that carries the flame.* wick is the lantern engine's own scripting language: Lua's size and feel, with the sharp edges designed out at the language level rather than papered over with lint rules. This chapter explains why a game engine grew its own language, what problems that language is meant to remove, and how the rest of the book is arranged.

## 1.1 Why wick exists

wick was not written because the world needed another language. It was written because we shipped real games in Lua, kept a list of what hurt, and found that every entry on the list had the same shape: a mistake the language allowed us to make, discovered only when a player hit it.

Lua is a wonderful language. It is small, embeddable, fast, and famously easy to bind to C. For a decade it has been the default answer to what should I script my game in.” We are not here to argue against any of that. We are here because a scripting language for a game engine is a specific job with specific failure modes, and Lua's defaults — chosen for generality and for the era it was born in — turn out to be exactly the wrong defaults for that job.

Consider the failure we keep coming back to, the one that gives Chapter 4 its spine. Lantern Night, our showcase game, saves the player's best score to a file. In the Lua build the load looked like this:

```
local best = tonumber(lt.load_save("lanternnight_best") or "0")
```

It reads correctly. It passed review. It shipped. And it crashed on the first frame whenever a save was interrupted mid-write, because an interrupted write leaves a zero-byte file, `lt.load_save` returns the empty string `""` rather than `nil`, the empty string is *truthy* in Lua so the `or "0"` fallback never fires, and `tonumber("")` is `nil`, which then poisons every arithmetic expression downstream. Four separate Lua design decisions — *truthy* empty strings, *nil*-punning error returns, silent *nil* propagation, and no static

checking — lined up to turn one interrupted disk write into a first-frame crash that no amount of reading the code would reveal.

You cannot write that bug in wick. Not because we added a special case for saves, but because the type system will not compile a program that uses a value which might be `nil` without first dealing with the `nil`. The wick version of that line is the only version the compiler accepts:

```
let best = num(lt.load_save("lanternnight_wick_best")) ?? "" ?? 0
```

That is the whole thesis of the language in one line. Every design decision below serves it.

## 1.2 The Lua lessons

Here is the catalogue — each Lua pain, and the wick decision it produced. None of these is a lint rule or a style guide; each is enforced by the compiler or the runtime, so it cannot be forgotten under deadline.

| Lua pain                                                          | wick answer                                                                                                                 |
|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <code>nil</code> errors at run time                               | Static types plus T? optionals; unchecked use does not compile.                                                             |
| Typos create silent globals                                       | Locals only; assigning an undeclared name is a compile error.                                                               |
| 1-based indexing, no <code>continue</code><br>"" and 0 are truthy | 0-based [], <code>continue</code> , <code>len(x)</code> .<br>Conditions must be <code>bool</code> ; there is no truthiness. |
| GC pauses mid-frame                                               | Collection runs <i>between</i> frames only.                                                                                 |
| Stack-index C bindings                                            | Engine calls are type-checked at compile time.                                                                              |
| <code>math.random</code> differs by machine                       | <code>rand()</code> is a fixed xorshift — deterministic everywhere.                                                         |
| JIT forbidden on iOS                                              | No JIT anywhere; specialized byte-code from static types.                                                                   |

The pattern is worth naming. In each row, Lua’s behaviour is defensible in the abstract and dangerous in a game loop. Truthiness is convenient until a score of zero silently takes the wrong branch. Implicit globals are terse until a typo becomes a `nil` that surfaces three functions away. A garbage collector that runs whenever it likes is fine for a batch job and a dropped frame in an action game. wick’s answer in every case is the same move: take the thing Lua does dynamically and at run time, and do it

statically at compile time instead — or, in the GC’s case, at a moment the host chooses.

## 1.3 Design principles

Four principles run through the whole language. When a later chapter explains why some feature works the way it does, the answer is almost always one of these.

**Small.** `wick` is deliberately Lua-1.0 small. The entire implementation is about two thousand lines of dependency-free C++: lexer, compiler, virtual machine, garbage collector, and built-ins. There are no closures, no classes, no modules, no metatables, no varargs, and no string methods in v0.3. Every one of those omissions is listed in Chapter 8 with its workaround and its place on the roadmap. Smallness is not an accident we are apologising for; it is a feature we are defending. A language you can read in an afternoon is a language whose behaviour you can predict at three in the morning.

**Typed.** Every expression has a static type, known at the moment the compiler emits code for it. Types are inferred wherever inference has something to work with, so the surface stays light — you write `let x = 10`, not `let x: num = 10` — but the checking is total. The headline of the type system is the optional, `T?`, which makes this “might be absent” a fact the compiler tracks and the programmer must discharge.

**Deterministic.** The same program, given the same inputs, produces the same frames on every machine. `rand()` is a fixed xorshift generator with no time seeding; if you want variety between runs you seed `srand()` yourself, with a number you chose and can log. Under `LANTERN_FIXED_DT` the frame clock is fixed-step, so a whole gameplay session replays byte-for-byte. This is what makes screenshot-based CI possible, and Chapter 6 builds a testing culture on top of it.

**Frame-aware.** `wick` knows it lives inside a 60-frames-per-second loop. The garbage collector runs only when the host asks, which lantern does exactly once per frame after presenting the image — the one moment a stall is invisible. The engine API is organised around two functions, `update(dt)` and `draw()`, that the host calls every frame. The language is not a general-purpose tool that happens to be embedded in a game; it is shaped by the game loop from the lexer up.

## 1.4 How to read this book

The book is meant to be read front to back the first time and used as a reference thereafter.

- **Chapter 2** is a tutorial. It installs the engine, runs the first program, and builds a small complete lamp game step by step, so you have working code under your hands before the reference chapters get precise.
- **Chapter 3** is the language reference: lexical structure, types, scope, expressions, precedence, statements, control flow, and functions.
- **Chapter 4** is about optionals — the heart of wick — told partly through the zero-byte-save war story from this introduction.
- **Chapter 5** covers the collections, lists and maps, and the parallel-lists idiom that stands in for structs.
- **Chapter 6** documents the `lt.*` engine interface: the frame contract, the typed API tables, determinism, and verification.
- **Chapter 7** opens the implementation: one-pass typed compilation, the stack VM, frame-boundary garbage collection, and how to embed wick in your own C++.
- **Chapter 8** is the design rationale — wick against Lua on the same real game, the honest trade-offs, and the roadmap philosophy.
- The **appendices** give the full grammar in EBNF, the complete compile-error catalogue, and quick-reference tables for the built-ins and the engine API.

Throughout, code set in **this face** is wick unless a caption says otherwise. Every listing is real v0.3 wick and would compile; where a listing is deliberately wrong to show a compile error, the error message is quoted next to it. When we show Lua, it is the twin of a wick sample, and it is labelled as Lua.

A note on honesty. This book will tell you, more than once, where Lua still does something better than wick does today — dynamic tables can hold nested data as tables, multiple return values are genuinely missing, the ecosystem is enormous and ours is one engine. A language book that only flatters its subject is not much use. wick earns its place by removing a specific and expensive class of bug, and it is worth being precise about what it removes and what it does not.

## Chapter 2

# A wick Tutorial

This chapter builds a small but complete game, from an empty folder to something you can play. We introduce language features as we need them and explain them only as far as the game requires; the precise rules wait for Chapter 3. By the end you will have written variables, functions, an optional, a list, a loop, and the two functions the engine calls every frame — and you will have used the two tools that make wick pleasant to work in, hot reload and the on-screen error screen.

## 2.1 Installing and running

wick ships inside the lantern engine. There is nothing separate to install: if you can build the engine, you can run wick.

```
git clone https://github.com/alikatgh/lantern && cd lantern
brew install sdl2 lua cmake pkg-config # macOS; Linux: same
packages
cmake -B build && cmake --build build -j8

./build/lantern games/wicklab # the first wick program
./build/lantern games/showcase_wick # Lantern Night, in wick
```

The first command builds the engine. The last two run games that already exist in the repository: `wicklab`, a small tour of the language, and `showcase_wick`, the wick build of Lantern Night that recurs throughout this book.

## 2.2 A game is a folder

A wick game is a directory that contains a file named `main.wick`. When you point the engine at a folder, it runs that file if present, falling back to `main.lua` otherwise. To start a new game:

```
mkdir games/lamp
$EDITOR games/lamp/main.wick
```

```
./build/lantern games/lamp
```

The screen is 400 by 240 pixels. The engine runs at 60 frames per second and looks up two functions by name each frame:

```
fn update(dt: num) {    // dt = seconds since last frame, capped
    at 0.1
}

fn draw() {             // 3D first, then 2D composites on top
    lt.clear(0.1, 0.1, 0.2)
    lt.print("HELLO 400X240", 4, 4, 1, 1, 1, 1)
}
```

Both functions are optional; a `main.wick` with only top-level statements is a legal program. Top-level statements — everything not inside a `fn` — run once, at load, and that is where you create resources and set up state. Save the file with those two functions in it, run the folder, and you have a window that clears to a dark blue and prints a line of text in the built-in 8-by-8 font. That is the smallest thing worth calling a game.

## 2.3 Variables and the first motion

Let us make something move. Every variable is introduced with `let`. Types are inferred from the initialiser, so you rarely write them:

```
let x = 200.0    // num, inferred
let y = 120.0
```

Those are top-level `lets`, so `x` and `y` are program globals: they persist across frames, which is exactly what we want for the player's position. Now read the keyboard in `update` and draw a square at `(x, y)` in `draw`:

```
let x = 200.0
let y = 120.0

fn update(dt: num) {
    let speed = 120.0
    if lt.key("left") { x = x - speed * dt }
    if lt.key("right") { x = x + speed * dt }
    if lt.key("up") { y = y - speed * dt }
    if lt.key("down") { y = y + speed * dt }
    x = max(8, min(392, x))
    y = max(8, min(232, y))
}
```

```

}

fn draw() {
  lt.clear(0.09, 0.07, 0.16)
  lt.rect(x - 4, y - 4, 8, 8, 1, 0.85, 0.3, 1)
}

```

A few things are already worth noticing. `speed` is declared with `let` inside `update`, so it is a local: it exists only for that call. Multiplying by `dt`, the seconds elapsed since the last frame, makes the motion frame-rate independent — the square moves 120 pixels per second whether the machine runs at 60 or 144 frames. `max` and `min` are built-ins, always available with no namespace. And `lt.key` returns a `bool`, which is the only thing an `if` will accept: there is no truthiness in wick, so you cannot write `if x` and hope it means if `x` is nonzero.” It simply will not compile.

## 2.4 Functions

A function is declared with `fn`, always at the top level — functions do not nest in v0.3. Parameter types are required; the return type follows the parameter list after a colon, and you omit it for a function that returns nothing. Here is a tiny helper that makes the square bob up and down:

```

fn wobble(v: num): num {
  return sin(v * 2) * 6
}

```

`sin` is a built-in and takes radians. To use `wobble` we need a clock, so add a top-level `let t = 0.0` and advance it in `update` with `t = t + dt`. Then the draw becomes:

```

lt.rect(x - 4, y - 4 + wobble(t), 8, 8, 1, 0.85, 0.3, 1)

```

One rule to remember: functions must be declared before they are called (the compiler is single-pass; Chapter 7 explains why). `update` and `draw` are called by the host, not by your code, so their position relative to each other does not matter — but `wobble` must appear above the line that calls it.

## 2.5 An optional, and a saved score

Now the part that makes wick wick. We want to keep a running count and save it to disk. Saving is easy:

```
lt.save("lamp_best", str(best))
```

`str` converts a `num` to a `str`; `lt.save` writes bytes to a named save slot. Loading is where the type system earns its keep. `lt.load_save` cannot promise a value — the save may not exist, or may be unreadable — so its return type is not `str` but `str?`, an *optional* string. You cannot use a `str?` where a `str` is wanted; the compiler stops you:

```
let s = lt.load_save("lamp_best")    // s: str?
lt.print(s, 4, 4, 1, 1, 1, 1)        // COMPILE ERROR: expected
                                     str, got str?
```

The error is `expected str, got str?`, and it is the good kind of error: it happens the moment you write the line, not on some player's machine a month after release. To turn a `str?` into a `str` you must say what happens when it is `nil`. The quickest way is `??`, which supplies a default:

```
let best = num(lt.load_save("lamp_best") ?? "") ?? 0
```

Read it inside out. `lt.load_save("lamp_best")` is a `str?`. The first `??` turns it into a `str`, using `""` when the save is absent. `num` parses a `str` into a number, but parsing can fail — `num("banana")` and even `num("")` cannot produce a number — so `num` returns `num?`. The second `??` turns that into a plain `num`, using `0` when the parse failed. Both ways the value can be missing are handled, in one line, because the compiler refused to let either slip through. This is the zero-byte-save bug from the introduction, made unwritable. Chapter 4 tells the whole story.

Wire it up: declare `best` at the top level, and bump it when the player presses Z:

```
let best = num(lt.load_save("lamp_best") ?? "") ?? 0

fn update(dt: num) {
  // ... movement from before ...
  if lt.pressed("z") {
    best = best + 1
    lt.save("lamp_best", str(best))
  }
}
```

`lt.pressed` is true only on the frame a key goes down, where `lt.key` is true for as long as it is held. Run the game, press Z a few times, quit, and run it again: the count survives, because it was saved and safely reloaded.

## 2.6 A list: the lamps

The game is called *lamp*, so let us add lamps. We will keep four of them, each with a position and a lit/unlit flag. This original tutorial uses parallel lists; records are introduced in Chapter 5, so we use the idiom that stands in for them: *parallel lists*, where index `i` names one lamp across several lists.

```
let lamp_x: list<num> = []
let lamp_z: list<num> = []
let lamp_lit: list<bool> = []
```

An empty list literal `[]` has no elements to infer a type from, so it *requires* an annotation — that is what `list<num>` on the left is for. Fill the lists with four lamps:

```
push(lamp_x, 3.4) push(lamp_z, 3.4) push(lamp_lit, true)
push(lamp_x, -3.4) push(lamp_z, 3.4) push(lamp_lit, true)
push(lamp_x, -3.4) push(lamp_z, -3.4) push(lamp_lit, true)
push(lamp_x, 3.4) push(lamp_z, -3.4) push(lamp_lit, true)
```

`push` appends, and it is type-checked: `push(lamp_lit, 3)` would not compile, because `lamp_lit` holds `bool`. Lists are 0-based and you iterate them with a numeric range. Here is a function that relights every lamp, using a `for` loop over the half-open range `[0, 4)`:

```
fn reset_lamps() {
  for i in 0..4 { lamp_lit[i] = true }
}
```

`for i in a..b` runs `i` from `a` up to but not including `b` — end-exclusive, to match 0-based indexing, so `0..4` and `0..len(lamp_lit)` mean the same thing here. Reading or writing an index that is out of range is a runtime error with a line number, not a silent `nil`; we will meet that error screen in a moment.

## 2.7 Putting a round together

We now have enough to make a game with a goal: lamps go dark over time, and the player relights the nearest one by standing next to it and pressing Z. The full logic is in `games/showcase_wick/main.wick`; here is the shape of the update, condensed to the ideas we have covered.

---

```

let score = 0

fn nearest_unlit(): num {      // lamp index, or -1 if all lit
  let bi = 0 - 1
  let bd = 0.0
  for i in 0..4 {
    if not lamp_lit[i] {
      let dx = lamp_x[i] - x
      let dz = lamp_z[i] - y
      let d = dx * dx + dz * dz
      if bi < 0 or d < bd { bi = i  bd = d }
    }
  }
  return bi
}

fn update(dt: num) {
  // ... movement, save on Z ...
  let i = nearest_unlit()
  if i >= 0 {
    if lt.pressed("z") { lamp_lit[i] = true  score = score + 1 }
  }
}

```

Two things here look forward to later chapters. `nearest_unlit` returns a plain `num` — the index, or `-1` as a sentinel — because this tutorial does not use multiple return values, so returning the index or nothing” is done with an out-of-band value rather than an optional. And the nested `if` on the last two lines, rather than one `if i >= 0 and lt.pressed("z")`, keeps the condition simple; Chapter 3 covers when the compiler asks you to parenthesise combined conditions.

## 2.8 The development loop

Two features turn the write-run-fix cycle from minutes into seconds.

**Hot reload.** Save `main.wick` while the engine is running and it recompiles and reloads live. All previously created resources are freed first, so reloading never leaks — you can iterate on gameplay for an hour without restarting. Change a constant, save, and watch the running game change.

**The error screen.** When your file has a mistake, the engine does not crash to the terminal. It renders the error in-engine, as `file:line:message`, and keeps waiting. Compile errors — a `str?` used as a `str`,

a + between a string and a number, a call to a function defined below its use — and runtime errors alike — an out-of-range index, a failed `check()` — all show the same way. Fix the file, save, and the game hot-reloads on top of the error. You rarely leave the running game.

For example, deleting the `?? 0` from our `best` line produces, on screen:

```
main.wick:1: expected num, got num?
```

and the game waits, showing you the line, until you put it back.

**Deterministic screenshots.** Set `LANTERN_SHOT` at a path and the engine captures frame 60 as a BMP and then exits. Add `LANTERN_FIXED_DT=1` and that capture is byte-identical on every machine, because the frame clock is fixed-step and wick's `rand()` is deterministic. Whole gameplay sessions replay exactly. Chapter 6 builds a testing discipline on this; for now it is enough to know that it looks right on my machine” can be turned into a check that means the same thing on every machine.

That is a complete game and the whole workflow around it. The rest of the book makes precise what this chapter showed by example.



## Chapter 3

# The Language

This is the reference chapter. It describes wick v0.3 completely and precisely: how source text is turned into tokens, what types exist, how variables are scoped, how expressions are formed and in what order they bind, what statements there are, how control flows, and how functions are declared and called. Where the tutorial was content to gesture, this chapter states rules. The grammar in Appendix A is the formal companion to the prose here.

## 3.1 Lexical structure

A wick program is a sequence of Unicode bytes, read left to right and top to bottom, that the lexer turns into a stream of tokens. Whitespace and newlines separate tokens but are otherwise insignificant — wick has no newline-sensitive rules and no semicolons. A statement ends where the grammar says it ends, not at a line break.

**Comments.** A comment begins with `//` and runs to the end of the line. There are no block comments.

```
let x = 10    // this is a comment; it ends at the newline
```

**Identifiers.** An identifier matches `[A-Za-z_][A-Za-z0-9_]*`: a letter or underscore, then any number of letters, digits, or underscores. Identifiers are case-sensitive. The following words are reserved and may not be used as identifiers:

```
let fn if elif else while for in break
continue return true false nil and or not
record
```

That is the keyword list. Type names (`num`, `bool`, `str`, `list`, `map`) are *not* keywords; they are contextual, recognised only where a type is expected, so nothing stops you from calling a variable `str` if you are determined to confuse yourself.

**Numbers.** There is one numeric type, `num`, an IEEE-754 double. A numeric literal is a run of digits with an optional single decimal point: `42`, `3.5`, `0.25`. Hexadecimal `0xFF` and binary `0b1010` integer literals are also accepted, through `0xFFFFFFFF` inclusive. Prefixes accept either case. Invalid digits, missing digits, underscores, fractions after a prefix, and overflow are compile errors. Exponent notation remains unsupported. The lexer is careful about the range operator: `1..5` lexes as the number `1`, the range token `..`, and the number `5` — never as `1.` followed by `.5`. If you want a literal with a fractional part next to a range you must space it, but in practice ranges use integer bounds.

**Strings.** A string literal is enclosed in double quotes and may contain the escapes `\n` (newline), `\t` (tab), `\"` (a literal quote), and `\\` (a literal backslash). Strings are immutable. There are no single-quoted strings and no multi-line string literals.

```
let a = "tenzin"
let b = "line one\nline two"
let c = "she said \"hi\""
```

**Operators and punctuation.** The lexer recognises the following multi-character tokens, greedily: `..` (range), `??` (nil-coalesce), `==`, `!=`, `<=`, `>=`. The single-character tokens are the arithmetic and comparison operators `+` `-` `*` `/` `%` `<` `>`, the assignment `=`, the optional marker `?`, the member dot `.`, and the grouping and separator characters `(` `)` `[` `]` `{` `}` `,` `..`. Any other character is a lexical error, reported as `unexpected character 'x'`.

## 3.2 Types

Every expression in wick has a static type, fixed at compile time. There are scalar, container, optional, and named record types.

| Type                       | Values                                 | Notes                                                      |
|----------------------------|----------------------------------------|------------------------------------------------------------|
| <code>num</code>           | IEEE-754 double                        | the only number type                                       |
| <code>bool</code>          | <code>true</code> , <code>false</code> | the only type conditions accept                            |
| <code>str</code>           | immutable text                         | + concatenates <code>str</code> with <code>str</code> only |
| <code>list&lt;T&gt;</code> | ordered, 0-based                       | scalar or flat record elements                             |
| <code>map&lt;T&gt;</code>  | string-keyed                           | scalar or flat record values                               |
| <code>T?</code>            | <code>T</code> or <code>nil</code>     | an <i>optional</i> — the headline feature                  |

The element type of a `list` or `map` may be `num`, `bool`, `str`, or a declared flat record type. There are no nested containers in v0.3: `list<list<num>>` and `map<list<str>>` do not exist, and attempting one produces `container elements must be scalar or record types`. Chapter 5 gives the index-arithmetic workaround.

**Records.** A top-level `record` declares named fields of `num`, `bool`, or `str`. Construct a value with every field named, then access or assign fields with a dot. There are no methods or nested fields.

```
record Register { value: num, name: str }
let a = Register { value: 0, name: "A" }
a.value = 0xFF
check(a.value == 255, "record field")
```

**nil and optionals.** `nil` is not a value of general type. It inhabits only optionals: an expression of type `T?` may be `nil` or may hold a `T`, and there is no way to write a bare `nil` whose type is unknown. `let x = nil` does not compile; the compiler cannot infer what the optional is optional *of*, and says so: `can't infer a type from nil`; `annotate: let x: str? = nil`. Supply the annotation and it is fine:

```
let x: str? = nil    // x is str?, currently nil
```

The whole of Chapter 4 is about optionals; here we note only that `T?` is a first-class type that the compiler tracks, and that you cannot use a `T?` where a `T` is required.

**void.** A function declared without a return type has return type `void`. `void` is not a value: a call to a void function is a statement, not an expression, and cannot appear on the right of a `let` or inside a larger expression. Attempting it yields `can't assign a void expression`.

## 3.3 Variables and scope

Every variable is introduced by `let`. There is no other way to bring a name into existence — in particular, assigning to a name that was never declared is not an implicit declaration but a compile error. This single rule removes the entire class of typo-creates-a-global bug that haunts Lua.

```
let x = 10           // num, inferred from 10
let y: num = 10      // the same, written out
```

```
let s = "hi"           // str
```

The type may be inferred from the initialiser or written explicitly after a colon. You must write it when there is nothing to infer from — an empty container literal or a bare `nil` — and you may write it anywhere for clarity. When both are present they must agree.

**Global versus local.** A `let` at the top level of the file creates a program *global*: it is initialised once when the file loads and it persists for the life of the program, across every frame. A `let` inside a function creates a *local*: it lives in that call’s stack frame and is gone when the call returns. This is why per-frame state — the player position, the score — lives in top-level `lets`, while scratch values live in locals.

**Blocks and shadowing.** A variable is scoped to the block — the brace-delimited region — in which it is declared, and to the blocks nested inside it. An inner block may *shadow* a name from an outer scope by declaring a new `let` with the same name; the inner name hides the outer one until the inner block ends. Redefining a name in the *same* scope is an error:

```
let n = 1
if true {
  let n = 2           // ok: shadows the outer n inside this block
}
let n = 3             // ERROR: 'n' already declared in this scope
```

The error for the last line is `'n' already declared`. To change an existing variable you assign to it; you do not re-`let` it.

**Assignment.** Assignment uses a single `=` and requires the name to have been declared. Reading an undeclared name reports `unknown variable 'x'` (declare it with `let`); assigning to one reports `unknown variable 'x'` -- wick has no implicit globals; use `let`. The assigned value must have a type compatible with the variable (see the compatibility rules below).

## 3.4 Expressions and precedence

Expressions combine literals, variables, calls, indexing, and operators. Operators bind according to the following precedence table, from loosest (evaluated last, binds least tightly) to tightest. All binary operators are left-associative.

| Level | Operators                          | Meaning                     |
|-------|------------------------------------|-----------------------------|
| 1     | <b>or</b>                          | logical or (short-circuit)  |
| 2     | <b>and</b>                         | logical and (short-circuit) |
| 3     | <b>??</b>                          | nil-coalesce                |
| 4     | <b>==   !=</b>                     | equality, inequality        |
| 5     | <b>&lt;   &lt;=   &gt;   &gt;=</b> | ordered comparison          |
| 6     | <b>+   -</b>                       | add / concatenate, subtract |
| 7     | <b>*   /   %</b>                   | multiply, divide, modulo    |
| 8     | <b>unary -, not</b>                | negation, logical not       |
| 9     | <b>f(x), a[i]</b>                  | call, index                 |

So **a or b and c** parses as **a or (b and c)**, and **1 + 2 \* 3** is **1 + (2\*3)**, as arithmetic convention demands. The one surprise for newcomers is that **??** binds looser than comparison and arithmetic but tighter than **and/or**, which is why **num(s) ?? 0** works without parentheses — the coalesce applies to the whole **num(s)**, and the result is a plain **num** ready for use.

**Arithmetic.** **+ - \* / %** operate on **num** and produce **num**. Both operands must be non-optional **num**; using an optional is an error until you unwrap it. Division by zero follows IEEE-754 (it produces infinity or NaN rather than trapping), as in Lua.

**The two faces of +.** **+** is numeric addition when both operands are **num**, and string concatenation when both are **str**. It is never mixed. There is no implicit conversion: **"score " + 5** does not compile, because one side is **str** and the other **num**. The error is explicit about the fix: **'+' needs num+num or str+str (use str(x))**. Write **"score " + str(5)**. This is deliberate — the silent stringification of numbers is a rich source of bugs, and wick would rather you say what you mean.

**Comparison and equality.** The ordered comparisons **< <= > >=** require **num** operands and produce **bool**. Equality **==** and inequality **!=** compare two values of the same type and produce **bool**; for **str**, equality compares contents, not identity. Comparing values of different types is an error (**'==' operands must have the same type**), with one blessed exception: comparing an optional to **nil**, which is the narrowing idiom covered in Chapter 4. Comparing a *non*-optional to **nil** is itself an error — **comparing non-optional to nil** — because a non-optional can never be **nil**, so the check is dead code.

**Logical operators.** **and** and **or** require **bool** operands, produce **bool**, and short-circuit: **a and b** does not evaluate **b** when **a** is false, and **a or b**

does not evaluate `b` when `a` is true. `not` requires a `bool` and negates it. Because there is no truthiness, you cannot write `a and b` where `a` is a number; you write the comparison that yields the `bool` you mean.

**The nil-coalesce `??`.** `a ?? b` requires `a` to be an optional `T`?; it evaluates to `a` unwrapped when `a` is non-`nil`, and to `b` otherwise. The default `b` must have the base type `T`, and the whole expression has type `T` — non-optional. The right side is evaluated only when the left is `nil`. Applying `??` to a non-optional left side is an error (`'??' left side must be an optional`), because there is nothing to coalesce.

**Combined conditions and parentheses.** When a narrowing comparison `x != nil` is combined with `and` or `or`, the compiler asks you to parenthesise, so that the reader — and the narrowing analysis — can see the structure unambiguously. The message is `parenthesize this condition: (a != b) and ...`. Adding the parentheses resolves it:

```
if (s != nil) and ready { }    // parenthesised as the compiler
                               asks
```

## 3.5 Statements

A wick program body is a sequence of statements. The statement forms are:

```
let x = expr           // declaration (with optional : type)
x = expr               // assignment to a declared variable
xs[i] = expr           // element assignment (list index or map
    key)
f(a, b)                // call statement; a non-void result is
    discarded
if c { } elif c2 { } else { }
while c { }
for i in a..b { }      // i: num, from a to b-1
break                 // exit the innermost loop
continue              // next iteration of the innermost loop
return expr           // or bare 'return' in a void function
```

Blocks always use braces; there is no single-statement form without them. A call used as a statement discards its result if it has one, which is how you call a function for its effect. Element assignment works on a list index

or a map key; the target must already be a list or map (indexing anything else is only lists and maps can be indexed).

## 3.6 Control flow

**Conditionals.** `if`, optional `elif` chains, and an optional `else` form the conditional. Every condition must be `bool`. There is no truthiness, so `if 1 { }` and `if name { }` do not compile; the error names the rule directly: `if condition must be bool (wick has no truthiness)`. Write the comparison you mean — `if x > 0`, `if s != ""`.

```
if score > best {
  best = score
} elif score == best {
  // tie
} else {
  // nothing
}
```

**while.** A `while` loop runs its body as long as the `bool` condition holds. The condition is checked before each iteration.

```
let i = 0
while i < 4 {
  i = i + 1
}
```

**for ranges.** `for i in a..b` iterates the loop variable `i` over the half-open range `[a, b)` — from `a` up to but not including `b`. Both bounds are `num` expressions, evaluated once. The loop variable is a fresh local, scoped to the loop body, of type `num`. This is the only iteration form in v0.3: there is no `for x in list`. To walk a list you iterate its indices:

```
for i in 0..len(xs) {
  lt.print(str(xs[i]), 4, 4 + i * 10, 1, 1, 1, 1)
}
```

End-exclusivity is a deliberate match to 0-based indexing: `0..len(xs)` visits exactly the valid indices 0 through `len(xs) - 1`, with no off-by-one to get wrong.

**break and continue.** `break` exits the innermost enclosing loop; `continue` skips to that loop’s next iteration. Both outside any loop are errors: `break` outside a loop, `continue` outside a loop.

## 3.7 Functions

A function is declared with `fn`, at the top level only — functions do not nest, and there are no closures or first-class function values in v0.3. Attempting a nested `fn` reports `fn declarations can’t nest`.

```
fn dist(ax: num, az: num, bx: num, bz: num): num {  
  let dx = bx - ax  
  let dz = bz - az  
  return sqrt(dx * dx + dz * dz)  
}
```

**Parameters and return type.** Every parameter must be written with its type; there is no inference for parameters. The return type, if any, follows the parameter list after a colon. Omit it and the function is void. A void function may use bare `return` to exit early but may not `return expr` (void function can’t return a value); a function with a declared return type must `return expr` of that type on the paths that produce a value, and may not `return` bare (this function must return T).

**Declare before use.** A function must be declared textually above its first call. The compiler is single-pass and resolves each call as it reaches it, so a call to a function defined later in the file reports `unknown function ‘f’` (wick requires `declare-before-use`). The remedy is to move the definition up. The two host-called functions, `update` and `draw`, are exempt from ordering relative to each other, because the host – not your code — looks them up by name each frame.

**Recursion.** A function may call itself, since its own name is in scope within its body:

```
fn fib(k: num): num {  
  if k < 2 { return k }  
  return fib(k - 1) + fib(k - 2)  
}
```

**Call checking.** Calls are fully type-checked. Passing the wrong number of arguments reports `f() needs at least N arguments` or `f() takes N arguments`; passing an argument of the wrong type reports, for instance, `f() argument 3: expected num, got str`, naming the function and the argument position. This applies equally to your own functions and to engine calls, which are registered with typed signatures (Chapter 6).

**Return-path coverage.** v0.3 does not yet verify that every path through a non-void function returns a value. If control falls off the end of a function that declares a return type, the call yields `nil` — do not rely on this. It is listed among the limits in Chapter 8, and the discipline is simple: make the last statement of a value-returning function a `return`.

## 3.8 Assignment compatibility

The rules that govern when one type may be stored where another is expected are short and worth stating together.

- A `T` may be stored where a `T?` is expected — widening to an optional is free and always safe.
- A `T?` may *not* be stored where a `T` is expected — you must unwrap it first, with `??` or by narrowing.
- Containers are invariant: a `list<num>` accepts only `num` elements, and a `list<num>` is not a `list<num?>` or anything else. The element type is part of the type and must match exactly.
- `void` cannot be stored at all, because it is not a value.

These four rules, together with the optional discipline of the next chapter, are the whole of wick's type checking. There is no subtyping beyond `T-into-T?` widening, no coercion, and no generics beyond the built-in `list` and `map`. The system is small on purpose — small enough to hold in your head, which is the point of the whole language.



## Chapter 4

# Optionals

If `wick` has a single reason to exist, it is the optional. Everything else — the static types, the ban on truthiness, the compile-time engine calls — is supporting structure around one claim: that this value might be absent” should be a fact the compiler knows and the programmer must discharge, not a runtime surprise that surfaces three functions away from its cause. This chapter explains optionals fully, and tells the story of the bug that made them non-negotiable.

## 4.1 The problem, in Lua

Start with the failure, because the feature is shaped exactly to its contours. Lantern Night keeps a high score in a save file. The Lua build loaded it like this:

```
local best = tonumber(lt.load_save("lanternnight_best") or "0")
```

Every clause is reasonable. `lt.load_save` returns the saved string, or `nil` if there is no save; the `or "0"` supplies a default for the `nil` case; and `tonumber` parses the string to a number. It reads correctly, it passed review, and it shipped.

Then a player force-quit the game during a save. A save is a write to disk, and a write interrupted at the wrong instant leaves the file created but empty — zero bytes. On the next launch, `lt.load_save` opened that file, read it successfully, and returned not `nil` but the empty string `""`. Now walk the expression again with `""` in hand:

- `lt.load_save(...)` is `""` — a real string, not `nil`.
- `"" or "0"` is `""`, because in Lua the empty string is *truthy*. The fallback never fires.
- `tonumber("")` is `nil`, because the empty string is not a number.
- `local best = nil`, and every later use of `best` — comparisons, arithmetic, drawing the number — now operates on `nil`.

The game crashed on the first frame. Not at the save, not at the load, but downstream, where `nil` finally met an operation that could not tolerate it. The stack trace pointed at a line that was completely innocent. Four separate Lua decisions had to line up for this — truthy empty strings, a load that returns "" rather than `nil` for an empty file, `nil`-punning parse failure, and no static checking to catch the `nil` before it spread — and all four are defensible in isolation. Together they turned an interrupted disk write into a first-frame crash that reading the code would never reveal.

## 4.2 The type that makes it unwritable

wick’s answer is a type. An operation that can fail to produce a value does not return that value; it returns an *optional*, written `T?`, which is either a `T` or `nil`. And the compiler will not let you use a `T?` as a `T`.

```
let s = lt.load_save("hi")      // s: str?
lt.print(s, 4, 4, 1, 1, 1, 1)  // COMPILER ERROR: expected str,
                               got str?
```

The second line does not compile. The error is `expected str, got str?`, and it appears the instant you write the line, on your machine, not on a player’s. There is no path by which a `str?` reaches `lt.print`, which wants a `str`. To get from `str?` to `str` you must state, in the program, what happens when the value is `nil`. wick gives you two ways to do that.

## 4.3 Unwrapping with ??

The nil-coalesce operator `??` unwraps an optional by supplying a default for the `nil` case. `a ?? b` evaluates to `a` when `a` is non-`nil`, and to `b` otherwise; the result type is the non-optional base type.

```
let text = s ?? "no save yet"   // text: str
let n = num("maybe") ?? 0     // n: num (num parses str ->
                               num?)
```

The default must match the base type: `s ?? "..."` is fine because `s` is `str?` and the default is `str`; a mismatched default reports `??? default must be str` (or whatever the base type is). The right side is evaluated lazily, only when the left is `nil`, so it is safe to put work there. And `??` requires a genuine optional on its left — using it on a value that can never be `nil` is the error `??? left side must be an optional`, because there is nothing to coalesce.

Now return to the war story. The wick translation of that fatal Lua line is:

```
let best = num(lt.load_save("lanternnight_wick_best") ?? "") ?? 0
```

Trace the zero-byte file that crashed the Lua build.

`lt.load_save(...)` returns `str?`. Whether it is `nil` (no file) or `""` (empty file) makes no difference to what follows, because the first `??` turns *both* into a `str`: `nil` becomes `""`, and `""` is already `""`. Then `num("")` is `num?` — and it is `nil`, because the empty string is not a number. The second `??` catches that and yields 0. There is no value of the save file, present or absent, well-formed or empty, that produces anything but a clean `num`. The bug is not fixed; it is *unwritable*. Delete either `??` and the program does not compile.

That is the difference worth internalising. In Lua the safe version and the buggy version look almost identical, and the compiler accepts both; the difference surfaces only when a specific rare input arrives in production. In wick the buggy version — the one missing a `??` — is a compile error, and the safe version is the only one that builds. The type system has moved the discovery of the bug from a player's first frame to the author's first save.

## 4.4 Narrowing with if x != nil

The second way to handle an optional is to *narrow* it: test it against `nil`, and inside the block where the test succeeded, use it as a plain `T`.

```
if s != nil {
  // inside this block, s has type str, not str?
  lt.print(s, 4, 4, 1, 1, 1, 1)
}
```

The comparison `s != nil` does double duty: it is a `bool` condition, and it tells the type checker that within the guarded block `s` cannot be `nil`, so its type there is `str`. This is the same move a reader makes intuitively — we already checked it is not `nil`, so here it is a string” — made formal and enforced.

Narrowing is the right tool when you want to do several things with the value and a default would be awkward, or when the `nil` case has its own behaviour:

```
let saved = lt.load_save("name")
if saved != nil {
```

```
lt.print("WELCOME BACK " + saved, 4, 4, 1, 1, 1, 1)
} else {
  lt.print("NEW PLAYER", 4, 4, 1, 1, 1, 1)
}
```

**The limits of narrowing.** Narrowing in v0.3 works on *local* variables. A module global is not narrowed by an `if`, because other code — another function, a later frame — could change it between the test and the use; to handle a global’s optional, use `??`, or copy it into a local first and narrow the local. Assigning a new optional value to a narrowed variable inside the block *un-narrows* it: the guarantee held only for the value that was tested. And narrowing does not thread through `and/or` chains in v0.3, which is why the compiler asks you to parenthesise combined conditions (Chapter 3); when in doubt, narrow with a nested `if` or copy to a local. Each of these is a known v0.3 boundary, listed in Chapter 8, not a bug.

## 4.5 What returns an optional

Optionals are not something you sprinkle on; they arrive from the operations that can genuinely fail, and the compiler propagates them from there. The sources in v0.3 are:

| Expression                   | Type              | nil when                                                |
|------------------------------|-------------------|---------------------------------------------------------|
| <code>lt.load_save(k)</code> | <code>str?</code> | no readable save                                        |
| <code>num(s)</code>          | <code>num?</code> | <code>s</code> is not a number (incl. <code>""</code> ) |
| <code>pop(xs)</code>         | <code>T?</code>   | the list is empty                                       |
| <code>m[key]</code>          | <code>T?</code>   | the key is absent                                       |
| <code>env(name)</code>       | <code>str?</code> | the variable is unset                                   |

Each is a place where, in another language, you would get a sentinel you might forget to check: an empty string, a false, a `nil`, a special “not found” value. `wick` makes the failure a type, so forgetting to check it is not possible — the forgotten check is a compile error. Map lookup is the purest example: `m[key]` is *always* `T?`, even when you are sure the key is present, because the type system cannot know that you are sure. The missing-key case is forced into view at the call site, where `??` disposes of it in three characters:

```
let hp = stats["health"] ?? 100
```

## 4.6 Optionals in the type system

A few rules tie optionals into the rest of the language; they were stated in Chapter 3 and are worth collecting here from the optional’s point of view.

- **Widening is free.** A `T` may be used wherever a `T?` is wanted; going from “definitely present” to “possibly absent” loses no safety.
- **Unwrapping is required.** A `T?` may not be used where a `T` is wanted. `??` and narrowing are the only ways across, and both make you handle `nil`.
- **Optionals do not nest.** There is no `T??`. An optional is a `T` or `nil`, full stop.
- **`nil` needs a type.** A bare `nil` has no inferable type; `let x: str? = nil` supplies one. `nil` is meaningful only as the empty case of some specific optional.
- **Comparing a non-optional to `nil` is an error.** If `x` is a plain `str`, then `x != nil` is dead code — `x` can never be `nil` — and the compiler says `comparing non-optional to nil` rather than let you write a check that can never fail.

## 4.7 The discipline in practice

In day-to-day wick you stop writing the defensive scaffolding that a dynamic language demands, because the compiler tracks what the scaffolding was guarding. You do not wrap every load and parse in an `if x ~= nil` chain; you write `??` once where the value enters, and from there on the value is a plain `T` that cannot betray you. The optional is not overhead. It is the removal of overhead — the manual, forgettable, review-it-and-still-miss-it overhead of remembering which values might be `nil` — replaced by a compiler that remembers for you and refuses to forget.

The zero-byte save is one bug. The class it belongs to — a fallible operation whose failure is representable as an ordinary value and therefore skippable — is enormous, and it is the class wick was built to close. Every `T?` in a wick program is a small proof that one member of that class has been handled.



# Collections

wick has two container types, the list and the map, and one idiom — parallel lists — that stands in for the structs it does not yet have. This chapter covers all three, along with the iteration patterns that go with 0-based indexing and the deliberate choice to make every fallible lookup an optional.

## 5.1 Lists

A list is a typed, ordered, 0-based sequence. Its element type is one of the three scalar types; a list literal infers its element type from its contents, and an empty literal must be annotated.

```
let xs = [1, 2, 3]           // list<num>, element type inferred
let names = ["a", "b"]      // list<str>
let flags: list<bool> = []  // empty literal REQUIRES the
                             annotation
```

The annotation on the empty list is not optional politeness; there is nothing in `[]` to infer from, and the compiler says so: **empty [] needs a type**: `let xs: list<num> = []`. A literal with mixed element types does not compile either — lists are homogeneous — and reports **list elements must share one type**, so `[1, "a"]` is rejected.

**Operating on lists.** The core operations are built-ins, dispatched by the static element type:

```
push(xs, 4)                  // append; type-checked against the
                             element type
let last = pop(xs) ?? 0      // remove and return the last; T?
                             (nil if empty)
xs[0] = 10                   // index assignment
let head = xs[0]             // index read
let n = len(xs)              // number of elements
```

`push` appends a value that must match the element type — `push(flags, 3)` does not compile. `pop` removes and returns the last element, but a list may be empty, so its result is an optional `T?`, `nil` when there is nothing to pop; you handle it like any optional, usually with `?? default`. `len` returns the element count as a `num`.

**Indexing is checked at run time.** Reading or writing an index outside `[0, len)` is a *runtime* error, not undefined behaviour and not a silent `nil`. It stops the frame and shows on the error screen with the line number and the message `list index N out of range (len M)`. The static type system cannot know the value of an index in general, so this one check remains dynamic — but it is a check, with a precise message, not a memory-safety hole.

**Element types.** List elements are `num`, `bool`, or `str`. There are no lists of lists in v0.3: `list<list<num>>` reports `container elements must be num, bool, or str`. The workaround, when you genuinely need a grid or a jagged structure, is index arithmetic on a flat list, shown at the end of this chapter.

## 5.2 Maps

A map is a string-keyed, value-typed collection. Keys are always `str`; values are one of the three scalar types.

```
let scores = ["ana": 3, "bo": 5]    // map<num>
scores["cid"] = 1                  // insert or update
let s = scores["dee"] ?? 0         // get is ALWAYS T? --
    missing key is nil
let n = len(scores)                // number of entries
```

A map literal pairs string-literal keys with values; the keys in a literal must be string literals, not computed expressions (**map keys must be str literals**). To insert under a computed key, use index assignment: `m[k] = v`. The values in a literal must share one type, just as list elements do.

**Lookup is always optional.** The single most important fact about maps is that `m[key]` has type `T?`, always, even for a key you just inserted. The type system cannot prove a key is present, so it treats every lookup as possibly missing and hands you a `T?`. This is deliberate: the missing-key case — the one everybody forgets — is forced into view at the call site,

where ?? handles it in three characters. There is no it was there a moment ago” surprise, because the type never let you assume it.

```
scores["ana"] = 10
let a = scores["ana"] ?? 0    // still T?; the ?? is required,
                             not optional
```

## 5.3 Records and lists of records

A lamp, register, or circuit component is one logical value. Since Wick 0.2, a flat record gives that value named, checked fields:

```
record Lamp { x: num, z: num, lit: bool }
let lamps: list<Lamp> = []
push(lamps, Lamp { x: 3.4, z: 3.4, lit: true })
push(lamps, Lamp { x: -3.4, z: 3.4, lit: true })
lamps[0].lit = false
check(lamps[1].x == -3.4, "named field")
fn relight(i: num) { lamps[i].lit = true }
```

Declare the record before use. Constructors name every field; missing, duplicate, unknown, or wrongly typed fields are errors. Fields are limited to `num`, `bool`, and `str`; records have no methods or nested containers. Records are heap objects: assigning one to another variable shares it, so a field mutation is visible through either reference. Construct a new value when an independent copy is wanted.

The older tutorial uses parallel scalar lists to remain comparable with its original game. That representation still works, but records remove the need to keep separate field lists at the same length. Nested data still needs flat IDs or separate lists. Chapter 9 uses a record for an eight-bit register.

## 5.4 Iteration patterns

There is one loop form for collections: iterate the indices, and index in.

```
for i in 0..len(xs) {
  use(xs[i])
}
```

Because the range is end-exclusive and lists are 0-based, `0..len(xs)` visits exactly the valid indices, with no off-by-one. There is no `for x in list` in v0.3; the index form is the whole story, and it has the advantage that

the index `i` is in hand when you need it — for parallel lists you need it always, and for finding-the-nearest problems you need it to return.

**Search that returns an index.** A common shape is find the element matching some criterion, or report that there is none.” With no multiple returns and no `for x in list`, you write a loop that carries a best-so-far index and a sentinel for none”:

```
fn nearest_unlit(): num {           // returns a lamp index, or -1
    let bi = 0 - 1
    let bd = 0.0
    for i in 0..len(lamp_x) {
        if not lamp_lit[i] {
            let dx = lamp_x[i] - px
            let dz = lamp_z[i] - pz
            let d = dx * dx + dz * dz
            if bi < 0 or d < bd {
                bi = i
                bd = d
            }
        }
    }
    return bi
}
```

The caller checks the sentinel before using the result:

```
let i = nearest_unlit()
if i >= 0 {
    // i is a valid index
}
```

Note that `bi` is written `0 - 1` rather than `-1` in the initialiser; either is legal — unary minus exists — and the game code happens to spell it as a subtraction. The `-1` sentinel is a plain `num`, not an optional, because it is cheaper to test `i >= 0` than to thread a `num?` through the caller; this is the idiomatic stand-in for the multiple-return the language lacks.

## 5.5 Nested data without nested containers

When you truly need a two-dimensional structure — a tile grid, say — and `list<list<num>>` is unavailable, flatten it: store row-major in a single list and compute the index.

```
let W = 16
let H = 12
let tiles: list<num> = []
for i in 0..W * H { push(tiles, 0) }

fn get(tx: num, ty: num): num {
    return tiles[ty * W + tx]
}

fn set(tx: num, ty: num, v: num) {
    tiles[ty * W + tx] = v
}
```

This is the same trick you would use in a systems language over a raw buffer, and it composes fine with the runtime bounds check: an out-of-range `ty * W + tx` is caught exactly as any other out-of-range index. It is more verbose than a nested container would be, and it is the honest v0.3 answer until nested containers arrive.



# The Engine Interface

A wick program talks to the lantern engine through the `lt` namespace. Every `lt.*` call is registered with a typed signature that the compiler checks: wrong arity or a wrong argument type is a compile error naming the call and the argument position, exactly as for your own functions. This chapter describes the frame contract that structures every game, documents the API by area, and then covers the two properties — determinism and verifiability — that the engine and language were co-designed to provide.

## 6.1 The frame contract

The engine drives the game by calling two functions it looks up by name, once each per frame, at 60 frames per second:

```
fn update(dt: num) { } // simulation; dt is seconds since the
                        last frame
fn draw() { }          // rendering; 3D calls first, 2D
                        composites on top
```

`update(dt)` advances the simulation. Its argument `dt` is the elapsed time in seconds, *capped at 0.1* so that a long stall (a breakpoint, a slow load) cannot produce a single enormous step that tunnels the player through a wall. `draw()` renders the frame; 3D calls are issued first and the 2D calls composite on top, so a HUD drawn with `lt.print` and `lt.rect` always sits over the scene.

Both functions are optional. A `main.wick` with neither is legal — it runs its top-level statements once and then presents a static frame. Top-level statements, everything not inside a `fn`, run exactly once at load, and that is where a game creates its meshes, loads its textures and sounds, and initialises its state. The order is fixed and worth holding in mind: top-level once, then `update` and `draw` every frame, then collection at the frame boundary (Chapter 7).

## 6.2 Frame and screen

| Call                                       | Notes                                                                |
|--------------------------------------------|----------------------------------------------------------------------|
| <code>lt.W</code> <code>lt.H</code>        | compile-time constants: 400, 240                                     |
| <code>lt.clear(r, g, b)</code>             | clear colour and depth; call first in <code>draw()</code>            |
| <code>lt.time(): num</code>                | seconds since boot (fixed-step under <code>LANTERN_FIXED_DT</code> ) |
| <code>lt.screenshot(path: str)</code>      | save the 400×240 frame as BMP                                        |
| <code>lt.quit()</code>                     | request a clean exit                                                 |
| <code>lt.escape_quits(enable: bool)</code> | Escape quits by default; disable for pause menus                     |

`lt.W` and `lt.H` are numeric constants, not calls — they resolve at compile time to 400 and 240 — so `lt.W / 2` is a constant expression you can use freely for layout. The screen is always 400 by 240; the engine scales the presented image to the window.

## 6.3 The 3D scene

| Call                                                       | Notes                                                       |
|------------------------------------------------------------|-------------------------------------------------------------|
| <code>lt.camera(ex,ey,ez, tx,ty,tz, fov=55)</code>         | eye position and look-at target                             |
| <code>lt.light(dx,dy,dz, ambient=0.35)</code>              | the directional light                                       |
| <code>lt.point_light(i, x,y,z, radius, r=1,g=1,b=1)</code> | slots 0–3; <code>radius &lt;= 0</code> turns the slot off   |
| <code>lt.fog(start, end, r,g,b)</code>                     | linear by view depth; <code>end &lt;= start</code> disables |

Trailing parameters with defaults — `fov`, `ambient` — may be omitted at the call site; the compiler fills the default in, so a native always receives full arity. There are four point-light slots, indexed 0 through 3; setting a slot’s radius to zero or below disables it, which is how Lantern Night extinguishes a lamp:

```
if lamp_lit[i] {
    lt.point_light(i, lamp_x[i], 0.8, lamp_z[i], 3.6, 1.0, 0.70,
        0.28)
} else {
    lt.point_light(i, 0, 0, 0, 0, 0, 0, 0) // radius 0: off
}
```

## 6.4 Meshes

| Call                                                              | Notes                                                |
|-------------------------------------------------------------------|------------------------------------------------------|
| <code>lt.cube(): num</code>                                       | unit cube                                            |
| <code>lt.plane(segs=1): num</code>                                | unit XZ plane, tessellated                           |
| <code>lt.sphere(seg=12),</code>                                   | unit primitives                                      |
| <code>lt.cylinder(seg=16),</code>                                 |                                                      |
| <code>lt.cone(seg=16)</code>                                      |                                                      |
| <code>lt.mesh(verts: list&lt;num&gt;): num</code>                 | custom mesh, 12 floats per vertex                    |
| <code>lt.load_mesh(path: str): num</code>                         | Wavefront OBJ (flat-normal fallback)                 |
| <code>lt.draw(m, x,y,z, rx,ry,rz, sx,sy,sz, r,g,b, tex=-1)</code> | gouraud-lit draw (rotations, scales, colour default) |
| <code>lt.draw_lerp(a, b, t, x,y,z, ..., tex=-1)</code>            | keyframe tween between two same-count meshes         |
| <code>lt.billboard(tex, x,y,z, w,h, u0,v0,u1,v1)</code>           | camera-facing quad (sprite characters)               |
| <code>lt.shadow(x,y,z, radius, alpha=0.35)</code>                 | grounding blob                                       |

Every mesh constructor returns a `num` handle; there is no mesh type, because handles are opaque integers the engine owns. `lt.mesh` takes a flat `list<num>` of twelve floats per vertex — position (3), normal (3), texture coordinates (2), and colour (4) — which is a natural fit for the flat-list discipline of Chapter 5. `lt.draw` has a long tail of defaulted parameters, so the common case stays short:

```
let ground = lt.plane(12)
lt.draw(ground, 0, 0, 0, 0, 0, 0, 15, 1, 15, 0.42, 0.40, 0.40)
```

## 6.5 2D

| Call                                                                     | Notes                                             |
|--------------------------------------------------------------------------|---------------------------------------------------|
| <code>lt.rect(x,y,w,h, r,g,b, a=1)</code>                                | filled, alpha-blended                             |
| <code>lt.load_texture(path: str): num</code>                             | BMP; magenta (255,0,255) is transparent           |
| <code>lt.sprite(tex, x,y, sx=1,sy=1)</code>                              | top-left anchored                                 |
| <code>lt.sprite_ex(tex, cx,cy, sx=1,sy=1, rot=0, r=1,g=1,b=1,a=1)</code> | centre anchor, rotate, tint; negative scale flips |
| <code>lt.sprite_uv(tex, x,y,w,h, u0,v0,u1,v1)</code>                     | atlas sub-rect (tilemaps)                         |
| <code>lt.print(text: str, x,y, r=1,g=1,b=1,a=1)</code>                   | built-in 8×8 font, <code>\n</code> aware          |

`lt.print` renders in a built-in 8-by-8 pixel font and understands `\n`. Because `+` concatenates only `str` with `str`, building a HUD line means converting numbers explicitly with `str` — which, unlike Lua’s `string.format`, prints integers cleanly with no trailing decimal:

```
lt.print("LAMPS RELIT: " + str(score), 4, 3, 1, 1, 1, 0.95)
```

## 6.6 Audio, input, and saves

| Call                                               | Notes                                                    |
|----------------------------------------------------|----------------------------------------------------------|
| <code>lt.load_sound(path: str): num</code>         | WAV                                                      |
| <code>lt.play(sound, volume=1, loop=0): num</code> | returns a channel, or <code>-1</code> if all 16 are busy |
| <code>lt.stop(channel)</code>                      |                                                          |
| <code>lt.volume(v)</code>                          |                                                          |
| <code>lt.key(name: str): bool</code>               | held; keyboard and gamepad merged                        |
| <code>lt.pressed(name: str): bool</code>           | went down this frame                                     |
| <code>lt.gamepad(): bool</code>                    |                                                          |
| <code>lt.rumble(low, high, ms)</code>              |                                                          |
| <code>lt.save(name: str, data: str): bool</code>   | binary-safe                                              |
| <code>lt.load_save(name: str): str?</code>         | <b>optional</b> — <code>nil</code> when unreadable       |

`lt.key` is true while a key is held; `lt.pressed` is true only on the frame

the key goes down — the difference between continuous movement and a one-shot action. The recognised input names are:

```
left right up down z x c space return escape a s d
w
```

Keyboard and gamepad are merged onto these names: the pad’s d-pad and stick map to the directions, and its face buttons map A to z, B to x, Y to c, and Start to `return`. So a game written against `lt.key("z")` works with either input with no extra code.

The one call in this table that returns an optional is `lt.load_save`, and it is no accident that the chapter which motivates the whole language turns on it. Its signature is `load_save(str): str?`; the compiler will not let you use the result until you have handled the `nil`, which is the entire point of Chapter 4.

## 6.7 Determinism

wick and lantern are built so that the same program with the same inputs produces the same frames on every machine. Two facilities make this true, and a testing culture is built on top of them.

**Deterministic randomness.** The built-in `rand()` returns a uniform value in  $[0, 1)$  from a fixed xorshift64\* generator — the *same sequence on every machine*. There is no time-seeded mode. If you want variation between runs, you seed it yourself with `srand(seed)`, using a number you chose and can log:

```
srand(7)                                // same layout every run, on every
    machine
let stars: list<num> = []
for i in 0..40 {
    push(stars, rand() * 400)
    push(stars, rand() * 240)
}
```

Because the sequence is fixed, a seeded run replays identically, which is what makes screenshot tests meaningful — the random” starfield is the same starfield in CI as on your desk.

**Fixed-step time.** `lt.time()` returns seconds since boot, and under the environment variable `LANTERN_FIXED_DT` the frame clock advances by a fixed step regardless of wall-clock speed. Combined with deterministic `rand()`, this makes an entire gameplay session reproducible frame for frame.

**Screenshot capture.** Setting `LANTERN_SHOT=/tmp/x` tells the engine to capture frame 60 as a BMP and exit. With `LANTERN_FIXED_DT=1` alongside it, that capture is byte-identical everywhere. The two together turn does it still look right” into a mechanical check:

```
LANTERN_SHOT=/tmp/x LANTERN_FIXED_DT=1 ./build/lantern
  games/mygame
```

## 6.8 A culture of verification

Determinism is only useful if the game can drive and check itself, and wick gives two built-ins for exactly that.

**Self-play through env.** `env(name): str?` reads an environment variable — an optional, `nil` when unset. A game can expose its own self-play switch and branch on it, so CI runs the game on autopilot with a fixed seed:

```
let AUTO = env("SHOWCASE_AUTO") != nil    // deterministic
      self-play (CI)
if AUTO { srand(7) }
```

In Lantern Night, `AUTO` drives the player toward the nearest dark lamp instead of reading the keyboard, so a headless CI run plays a real, deterministic round and the frame-60 screenshot is a true regression test.

**Assertions with check.** `check(cond: bool, msg: str)` is a runtime assertion: when `cond` is false it stops the frame and shows `check failed: <msg>` on the error screen. The wick test suite is built from these — a scene that sets up a known state and `checks` the results, run under fixed-step time so the outcome is exact:

```
check(len(lamp_x) == 4, "expected four lamps")
check(nearest_unlit() == -1, "all lamps should start lit")
```

Together, `env`-driven self-play, deterministic `rand`, fixed-step time, and `check` turn a game into something that tests itself the same way on every machine — the practical payoff of designing the language and the engine for determinism from the start.

## Chapter 7

# Internals

This chapter opens the implementation. You do not need any of it to write wick games, but a language’s design is best understood through its machinery, and wick’s machinery is small enough to explain honestly in a chapter. The whole language — lexer, compiler, virtual machine, garbage collector, built-ins — is about two thousand lines of dependency-free C++ in three files:

---

|                                  |                                                         |
|----------------------------------|---------------------------------------------------------|
| <code>wick/wick.hpp</code>       | the embed API (about 120 lines)                         |
| <code>wick/wick_front.cpp</code> | lexer and one-pass typed compiler →<br>bytecode         |
| <code>wick/wick_vm.cpp</code>    | stack VM, mark/sweep GC, built-ins,<br>signature parser |

---

## 7.1 One-pass typed compilation

The defining choice is that **there is no abstract syntax tree**. The compiler is single-pass, in the style of Crafting Interpreters’ *clox*: a Pratt expression parser emits bytecode as it goes, and it carries a static **Type** for every expression as it emits. The type checker and the code generator are not two passes over a tree; they are the same walk. When the parser finishes an expression, it has both produced the bytecode that computes the value and determined the value’s type, together, in one sweep.

Two visible language rules fall directly out of this design.

**Declare before use.** Because the compiler resolves each name and each call at the moment it reaches it — there is no later pass to fix up forward references — a function must be defined textually before it is called, and a variable must be declared before it is used. This is not a limitation the designers chose for its own sake; it is what single-pass compilation costs, and the price is small.

**Types are known at emit time.** Since every expression’s type is in hand as its code is generated, the compiler can *specialise* the bytecode. This is the performance story, and it is worth being concrete about.

## 7.2 Specialised opcodes

In a dynamically typed VM, `a + b` compiles to a single generic `add`” opcode that, at run time, inspects the tags of its operands and decides whether to add numbers, concatenate strings, or raise an error. That tag check happens on every execution of the instruction. `wick` pays it *once*, at compile time, because it already knows the types:

- `+` on two `num` emits `OP_ADD`; `+` on two `str` emits `OP_CONCAT`. There is never a runtime tag check to choose between them — the choice was made when the code was generated.
- The ordered comparisons compile to numeric compare opcodes (`OP_LT`, `OP_LE`, `OP_GT`, `OP_GE`), because their operands are known to be `num`.
- Local variables resolve to *frame slots* — small integer offsets into the current call’s stack frame — so `OP_LOADL` and `OP_STOREL` take a slot number, and there is no name lookup at run time. Globals resolve to slot indices too, via `OP_LOADG` and `OP_STOREG`.
- Engine natives resolve to integer ids at compile time. The VM’s native call, `OP_NCALL`, is an array index into the natives table, not a hash-table walk by name.

The bytecode has roughly forty opcodes. Alongside the arithmetic and comparison ops there are the control-flow jumps (`OP_JMP`, `OP_JF` which pops a `bool`, `OP_JB` for backward jumps in loops), two jumps that implement optionals (`OP_JNN`, jump if not nil”, and `OP_ISNIL`), the call ops (`OP_CALL` for wick functions, `OP_NCALL` for natives), the container ops (`OP_LIST`, `OP_MAP`, and the index get/set pairs `OP_IGET/OP_ISET` and `OP_MGET/OP_MSET`), and the specialised built-ins (`OP_LEN`, `OP_TOSTR`, `OP_TONUM`, `OP_LPUSH`, `OP_LPOP`). Each corresponds to something the type checker proved about the code around it.

## 7.3 The stack VM

The runtime is a classic stack machine. There is a constants pool, a stack of values, and a stack of call frames; each frame records its base — the point in the value stack where its slots begin — and its instruction pointer. Arguments to a call are already on the stack when the frame is pushed, and the callee’s locals live in the slots just above them. Returning resizes the stack back to the frame’s base, so the stack is empty between top-level statements and, crucially, empty between frames.

Runtime errors carry a bytecode-offset-to-line table, built during compilation, so that when the VM raises an error — an out-of-range index, a failed

**check** — it can report the source line, and the engine’s error screen can point at it. This is why a wick runtime error reads `file:line: message` and not a raw bytecode offset.

There is **no JIT**, and this is deliberate, not unfinished. The interpreter is fast enough for game logic at 400 by 240 — the work per frame is small, and the specialised opcodes remove the overhead a naive interpreter would carry. More importantly, the engine targets platforms, iOS chief among them, where generating and executing code at run time is forbidden. A JIT would make wick illegal on those platforms; the specialised-bytecode interpreter is legal everywhere by construction, and its performance is predictable, with no warm-up and no deoptimisation cliffs.

## 7.4 Frame-boundary garbage collection

wick manages three kinds of heap object — strings, lists, and maps — with a mark-and-sweep collector. What makes it a game collector rather than a general one is *when* it runs: never on its own initiative, only when the host calls `wick::collect()`. lantern calls that function exactly once per frame, after presenting the image.

The consequence is that a collection can never land in the middle of a draw. The one moment a pause is invisible — after the frame is on screen, before the next one begins — is the only moment collection happens. The worst-case cost of a collection is bounded by how much garbage a single frame produced, which for a game loop that reuses its state is modest. There is no unpredictable, generation-triggered pause of the kind that drops a frame in a Lua game.

The collector’s roots are simple, and they are simple *because* the language is small. The mark phase starts from the constants pool, the globals, and the VM value stack, and follows the references inside lists and maps. There are no closures and no upvalues in v0.3, so there are no captured environments to trace; the object graph is shallow. And because the value stack is empty between frames, at the moment collection runs the only live roots are the globals and the constants — exactly the long-lived state — so the mark phase has little to do.

## 7.5 Embedding wick

wick is meant to be embedded. The public surface is `wick.hpp` alone; embedders never touch the internal headers. The lifecycle is create, register the environment, load a program, call its functions each frame, collect at your safe point, and reset for hot reload.

---

```

#include "wick.hpp"

wick::VM* vm = wick::create();
std::string err;

// typed natives: a signature DSL the compiler enforces
wick::addNative(vm, "game", "spawn(num, num, str): num",
    [](wick::VM& vm, const wick::Value* a, int) {
        int id = mySpawn(a[0].d, a[1].d, wick::getStr(a[2]));
        return wick::Value::num(id);
    }, err);
wick::addConst(vm, "game", "MAX", 64);

wick::load(vm, source, "main.wick", err); // compile + run top
level
wick::call(vm, "update", dt, true, err); // each frame
wick::collect(vm); // at YOUR safe point
wick::reset(vm); // hot-reload support

```

**The signature DSL.** A native is registered with a signature string that the compiler parses once and then enforces at every call site. The grammar is:

```
name(type[, type][, type = default]) [: type]
```

where a type is `num`, `bool`, `str`, or `list`, with a trailing `?` for an optional. This one string is how `lt.load_save(str): str?` teaches the compiler that loading a save returns an optional — the entire optional discipline of Chapter 4, for an engine call, is carried by two characters in a signature. Defaults, which must be numeric literals, make trailing parameters optional at the call site; the compiler fills them in, so the native always receives full arity and never has to check `argc`. Registering a native under a namespace (`"lt"`, `"game"`) qualifies its name; passing a null namespace puts it in the global scope, which is how the built-ins like `floor` and `sqrt` are installed.

**Reporting errors from a native.** A native signals failure by calling `wick::setError(vm, msg)`. The VM turns that into a `file:line` runtime error pointing at the call site, so a native's failure is reported the same way as any other runtime error and lands on the same error screen. The value helpers — `makeStr`, `getStr`, `listLen`, `listGet` — cover reading and building the heap values a native receives and returns.

**Hot reload.** `wick::reset(vm)` clears all program state — globals, functions, the compiled protos — while leaving the registered environment (the

natives and constants) intact. That is exactly what hot reload needs: tear down the program, keep the bindings, load the new source. Because **reset** frees the old program's resources before the new one loads, reloading never leaks — the property the tutorial promised.

None of this is large. The embed API is about 120 lines, and the whole implementation is around two thousand. That smallness is the recurring theme of this book, and it is nowhere more visible than here: a statically typed, optional-checked, deterministic scripting language, with a garbage collector and a typed foreign-function interface, that you can read in an afternoon and own completely.



# Design and Trade-offs

A language is a set of decisions, and decisions have costs as well as benefits. This chapter argues for wick's decisions honestly: it puts the wick and Lua builds of the same real game side by side, it says plainly where Lua still reads better, and it explains the roadmap philosophy that makes v0.3's omissions deliberate rather than embarrassing.

## 8.1 The same game, twice

Lantern Night is the proof. It ships in both languages, as two deliberately separate projects — `games/showcase` in Lua and `games/showcase_wick` in wick — kept side by side so the two languages stay comparable on the same real game. Read the pair like a Rosetta stone. Three excerpts carry the argument.

### 8.1.1 The hi-score load

This is the bug from Chapter 4, in its natural habitat. The Lua build shipped with a frame-1 crash:

```
-- Lua: a 0-byte save makes load_save return "" (truthy!), the  
-- fallback never fires, and tonumber("") = nil poisons  
everything.  
local best = tonumber(lt.load_save("lanternnight_best") or "0")
```

The wick build cannot express that bug. The line below is the only version that compiles, because both ways the value can be absent — no file, empty file — are forced to be handled:

```
// wick: the ONLY version the compiler accepts.  
let best = num(lt.load_save("lanternnight_wick_best") ?? "") ?? 0
```

Here wick wins outright. The safe code is shorter than the buggy Lua code, and the unsafe code does not compile.

### 8.1.2 Records earned their place

Wick 0.1 represented lamps and props with parallel scalar lists. That kept the compiler small but scattered each logical object across several lists. KORA’s kitchen made the cost concrete: five lists had to remain in step. Wick 0.2 therefore admitted flat records; Chapter 5 explains their checked fields and list storage. Lua still supports richer nested tables and methods; Wick intentionally does not claim that whole surface.

Wick 0.3 follows the same rule. The processor-building project needs masks, shifts, register wrapping, and readable bus values. Bit Lab exercises those operations directly; Chapter 9 documents the result. The new functions use the existing typed-native mechanism without changing expression precedence.

### 8.1.3 Multiple returns

A smaller loss. Lua returns the nearest lamp and its distance together:

```
local i, dist2 = nearest_unlit()
```

wick has no multiple returns, so the two queries are split, and no lamp” becomes a sentinel index rather than an absent second value:

```
let i = nearest_unlit()      // returns the index, or -1
let d = dist2_to(i)         // second query, explicit
```

This is more verbose, but it is not worse in kind — the split is mechanical, and the sentinel is a well-understood idiom. Multiple returns are on the roadmap; their absence is an inconvenience, not a hazard.

## 8.2 What you stop writing

Set against those losses is a real reduction in the defensive code a dynamic language demands. Porting Lantern Night to wick, the following simply left the source:

- The `if x ~= nil` chains around every save and parse. The compiler tracks nullability, so you write `??` once at the boundary and the value is a plain `T` thereafter.
- The `math.` prefixes. `sin`, `floor`, `rand`, and their kin are built-ins with no namespace.
- The `string.format` calls that existed only to print an integer without a trailing decimal. `str(n)` prints integers cleanly.

- The argument-order paranoia on engine calls. Arity and types are checked at compile time with the call name and the argument number in the message, so a mistake is caught the moment it is written, not discovered by a wrong-looking frame.

### 8.3 What stays the same

It is worth being clear about what the language change did *not* disturb, because it bounds the cost of adopting wick. The engine API is call-for-call identical: `lt.draw(...)` is the same line in both builds. The frame contract — `update(dt)` and `draw()` — is the same. Hot reload and the error screen behave the same. And the two games render the same frames, which is not a figure of speech: the deterministic screenshot tests of Chapter 6 hold the wick and Lua builds to the same pixels. wick changed the language, not the engine and not the workflow.

### 8.4 The v0.3 limits

wick is deliberately small — Lua-1.0 small. Every omission below is a known gap with a workaround today and a place on the roadmap, not a surprise waiting to be discovered. The bar for v0.3 was fixed and narrow: run real lantern games, kill the Lua bug classes, and stay around two thousand lines.

| Not in v0.3                                                       | Workaround today                                                           |
|-------------------------------------------------------------------|----------------------------------------------------------------------------|
| Closures, nested and first-class functions                        | top-level <code>fn</code> plus module globals                              |
| Multiple return values                                            | two functions, or out-of-band state                                        |
| Nested containers<br>( <code>list&lt;list&lt;num&gt;&gt;</code> ) | index arithmetic on a flat list                                            |
| Modules / imports (one <code>main.wick</code> )                   | it is a game script, not an app platform — yet                             |
| <code>for x in list</code> iteration                              | <code>for i in 0..len(xs)</code>                                           |
| String indexing / slicing / methods                               | <code>len</code> , <code>+</code> , <code>str()</code> cover the HUD cases |
| Exhaustive return-path checking                                   | a typed function that falls off the end returns <code>nil</code> — don't   |
| Narrowing for globals / <code>and</code> -chains                  | ??, or copy to a local first                                               |
| Exponent number literals                                          | decimal; hex/binary are supported                                          |

Two entries deserve to be pulled out, because they are not limits at all but choices, and they are choices the roadmap will not reverse:

**No JIT.** This is iOS-legal by construction and gives deterministic performance with no warm-up. It is a feature, permanent.

**No time-seeded randomness.** Determinism is the default so that replays and CI screenshots work; if you want variety between runs you seed `srand()` yourself, with a number you can log. Also permanent.

## 8.5 Roadmap philosophy

The rule that governs what happens next deserves stating as law, because it is the core thing this language follows: *wick contains only what we truly need — nothing for the sake of having it*. A feature is admitted when real game code demonstrably hurt without it, and not before. The paired Lantern Night builds are the standing evidence mechanism: missing-feature pain must show up in a real game or focused example before the language grows. Optionals are in this book because a shipped crash demanded them; closures are not, because no shipped program has yet paid a price for their absence. Feature parity with other languages is explicitly not a reason to add anything.

The organising idea is Lua 1.0 small.” Lua did not begin with metatables, coroutines, and an ecosystem; it began as a small, clean, embeddable core and grew as real use justified each addition. wick is taking the same path deliberately. Every omission in the table above is addable behind the same typed front end that already exists — closures and multiple returns do not require rethinking the compiler, only extending it — and each will be added when a real game needs it enough to justify the size, not before.

This is why the limits are stated so plainly. A language that hid its gaps would be inviting you to discover them at the worst moment. wick lists them, gives each a workaround, and puts each on a roadmap, because the whole thesis of the language is that a program’s failure modes should be visible up front rather than discovered in production — and a language owes its users the same honesty it enforces in their code.

## Chapter 9

# Bits, Bytes, and Buses

Wick 0.3 grew from a concrete next project: a game in which the player builds a processor, using the Intel 8080 as the target. A language for that game needs to make bits visible and register widths explicit. This chapter builds the small pieces; it does not implement a full CPU or its timing.

## 9.1 Write the bits you mean

```
let opcode = 0x76
let sign_mask = 0b10000000
let last_address = 0xFFFF
check(opcode == 118, "hexadecimal")
check(sign_mask == 128, "binary")
```

Hexadecimal and binary prefixes accept either case. Digits must be valid for the base and the whole value must fit in 32 unsigned bits. A bare prefix, an underscore, a fractional prefixed literal, or a value above 0xFFFFFFFF is a compile error. Decimal numbers keep their existing rules. All these values still have type `num`; no integer type was added.

## 9.2 Gates and masks

The six bit functions are `bit_and`, `bit_or`, `bit_xor`, `bit_not`, `bit_shl`, and `bit_shr`. The first three combine two values. NOT complements all 32 bits. Left shift discards bits beyond bit 31; right shift fills with zero. Operands must be whole numbers in 0 through 4294967295. Shift counts must be whole numbers in 0 through 31.

```
let wires = 0b10100101
check(bit_and(wires, 0x0F) == 5, "low nibble")
check(bit_xor(wires, 0x80) == 0x25, "toggle bit 7")
check(bit_shr(wires, 7) == 1, "read high bit")
check(bit_shl(1, 31) == 0x80000000, "high bit")
```

Type and argument-count mistakes are compile errors. Numeric domain errors, such as a fractional operand or a shift count of 32, are runtime errors reported at the Wick call site. The VM validates the value before any C++ integer conversion or shift. NaN and infinity are rejected too.

These are ordinary typed functions. Their addition changes neither operator precedence nor the meaning of `and`, `or`, and `not`, which still operate on booleans.

## 9.3 A register has a width

`u8(n)` wraps a safe integer modulo 256; `u16(n)` wraps it modulo 65536. Both accept negative values. Inputs must be finite whole numbers between -9007199254740991 and 9007199254740991 inclusive. Ordinary arithmetic remains double arithmetic: wrapping happens only where you request it.

```
record Register { value: num, name: str }
let a = Register { value: 0xFF, name: "A" }
let wide = a.value + 1
let carry = wide > 0xFF
a.value = u8(wide)
check(a.value == 0 and carry, "keep the carry")
check(u8(-1) == 255, "byte underflow")
check(u16(0xFFFF + 1) == 0, "PC rollover")
check(u8(bit_not(0x0F)) == 0xF0, "8-bit NOT")
```

Keep the wide result until the flags have been calculated. Wrapping first would erase the very carry the player needs to see. For an eight-bit ADD, zero is `result == 0`, sign tests mask `0x80`, and auxiliary carry tests whether the low-nibble sum exceeds `0x0F`. Parity counts the set bits of the wrapped result and is true for an even count.

## 9.4 Pairs and memory

A high byte and a low byte make a sixteen-bit address:

```
let h = 0x80
let l = 0x05
let address = bit_or(bit_shl(h, 8), l)
check(address == 0x8005, "register pair")
check(bit_shr(address, 8) == h, "high byte")
check(u8(address) == l, "low byte")
let memory: list<num> = []
for i in 0..65536 { push(memory, 0) }
```

```
memory[address] = 0x76
check(memory[0x8005] == 118, "memory write")
```

This list models the address space without a new container type. Its entries are still numbers, so write `u8(value)` at a byte-write boundary. It is not a packed byte buffer and makes no promise of hardware clock speed. The game can advance the processor by deliberate simulation steps independently of render frames.

## 9.5 Make the state readable

`hex(value, width=1)` returns uppercase digits without a prefix; `bin(value, width=1)` returns binary digits. Values use the unsigned 32-bit domain. Width is an integer minimum of 1–8 for hex and 1–32 for binary. A value is never truncated to fit a label.

```
check(hex(10, 2) == "0A", "byte label")
check(bin(5, 8) == "00000101", "bus label")
check(hex(256, 2) == "100", "overflow stays visible")
```

## 9.6 The workbench and the next game

Run `./build/lantern_games/bitlab` to see these operations in a small Wick workbench. Arrow keys select and edit the input registers; Z cycles ADD, AND, and XOR. Touch or click an input bit to toggle it. The flags follow 8080 ADD, ANA, and XRA behaviour, including ANA’s auxiliary-carry rule: it reflects bit 3 of either input, rather than the 8085 rule of always setting that flag. The reference is Intel’s *8080/8085 Assembly Language Programming* manual (1977), chapter 1, Auxiliary Carry Flag.”

Bit Lab is an example for the language release. Building the player’s circuit editor, progression, instruction decoder, and complete processor verification remains game work. No circuit simulator is hidden in the host; the example’s logic is Wick code.

**Upgrading.** The ten new built-in names reject same-named user function declarations at compile time. Rename a colliding user function. No imports, closures, nested records, string indexing, or new numeric types are introduced in this release.



## Appendix A

# Grammar

The following EBNF describes wick v0.3, derived from the syntax reference and the reference compiler's front end. Terminals are quoted; { } means zero or more repetitions, [ ] means optional, and | separates alternatives. Comments and whitespace are handled by the lexer and are not shown. The precedence of the expression rules, from `orExpr` (loosest) down to `postfix` (tightest), matches the table in Chapter 3.

```
program          = { toplevel } ;
toplevel         = recordDecl | fnDecl | statement ;
recordDecl       = "record" ident "{" field { "," field } "}" ;
field            = ident ":" scalar ;

fnDecl           = "fn" ident "(" [ params ] ")" [ ":" type ]
                 block ;
params           = param { "," param } ;
param            = ident ":" type ;
block            = "{" { statement } "}" ;

statement        = letStmt | assign | callStmt | ifStmt | whileStmt
                 | forStmt | "break" | "continue" | returnStmt ;
letStmt          = "let" ident [ ":" type ] "=" expr ;
assign           = lvalue "=" expr ;
lvalue           = ident [ "[" expr "]" ] [ "." ident ] ;
callStmt         = call ;
ifStmt           = "if" expr block { "elif" expr block } [ "else"
                 block ] ;
whileStmt        = "while" expr block ;
forStmt          = "for" ident "in" expr ".." expr block ;
returnStmt       = "return" [ expr ] ;

type             = element [ "?" ]
                 | "list" "<" element ">" [ "?" ]
                 | "map" "<" element ">" [ "?" ] ;
scalar           = "num" | "bool" | "str" ;
element          = scalar | ident ; (* ident: declared record
                 type *)
```

```

expr          = orExpr ;
orExpr        = andExpr { "or" andExpr } ;
andExpr       = coalesce { "and" coalesce } ;
coalesce      = equality { "??" equality } ;
equality      = comparison { ( "==" | "!=" ) comparison } ;
comparison    = addition { ( "<" | "<=" | ">" | ">=" )
    addition } ;
addition      = term { ( "+" | "-" ) term } ;
term          = unary { ( "*" | "/" | "%" ) unary } ;
unary         = ( "-" | "not" ) unary | postfix ;
postfix       = primary { "(" [ args ] ")" | "[" expr "]" | "."
    ident } ;
primary       = number | string | "true" | "false" | "nil"
    | qualified | ident | "(" expr ")"
    | listLit | mapLit | recordLit ;
recordLit     = ident "{" ident ":" expr { "," ident ":" expr }
    "}";
qualified     = ident "." ident ;          (* e.g. lt.draw *)
call          = ( ident | qualified ) "(" [ args ] ")" ;
args          = expr { "," expr } ;

listLit       = "[" [ expr { "," expr } ] "]" ;
mapLit        = "[" string ":" expr { "," string ":" expr } "]"
    ;

ident         = letter { letter | digit } ;    (* letter
    includes _ *)
number        = digit { digit } [ "." digit { digit } ]
    | ( "0x" | "0X" ) hexDigit { hexDigit }
    | ( "0b" | "0B" ) bit { bit } ;
hexDigit      = digit | "a".."f" | "A".."F" ;
bit           = "0" | "1" ;
string        = '"' { char | escape } '"' ;
escape        = "\n" | "\t" | '\\" | "\\\" ;

```

Notes on the grammar. An empty `listLit` (`[]`) is legal syntactically but requires a type annotation on its enclosing `let`, since the element type cannot be inferred. A `mapLit` requires string-literal keys. The `..` in `forStmt` is a single token; the lexer guarantees that `1..5` tokenises as `1`, `..`, `5`. Type names are contextual, recognised only in the `type` position, so they are not reserved words.

## Appendix B

# Compile-Error Catalogue

Every wick diagnostic has the form `file:line: message` and appears on the in-engine error screen. Fix the file and save; the game hot-reloads. The table below lists the compile-time messages by cause, with the meaning and the usual remedy. Message text in the reference compiler may vary in minor wording; the patterns here are canonical.

| Message pattern                                                                               | Meaning and fix                                                                                                                             |
|-----------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <code>expected str, got str?</code> (any types)                                               | An optional (or a wrong type) used where a plain value is needed. Unwrap with <code>??</code> or narrow with <code>if x != nil { }</code> . |
| <code>if condition must be bool</code><br>(wick has no truthiness)                            | <code>if x</code> on a number or string. Write the comparison: <code>if x &gt; 0</code> , <code>if s != ""</code> .                         |
| <code>while condition must be bool</code>                                                     | As above, for <code>while</code> .                                                                                                          |
| <code>unknown variable 'x'</code> --<br>wick has no implicit<br>globals; use <code>let</code> | Assigned a name never declared (often a typo). Add <code>let</code> , or fix the spelling.                                                  |
| <code>unknown variable 'x'</code><br>(declare it with <code>let</code> )                      | Read of an undeclared name. Declare it, or fix the spelling.                                                                                |
| <code>'+' needs num+num or<br/>str+str</code> (use <code>str(x)</code> )                      | Mixed-type <code>+</code> . Convert explicitly: <code>"n=" + str(n)</code> .                                                                |
| <code>'-' needs num operands</code><br>/<br>arithmetic needs num<br>operands                  | An arithmetic operator with a non-num operand.                                                                                              |
| <code>f() argument 3: expected<br/>num, got str</code>                                        | A typed call mismatch (engine or your own function). Check the signature.                                                                   |
| <code>f() needs at least N<br/>arguments</code> / <code>takes N<br/>arguments</code>          | Arity mismatch. See the signature.                                                                                                          |

*continued on next page*

| Message pattern                                                 | Meaning and fix                                                                    |
|-----------------------------------------------------------------|------------------------------------------------------------------------------------|
| unknown function<br>'f' (wick requires<br>declare-before-use)   | Called above its definition. Move the <b>fn</b> up.                                |
| can't infer a type from<br>nil; annotate: let x:<br>str? = nil  | A bare <b>nil</b> initialiser. Add the annotation.                                 |
| empty [] needs a type: let<br>xs: list<num> = []                | An untyped empty literal. Annotate the<br><b>let</b> .                             |
| list elements must share<br>one type                            | A heterogeneous literal such as [1, "a"].<br>Lists are homogeneous.                |
| list elements must be num,<br>bool, or str                      | A container element of a disallowed type.                                          |
| comparing non-optional to<br>nil                                | <b>x != nil</b> on a plain type; it can never be<br><b>nil</b> . Delete the check. |
| '??' left side must be an<br>optional                           | <b>a ?? b</b> where <b>a</b> can never be <b>nil</b> . Delete<br>the <b>??</b> .   |
| '??' default must be T                                          | The default's type does not match the op-<br>tional's base type.                   |
| 'x' already declared in<br>this scope / 'x' already<br>declared | A duplicate <b>let</b> . Assign instead, or rename.                                |
| parenthesize this<br>condition: (a != b) and<br>...             | A narrowing pattern mixed with <b>and/or</b> .<br>Add parentheses.                 |
| break outside a loop /<br>continue outside a loop               | A loop-control statement with no enclosing<br>loop.                                |
| fn declarations can't nest                                      | An <b>fn</b> inside another <b>fn</b> . Top level only<br>(v0.3).                  |
| return outside a function                                       | A <b>return</b> at top level.                                                      |
| void function can't return<br>a value                           | <b>return expr</b> in a function with no declared<br>return type.                  |
| this function must return T                                     | A bare <b>return</b> , or a type mismatch, in a<br>value-returning function.       |
| map keys must be str<br>literals                                | Computed keys in a map literal. Build with<br><b>m[k] = v</b> .                    |
| container elements must be<br>num, bool, or str                 | A nested container. Flatten it (Chapter 5).                                        |

*continued on next page*

| Message pattern                                 | Meaning and fix                             |
|-------------------------------------------------|---------------------------------------------|
| <code>only lists and maps can be indexed</code> | Indexing a value that is not a list or map. |
| <code>'and'/'or' needs bool operands</code>     | A logical operator with a non-bool operand. |
| <code>unexpected character 'x'</code>           | A byte the lexer does not recognise.        |

## Runtime errors

Static typing removes the compile-time classes above but cannot remove the following, which depend on values only known at run time. Each stops the frame and shows on the error screen with a line number.

| Message                                                   | Cause                                                |
|-----------------------------------------------------------|------------------------------------------------------|
| <code>list index N out of range (len M)</code>            | An out-of-bounds <code>xs[i]</code> , read or write. |
| <code>check failed: &lt;msg&gt;</code>                    | Your own <code>check()</code> assertion failed.      |
| <code>load_texture/mesh/sound failed: &lt;path&gt;</code> | A missing or malformed asset.                        |



## Appendix C

# Quick Reference

## Built-ins

Always available, with no namespace. The generic built-ins (`len`, `str`, `num`, `push`, `pop`) are dispatched by static type at compile time.

| Signature                                                                                                                                                                                                                                                                                                            | Notes                                                                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>len(x): num</code><br><code>str(x): str</code>                                                                                                                                                                                                                                                                 | length of a <code>str</code> , <code>list</code> , or <code>map</code><br><code>num</code> , <code>bool</code> , or <code>str</code> to text; integers<br>print clean |
| <code>num(s: str): num?</code><br><code>push(xs: list&lt;T&gt;, v: T)</code><br><code>pop(xs: list&lt;T&gt;): T?</code>                                                                                                                                                                                              | parse; <code>nil</code> on failure (including "")<br>append<br>remove and return last; <code>nil</code> when<br>empty                                                 |
| <code>floor(x: num): num</code><br><code>num): num</code><br><code>abs(x: num): num</code><br><code>num): num</code><br><code>min(a: num, b: num): num</code><br><code>max(a, b): num</code><br><code>sin(x: num): num</code><br><code>num): num</code><br><code>atan(y: num, x: num): num</code><br><code>pi</code> | <code>ceil(x:</code><br><code>sqrt(x:</code><br><br><code>cos(x: radians</code><br><br>two-argument arctangent<br>constant, 3.14159...                                |
| <code>rand(): num</code><br><br><code>srand(seed: num)</code><br><code>env(name: str): str?</code><br><br><code>check(cond: bool, msg: str)</code>                                                                                                                                                                   | uniform [0, 1); same sequence every-<br>where<br>reseed for replayable runs<br>read an environment variable; <code>nil</code> if<br>unset<br>runtime assertion        |

## The `lt.*` engine interface

| Call                                                              | Notes                       |
|-------------------------------------------------------------------|-----------------------------|
| <i>Frame and screen</i>                                           |                             |
| <code>lt.W lt.H</code>                                            | constants 400, 240          |
| <code>lt.clear(r,g,b)</code>                                      | clear colour and depth      |
| <code>lt.time(): num</code>                                       | seconds since boot          |
| <code>lt.screenshot(path: str)</code>                             | save frame as BMP           |
| <code>lt.quit()</code>                                            |                             |
| <code>lt.escape_quits(enable: bool)</code>                        |                             |
| <i>3D scene</i>                                                   |                             |
| <code>lt.camera(ex,ey,ez, tx,ty,tz, fov=55)</code>                | position and look-at        |
| <code>lt.light(dx,dy,dz, ambient=0.35)</code>                     | directional light           |
| <code>lt.point_light(i, x,y,z, radius, r=1,g=1,b=1)</code>        | slots 0–3                   |
| <code>lt.fog(start, end, r,g,b)</code>                            | linear by depth             |
| <i>Meshes</i>                                                     |                             |
| <code>lt.cube()</code> <code>lt.plane(segs=1)</code>              | unit primitives             |
| <code>lt.sphere(seg=12)</code>                                    |                             |
| <code>lt.cylinder(seg=16)</code>                                  |                             |
| <code>lt.cone(seg=16)</code>                                      |                             |
| <code>lt.mesh(verts: list&lt;num&gt;): num</code>                 | 12 floats per vertex        |
| <code>lt.load_mesh(path: str): num</code>                         | Wavefront OBJ               |
| <code>lt.draw(m, x,y,z, rx,ry,rz, sx,sy,sz, r,g,b, tex=-1)</code> | lit draw                    |
| <code>lt.draw_lerp(a, b, t, ..., tex=-1)</code>                   | keyframe tween              |
| <code>lt.billboard(tex, x,y,z, w,h, ...)</code>                   | camera-facing quad          |
| <code>lt.shadow(x,y,z, radius, alpha=0.35)</code>                 | grounding blob              |
| <i>2D</i>                                                         |                             |
| <code>lt.rect(x,y,w,h, r,g,b, a=1)</code>                         | filled, alpha-blended       |
| <code>lt.load_texture(path: str): num</code>                      | BMP; magenta is transparent |
| <code>lt.sprite(tex, x,y, sx=1,sy=1)</code>                       | top-left anchored           |
| <code>lt.sprite_ex(tex, cx,cy, sx,sy, rot,r,g,b,a)</code>         | centre, rotate, tint        |
| <code>lt.sprite_uv(tex, x,y,w,h, u0,v0,u1,v1)</code>              | atlas sub-rect              |
| <code>lt.print(text: str, x,y, r=1,g=1,b=1,a=1)</code>            | 8×8 font                    |
| <i>Audio, input, saves</i>                                        |                             |

| Call                                       | Notes                                      |
|--------------------------------------------|--------------------------------------------|
| <code>lt.load_sound(path: str):</code>     | WAV                                        |
| <code>num</code>                           |                                            |
| <code>lt.play(sound, volume=1,</code>      | channel, or <code>-1</code>                |
| <code>loop=0): num</code>                  |                                            |
| <code>lt.stop(channel) lt.volume(v)</code> |                                            |
| <code>lt.key(name: str): bool</code>       | held                                       |
| <code>lt.pressed(name: str): bool</code>   | went down this frame                       |
| <code>lt.gamepad(): bool</code>            |                                            |
| <code>lt.rumble(low, high, ms)</code>      |                                            |
| <code>lt.save(name: str, data:</code>      | binary-safe                                |
| <code>str): bool</code>                    |                                            |
| <code>lt.load_save(name: str):</code>      | optional; <code>nil</code> when unreadable |
| <code>str?</code>                          |                                            |

Input names: `left right up down z x c space return escape a s d w`. On a gamepad, the d-pad and stick map to the directions, and A, B, Y, Start map to `z, x, c, return`.

### C.1 Bits and register widths (0.3)

`bit_and`, `bit_or`, `bit_xor`, `bit_not`, `bit_shl`, and `bit_shr` accept unsigned 32-bit integers. Shift counts are integers 0–31. `u8` and `u16` wrap safe integers modulo 256 and 65536. `hex(n, width=1)` and `bin(n, width=1)` format unsigned values with minimum-width zero padding (1–8 and 1–32); no truncation. All functions reject invalid numeric domains at runtime. See Chapter 9.