1. Loft Language Reference

A statically-typed scripting language with null safety and built-in parallel execution.

Version 0.8.0

Every code example in this document is an executable part of the test suite.

Contents

2. Getting Started

Loft is a lightweight scripting language with null safety, built-in parallel execution, and a fast interpreter called **loft**. This page shows how to install loft and write your first program.

2.0.1. Prerequisites

Building from source requires the Rust toolchain (Rust 1.82 or later). Install it from rustup.rs if you do not already have it:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

2.0.2. Install from source

The quickest way to get the `loft` binary on your PATH:

```
cargo install --git https://github.com/jjstwerff/loft --bin loft
```

This compiles loft in release mode and places the binary in `~/.cargo/bin/`, which is already on your PATH if you used rustup.

2.0.3. Build locally

Clone the repository and build with Cargo:

```
git clone https://github.com/jjstwerff/loft
cd loft
cargo build --release
```

The binary is at `target/release/loft`. Copy it anywhere on your PATH, for example:

```
cp target/release/loft ~/.local/bin/
```

2.0.4. Verify the installation

```
loft --version
```

2.0.5. Hello, World

Create a file called `hello.loft`:

```
fn main() {
    println("Hello, Loft!");
}
```

Run it:

```
loft hello.loft
```

```
Hello, Loft!
```

2.0.6. A slightly bigger program

Variables, loops, and string interpolation all work without any ceremony:

```
fn main() {
    // Variables are mutable by default; types are inferred.
    total = 0;
    for n in 1..=10 {
        total += n;
    }
    println("Sum 1..10 = {total}");

    // Vectors use square-bracket literals.
    words = ["loft", "is", "fast"];
    for w in words {
        print("{w} ");
    }
    println("");
}
```

2.0.7. Standard library

The standard library is a set of `.loft` files bundled alongside the loft binary. They are loaded automatically before your program runs — you do not need any `import` or `use` statement to call `println`, `assert`, or any other built-in function.

The full function reference is in the Standard Library section of these docs.

2.0.8. Using libraries

Place a library file (e.g. `lib/myutil.loft`) next to your program, then bring it into scope:

```
use myutil;

fn main() {
    result = myutil::compute(42);
    println("{result}");
}
```

See Libraries for the full search path and naming rules.

2.0.9. Editor setup

Loft syntax highlighting extensions are planned. In the meantime:

- **VS Code** — use Rust syntax highlighting as a close approximation.
- **Vim / Neovim** — associate `*.loft` with Rust via `autocmd BufRead *.loft set filetype=rust` in your `vimrc`.
- **Any editor** — the Web IDE (in progress) provides syntax highlighting and navigation in the browser.

2.0.10. Next steps

- Read the language overview starting with Keywords and control flow.
- See vs Rust if you are coming from Rust.
- Browse the Standard Library reference for built-in functions.

- Check the Roadmap for planned features.

3. vs Rust

Loft is inspired by Rust but designed for general-purpose scripting with a shorter learning curve and less ceremony. It trades some of Rust's safety guarantees and expressive power for a simpler mental model. This page documents the most important differences so that Rust programmers know exactly what they gain and what they give up.

3.0.1. Variables — no let or mut

```
x = 42
x += 1
const y = 42   // opt-in: locked in debug builds
```

```
let mut x = 42;
x += 1;
let y = 42;     // immutable by default; compiler-enforced
```

Upside Less boilerplate. No need to decide upfront whether a variable will be mutated. Variables are mutable by default, so refactoring is frictionless. Use const to signal that a value should not change — in debug builds the runtime locks the store immediately after initialisation, catching accidental writes early.

Downside Immutability is opt-in, not the default. Rust makes variables immutable unless you write mut, catching accidents at compile time for free. Loft's const lock is only checked at runtime and only in debug builds, so it provides less of a safety net.

3.0.2. Null instead of Option<T>

```
struct Point {
    label: text,          // nullable
    x: integer not null,  // never null
}
p = Point { x: 1 };
assert(p.label == null, "absent");
```

```
struct Point {
    label: Option<String>,  // explicit absence
    x: i32,                 // always present
}
let p = Point { label: None, x: 1 };
assert!(p.label.is_none());
```

Upside Natural for database records where fields are often absent. No Some(x)/None wrapping. Conditionals on null are concise: if v == null.

Downside Null is opt-out, not opt-in. A field is nullable unless explicitly marked not null, so forgetting the annotation leaves a potential null silently. In Rust, Option is visible in the type and forces handling at every call site.

3.0.3. Parameter mutability — const and & instead of ownership

```
// &: vector growth (append) is visible to caller
fn push_two(v: &vector<integer>) {
    v += [1, 2];  // caller sees the change
}
// const: read-only (locked in debug builds)
fn count(v: const vector<integer>) -> integer {
    length(v)
}
// &: explicit write-back for primitives
fn add_to(n: &integer, delta: integer) {
    n += delta;
}
```

```
fn push_two(v: &mut Vec<i32>) {
    v.push(1);
    v.push(2);
}
fn count(v: &Vec<i32>) -> usize {
    v.len()
}
fn add_to(n: &mut i32, delta: i32) {
    *n += delta;
}
```

Upside No borrow-checker errors. No lifetime annotations. No ownership transfer to reason about. Struct field mutations through a parameter are always visible to the caller. Use & on a collection parameter to allow vector growth (append) to propagate back. const parameters signal read-only intent, and in debug builds the runtime locks the store for the duration of the call — enough to catch most accidents during development.

Downside The compile-time guarantees are much weaker. Rust's borrow checker statically proves no aliased mutations, no dangling references, and no data races — before the program ever runs. Loft's const is a debug-only runtime check. Aliasing is unchecked at compile time, and the engine's Rust runtime is what truly enforces memory safety.

3.0.4. ^ is XOR; exponentiation uses pow()

```
area = PI * pow(r, 2.0)  // exponentiation via pow()
bits = a | b             // bitwise OR
mask = a & b             // bitwise AND
xor  = a ^ b             // bitwise XOR
```

```
let area = PI * r.powi(2); // or r.powf(2.0)
let bits = a | b;          // bitwise OR
let mask = a & b;          // bitwise AND
let xor  = a ^ b;          // bitwise XOR
```

Upside ^ behaves exactly as Rust developers expect — it is bitwise XOR. Bitwise operator precedence matches Rust and C: | (loose) ⟶ ^ ⟶ & ⟶ shifts ⟶ arithmetic (tight).

Downside Exponentiation requires a function call: pow(base, exp) for float/single — there is no ** or ^ exponentiation operator. Integer exponentiation is not in the standard library; compute it with a loop or cast to float first.

3.0.5. No loop or while

```
for _ in 0..2147483647 {  // large upper bound
    if done() { break; }
    step();
}
```

```
loop {
    if done() { break; }
    step();
}
while !done() { step(); }
```

Upside Every loop has an iteration variable, making it easy to add index tracking or break conditions without restructuring. For a near-unbounded poll loop use for _ in 0..2147483647 — the break will fire long before the limit is reached in practice.

Downside while condition { } is universally understood and directly expresses intent. Its absence surprises Rust and C programmers. Event loops and polling patterns are more verbose with the open-ended range workaround.

3.0.6. Filtered loops — for … if …

```
for x in items if x > 0 {
    total += x;
}
```

```
for x in items.iter().filter(|&&x| x > 0) {
    total += x;
}
```

Upside Reads like natural language. No closure syntax, no double-reference in the filter predicate. v#remove inside a filtered loop also safely removes the current element while iterating — something that requires careful index management in Rust.

Downside The inline if filter is limited to the loop it lives in. For multi-stage pipelines, loft now offers map(v, fn f), filter(v, fn pred), and reduce(v, fn f, init) as composable higher-order functions — see section 11. Rust's lazy iterator chain (.filter().map().take_while()) avoids intermediate allocations; loft's higher-order functions allocate a new vector at each stage.

3.0.7. Loop attributes: #first, #count, #index

```
for x in 1..=5 {
    if !x#first { b += ", "; }
    b += "{x#count}:{x}";
}
```

```
for (i, x) in (1..=5).enumerate() {
    if i != 0 { b.push_str(", "); }
    b.push_str(&format!("{i}:{x}"));
}
```

Upside No tuple destructuring needed. x#first reads as "is this the first x", which is self-explanatory. Eliminates helper counter variables for the common pattern of comma-separated output.

Downside The #attribute syntax is unique to loft and unfamiliar to everyone else. Rust's .enumerate() composes freely with other adapters; loft's attributes are only available on the loop variable of the innermost loop.

3.0.8. Named loop break — x#break

```
for x in 1..5 {
    for y in 1..5 {
        if x * y >= 6 { x#break; }
    }
}
```

```
'outer: for x in 1..5 {
    for y in 1..5 {
        if x * y >= 6 { break 'outer; }
    }
}
```

Upside Reuses the existing loop variable name as the label. No separate 'label declaration at the loop head. The name x#break reads as "break out of the loop over x".

Downside Rust's lifetime-label syntax 'outer is explicit and visually separate from the loop body. x#break is easy to confuse with the loop attributes x#first and x#count, which have completely different semantics.

3.0.9. Methods by self name, not impl blocks

```
fn greet(self: Person) -> text {
    "Hello, {self.name}!"
}
fn name_len(self: const Person) -> integer {
    length(self.name)  // const: read-only access guaranteed
}
p.greet()     // dot syntax
greet(p)      // free-function call — identical
```

```
impl Person {
    fn greet(&self) -> String {
        format!("Hello, {}!", self.name)
    }
    fn name_len(&self) -> usize {
        self.name.len()
    }
}
p.greet();     // only dot syntax
```

Upside No impl block ceremony. Methods and free functions are the same thing — just a naming convention. Functions can be added to any type from any file without modifying the original struct definition, similar to extension methods. Mark the first parameter const to guarantee the method never modifies the receiver — analogous to Rust's &self.

Downside No consuming self. Plain (non-const) methods behave like &mut self — the caller's value is always mutably aliased. Multiple functions with the same name but different non-variant self types are a compile error — no standard overloading.

3.0.10. Polymorphic enum dispatch

```
enum Shape {
    Circle { r: float },
    Rect   { w: float, h: float }
}
fn area(self: Circle) - float { PI * pow(self.r, 2.0) }
fn area(self: Rect)   - float { self.w * self.h }

s.area()    // dispatches on the runtime variant
```

```
enum Shape {
    Circle { r: f64 },
    Rect   { w: f64, h: f64 },
}
fn area(s: &Shape) -> f64 {
    match s {
        Shape::Circle { r } => PI * r * r,
        Shape::Rect { w, h } => w * h,
    }
}
```

Upside Each variant's behaviour lives in its own small function — easy to read and extend. Adding a new type of shape only requires a new fn area(self: NewShape) without modifying the dispatch site. Feels like OOP method overriding without the inheritance.

Downside Rust's match is exhaustive: the compiler forces you to handle every variant. In loft, leaving a variant without an implementation emits a compiler warning but does not stop compilation. To suppress the warning and produce a null return, write an explicit empty-body stub: fn area(self: NewShape) -> float { }. Exhaustiveness is enforced by discipline, not the type system.

3.0.11. No closures — but compile-checked function references

```
// fn <name> produces a compile-time-checked function reference
fn double(x: integer) - integer { x * 2 }
fn is_even(x: integer) - integer { x % 2 == 0 }
fn add(a: integer, b: integer) - integer { a + b }

// Store, call, or pass a fn-ref like any other value:
f = fn double;
assert(f(5) == 10);

// Higher-order functions use fn-refs directly:
nums = [1, 2, 3, 4, 5];
doubled = map(nums, fn double);
evens  = filter(nums, fn is_even);
total  = reduce(nums, 0, fn add);

// Parallel dispatch — worker result collected per element:
for item in scores par(result=score_fn(item), 4) {
    total += result;
}
```

```
fn double(r: &Score) -> i32 { r.value * 2 }

// Rayon parallel iterator:
let total: i32 = scores.par_iter().map(double).sum();

// Closure captures context freely:
let offset = 5;
let shifted: Vec<i32> = v.iter()
    .map(|x| x + offset)
    .collect();
```

Upside Simpler mental model. No capture modes (move vs borrow), no Fn/FnMut/FnOnce trait bounds, no lifetime constraints on captured variables. The fn <name> expression gives a compile-checked reference to any named function — the compiler verifies the name exists and has a matching signature before emitting any code.

Downside No closures or lambdas. A fn <name> reference cannot capture surrounding variables — context must be embedded in the data or passed as an explicit parameter. Indirect calls are supported (f(args) where f holds a fn-ref), but fn-refs cannot be compared, printed, or used in arithmetic. Rust's Fn/FnMut/FnOnce traits give far more flexibility for callbacks that need local state.

3.0.12. No generic functions or trait system

```
// No generic functions — must write a version per type
fn max_int(a: integer, b: integer) - integer {
    if a  b { a } else { b }
}
fn max_float(a: float, b: float) - float {
```

```
    if a  b { a } else { b }
}
```

```
fn max<T: PartialOrd>(a: T, b: T) -> T {
    if a > b { a } else { b }
}
```

Upside Nothing to learn about type parameters, trait bounds, where clauses, dyn Trait, impl Trait, or associated types. Collections (vector<T>, hash<T>, sorted<T>) are generic at the engine level, covering the most common need.

Downside Code cannot be written once and reused across types. Every generic algorithm must be duplicated per type or pushed down into the standard library. There is no way for user code to define an "anything sortable" or "anything printable" abstraction.

3.0.13. String formatting — embedded expressions

```
msg  = "Hi {name}, score: {score:+6.2}"  // width.prec
prec = "{value:.2}"                      // precision only
hex  = "{n:#x}"
list = "{for x in 1..4 {x*2}}"          // "2,4,6"
     = "{\"hello\":3}"                    // nested string literal — works
```

```
let msg  = format!("Hi {name}, score: {:+6.2}", score);
let prec = format!("{:.2}", value);
let hex  = format!("{:#x}", n);
// no in-string iteration; needs a separate collect step
// nested string literals in format args are fine in Rust
```

Upside Concise — no format!() call, no separate variable. Full format expressions (arithmetic, slices, for comprehensions) can appear directly inside {}. Specifiers mirror Rust: :width, :.precision, :width.precision, sign (+), radix (#x, #o, b), alignment (<, >, ^), and zero-padding (0N) all work.

Downside Unknown radix letters in specifiers are compile-time errors (e.g. :5z or :5B are both rejected). Radix letters are case-sensitive: valid ones are b, o, x/X, e, and j/json — uppercase B or O produce an error. Applying a numeric specifier to an incompatible type (such as :x on a text value) is silently ignored rather than flagged.

3.0.14. Built-in parallel for-loops — par(...)

```
fn double(r: const Score) - integer { r.value * 2 }

sum = 0;
for item in scores par(b=double(item), 4) {
    sum += b;   // b is double(item), computed in parallel
}               // results arrive in original order

// Method form:
for item in scores par(b=item.value(), 4) { ... }
```

```
use rayon::prelude::*;

fn double(r: &Score) -> i32 { r.value * 2 }

let sum: i32 = scores.par_iter()
    .map(double)
    .sum();      // order is not guaranteed without extra work

// rayon is an external crate; add to Cargo.toml first
```

Upside Built into the language — no external crate, no Cargo.toml edit. The compiler validates the worker function signature at the call site. Results are delivered in the original vector order. The thread count is set per call, making it easy to tune for the hardware at hand.

Downside The worker must return a primitive (integer, long, float, or boolean) — returning text or a struct reference is not yet supported. Workers cannot capture local variables: context must be embedded as fields alongside the data in the element struct. For multi-stage transformations, use map() / filter() / reduce() sequentially — each stage allocates a new intermediate vector.

4. vs Python

Loft and Python share a similar surface syntax — assignment without type annotations, brace-free expressions, and names that just work. But loft is statically typed and compiled to bytecode, while Python is dynamically typed and interpreted. This page documents the most important differences so that Python programmers know exactly what they gain and what they give up.

4.0.1. Variables — static types inferred at first assignment

```
x = 42           // integer — fixed at first assignment
x += 1
name = "Bob"     // text — a different variable, not a rebind
x = "oops"      // compile error: cannot assign text to integer
```

```
x = 42           # int inferred; can be rebound to any type
x += 1
name = "Bob"
x = "oops"       # fine — x is now a str
```

Upside Type errors are caught before the program runs. Renaming or reshaping data structures surfaces mismatches immediately. There is no equivalent of Python's runtime TypeError, AttributeError, or "NoneType has no attribute X" crash — those classes of bug are detected at compile time.

Downside Types are fixed at first assignment and cannot change. Python's dynamic typing genuinely speeds up prototyping: a function that accepts anything, a list that holds mixed types, or a variable that starts as an integer and becomes a float later are all natural in Python and impossible in loft. Adding a type annotation layer (mypy, pyright) gives Python static safety without giving up its flexibility.

4.0.2. Null — structured absence, not a universal wildcard

```
struct User {
    email: text,             // nullable by default
    age:   integer not null,  // can never be null
}
u = User { age: 30 };
if u.email == null { print("no email"); }
u.age = null;  // compile error: age is not null
```

```
from dataclasses import dataclass
from typing import Optional

@dataclass
class User:
    age: int
    email: Optional[str] = None  # explicit Optional

u = User(age=30)
if u.email is None:
```

```
    print("no email")
u.age = None  # allowed at runtime; mypy catches this
```

Upside Nullability is part of the struct definition — readable at a glance. not null fields are enforced by the compiler and additionally locked at runtime in debug builds, catching accidental null writes early in development. No Optional[T] wrapping needed; the comparison v == null is natural.

Downside The default is nullable, not non-nullable — the safe default is backwards from what static analysis advocates recommend. Python with mypy and Optional[T] annotation gives static guarantees that loft's runtime checks do not. Python's None also participates in truthiness testing (if not email:), pattern matching, and or chaining in ways that loft's null does not support.

4.0.3. Structs vs classes and dicts

```
struct Point { x: float, y: float }
p = Point { x: 1.0, y: 2.0 };
p.x += 0.5;
p.z;          // compile error: no field z on Point
p.x = "hi"; // compile error: cannot assign text to float

fn distance(self: const Point) - float {
    sqrt(self.x * self.x + self.y * self.y)
}
```

```
from dataclasses import dataclass
import math

@dataclass
class Point:
    x: float
    y: float

    def distance(self) -> float:
        return math.sqrt(self.x**2 + self.y**2)

p = Point(x=1.0, y=2.0)
p.z        # AttributeError at runtime (or dict: KeyError)
p.x = "hi" # allowed at runtime; mypy catches this
```

Upside Field access is checked at compile time — typos in field names are caught before any code runs. Struct memory layout is fixed and unboxed; integer and float fields live directly in memory with no heap allocation overhead. Methods are ordinary named functions — they can be added from any file, at any time, without modifying the struct definition.

Downside No inheritance, no __repr__, no operator overloading (__add__, __eq__, etc.), no properties or descriptors. Python's @dataclass generates __init__, __repr__, and __eq__ automatically. Loft structs are data holders; all display and comparison logic must be written by hand. Python also supports plain dicts as lightweight records, which is often more convenient for ad-hoc data.

4.0.4. No while loop

```
// Condition-driven poll loop — use a large upper bound:
for _ in 0..2147483647 {
    if ready() { break; }
    step();
}

// Draining a collection while it has elements:
for _ in 0..2147483647 {
    if length(queue) == 0 { break; }
    process(queue[0]);
    queue#remove;
}
```

```
while not ready():
    step()

while queue:
    process(queue.pop(0))
```

Upside Every loop has an iteration variable, making it easy to add a cycle limit or index tracking without restructuring. The break fires long before the range limit in practice. Filtered loops (for x in v if pred(x)) and loop attributes (x#first, x#count) are only available on for, so a single loop construct covers all cases.

Downside while condition: is instantly understood by every programmer and reads exactly as its semantics. Its absence is surprising and the workaround is verbose. Python also has while … else (executes when the condition first becomes false without a break), a pattern with no clean equivalent in loft.

4.0.5. Polymorphic enum dispatch vs isinstance

```
enum Shape {
    Circle { r: float },
    Rect   { w: float, h: float }
}
fn area(self: Circle) - float { PI * pow(self.r, 2.0) }
fn area(self: Rect)   - float { self.w * self.h }

s.area()   // dispatches on the runtime variant
```

```
import math
from dataclasses import dataclass

@dataclass
class Circle: r: float
@dataclass
class Rect:   w: float; h: float
```

```
def area(s):
    if isinstance(s, Circle): return math.pi * s.r ** 2
    if isinstance(s, Rect):   return s.w * s.h

# Python 3.10+ structural pattern matching:
match s:
    case Circle(r=r): return math.pi * r ** 2
    case Rect(w=w, h=h): return w * h
```

Upside Each variant's behaviour lives in its own small, named function — easy to read, easy to navigate in an editor. Adding a new shape only requires a new fn area(self: NewShape) with no changes to existing code or a central dispatch function. The compiler warns when a variant has no implementation for a called method.

Downside Unlike Rust's match, loft does not enforce exhaustiveness — a missing variant implementation produces only a warning, not a compile error. Python 3.10+ structural pattern matching (match/case) fully destructs the matched value and is exhaustive when case _: is present. Python also supports inheritance-based polymorphism (class Circle(Shape)) and abstract base classes, giving far richer dispatch options.

4.0.6. String formatting — embedded expressions

```
msg  = "Hi {name}, score: {score:+8.2}"
hex  = "{n:#x}"
list = "{for x in 1..4 {x*2}}"   // "2,4,6"
pad  = "{value:08}"               // zero-padded
```

```
msg  = f"Hi {name}, score: {score:+8.2f}"
hex  = f"{n:#x}"
# no loop in f-string; use a join:
lst  = ",".join(str(x*2) for x in range(1, 4))  # "2,4,6"
pad  = f"{value:08}"
```

Upside All loft strings are implicitly format strings — no f prefix needed. Inline for loops inside {} produce comma-separated output without a separate join. Format specifiers mirror Python's f-string mini-language: width, precision, sign, alignment, zero-padding, and radix (#x, #o, b) all work.

Downside Python f-strings accept arbitrary expressions: method calls ({obj.method():.2f}), ternary expressions ({"yes" if ok else "no"}), and join operations inline. Loft restricts what can appear inside {}. Python also supports conversion flags (!r for repr, !s for str, !a for ASCII) and the = debug specifier ({x=} prints x=42); loft has none of these.

4.0.7. Collections — typed and built-in

```
nums:   vector<integer> = [1, 2, 3];
lookup: hash<text>    = {};
lookup["key"] = "value";
scores: sorted<integer> = {};
```

```
scores[user] = 95;  // O(log n) keyed insert

// Element type is enforced at compile time:
nums += ["x"];  // compile error: expected integer
```

```
nums   = [1, 2, 3]          # list — any element type
lookup = {}                 # dict — any key/value type
lookup["key"] = "value"

# sorted dict requires an external package:
from sortedcontainers import SortedDict
scores = SortedDict()
scores[user] = 95

nums.append("x")            # allowed at runtime
```

Upside All collection types — vector, hash, and sorted map — are built in; no extra import or install needed. Element types are checked at compile time. The same [] indexing syntax works on all three. for x in v if pred(x) { v#remove; } safely removes the current element while iterating — something Python requires careful index management for.

Downside Python's built-in dict and list are among the most heavily optimised data structures in any scripting runtime. Python also has set and frozenset (no loft equivalent), ordered insertion semantics on dict (Python 3.7+), and an enormously expressive comprehension syntax ([x*2 for x in v if x > 0]). Loft's map() / filter() / reduce() higher-order functions allocate a new vector at each stage, while Python's generators are lazy and allocation-free until consumed.

4.0.8. No closures or lambdas — but compile-checked function references

```
fn double(x: integer) -> integer { x * 2 }
fn is_even(x: integer) -> boolean { x % 2 == 0 }
fn add(a: integer, b: integer) -> integer { a + b }

f = fn double;          // compile-checked fn reference
assert(f(5) == 10);

nums = [1, 2, 3, 4, 5];
doubled = map(nums, fn double);
evens   = filter(nums, fn is_even);
total   = reduce(nums, 0, fn add);
```

```
double  = lambda x: x * 2
is_even = lambda x: x % 2 == 0

f = double
assert f(5) == 10

nums = [1, 2, 3, 4, 5]
```

```
doubled = list(map(lambda x: x * 2, nums))
evens   = list(filter(lambda x: x % 2 == 0, nums))
total   = sum(nums)

# capture context freely:
offset = 10
shifted = [x + offset for x in nums]
```

Upside Simpler mental model — no capture modes, no scope surprises. The fn <name> expression gives a compile-checked reference to any named function; the compiler verifies the name exists and has a matching signature before emitting code. Higher-order functions (map, filter, reduce) accept fn-refs directly.

Downside No closures means context cannot be captured — any extra data must be embedded in the element struct or passed as an explicit parameter. Python's lambda and nested def close over surrounding variables naturally, making short callbacks, key functions, and event handlers concise. List comprehensions ([f(x) for x in v]) are shorter and more Pythonic than map(v, fn f). Indirect calls are supported, but fn-refs cannot be compared or inspected at runtime.

4.0.9. No exception handling

```
// Preconditions are assertions; failures abort the program:
fn divide(a: float, b: float) - float {
    assert(b != 0.0, "division by zero");
    a / b
}

// I/O errors surface as null:
f = open("data.txt");
if f == null { print("file not found"); }
```

```
def divide(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("division by zero")
    return a / b

try:
    result = divide(x, y)
except ValueError as e:
    print(f"error: {e}")

try:
    with open("data.txt") as f:
        data = f.read()
except FileNotFoundError:
    print("file not found")
```

Upside No exception hierarchy to learn, no accidental exception swallowing, no overhead from unwinding the stack. File and I/O operations signal failure by returning null, which is explicit and

cheap to check. The control flow of a loft function is always straightforward — there are no hidden exit paths.

Downside There is no structured way to recover from errors in user code. An assertion failure aborts the entire program. Python's try/except/finally/else and user-defined exception hierarchies allow fine-grained error handling, retry logic, cleanup on failure, and graceful degradation — patterns that are impossible to express in loft today.

4.0.10. Built-in parallel for-loops — par(...)

```
fn score(item: const Record) - integer { item.value * 2 }

total = 0;
for item in records par(s=score(item), 4) {
    total += s;   // results arrive in original order
}                 // 4 worker threads, no GIL
```

```
from concurrent.futures import ProcessPoolExecutor

def score(item): return item.value * 2

# ProcessPoolExecutor: bypasses GIL via separate processes
with ProcessPoolExecutor(max_workers=4) as ex:
    results = list(ex.map(score, records))
total = sum(results)

# ThreadPoolExecutor: simpler but GIL limits CPU-bound work
```

Upside Built into the language — no import, no boilerplate. The GIL does not apply; worker threads run on separate OS threads inside the same process. Results arrive in the original vector order. The compiler validates the worker function signature at the call site. The thread count is set per call, making it easy to tune for the hardware.

Downside Workers must return a primitive (integer, long, float, or boolean) — returning text or a struct reference is not yet supported. Context must be embedded as fields in the element struct; workers cannot capture local variables. Python's ProcessPoolExecutor works with any picklable object and gives full control over timeouts, cancellation, and error propagation.

4.0.11. No generic functions

```
// Must write a version per type — no type parameters:
fn max_int(a: integer, b: integer) - integer {
    if a  b { a } else { b }
}
fn max_float(a: float, b: float) - float {
    if a  b { a } else { b }
}
```

```
# Duck typing: works for any T that supports >
def max_val(a, b):
    return a if a > b else b

# With type hints (Python 3.12+):
from typing import TypeVar
T = TypeVar("T")
def max_val(a: T, b: T) -> T:
    return a if a > b else b
```

Upside Nothing to learn about type parameters, bounds, variance, or protocols. Collections (vector<T>, hash<T>, sorted<T>) are generic at the engine level, covering the most common need without any user-visible type parameter syntax.

Downside Code cannot be written once and reused across types. Every generic algorithm must be duplicated per type or moved into the standard library. Python's duck typing means a function that calls len(x) automatically works on any type that implements __len__ — no explicit annotation required. Python 3.12 TypeVar syntax and structural subtyping (Protocol) give this generality with optional static checking.

4.0.12. Function signatures — no defaults, keyword args, or *args

```
// All arguments are positional and required:
fn connect(host: text, port: integer, timeout: integer) { ... }

connect("localhost", 8080, 30);  // OK
connect("localhost", 8080);      // compile error: wrong argument count
```

```
def connect(host: str, port: int = 80, timeout: int = 30):
    ...

connect("localhost")            # uses defaults
connect("localhost", 8080)      # overrides port only
connect("localhost", timeout=5) # keyword arg — skips port

def log(*args, **kwargs): ...   # variadic
```

Upside Every call site is explicit — there are no hidden default values to look up. The number and order of arguments is always visible at the call site, making code easier to follow without jumping to the function definition.

Downside No default parameter values means wrapper overloads must be written by hand. No keyword arguments means callers of functions with many parameters must remember the correct order. No *args or **kwargs means variadic dispatch must be handled with a vector parameter. Python's flexible argument syntax is one of its most ergonomic features, enabling clean APIs, decorator patterns, and configuration-driven code that is awkward to express in loft.

4.0.13. Ecosystem — minimal standard library

```
// import "mylib" loads a .loft file relative to the script.
// The full standard library ships inside the interpreter binary;
// there is no package manager or external dependency system.

// Built-in: text, math, file I/O, collections,
//           logging, threading, image, lexer/parser
```

```
import numpy as np          # numerical arrays / SIMD
import pandas as pd         # dataframes
import requests             # HTTP
import flask                # web framework
import sqlalchemy           # database ORM
import scikit_learn         # machine learning
# 500 000+ packages on PyPI, installable with:
#   pip install <package>
```

Upside Zero external dependencies — the interpreter is a single self-contained binary. Deployment is copying one file. There is no requirements.txt, no virtual environment, no version conflict, and no supply-chain risk. The built-in library covers text manipulation, math, file I/O, typed collections, parallel execution, image handling, and a full lexer/parser framework.

Downside Python's ecosystem is its defining advantage. NumPy, pandas, scikit-learn, TensorFlow, requests, Flask, SQLAlchemy, pytest, and hundreds of thousands of other packages are not available to loft programs. Any data science, web development, database integration, or protocol implementation task will require reimplementing from scratch what Python solves with a single pip install. For these domains, loft is the wrong tool today.

4.0.14. Exponentiation uses pow(); ^ is XOR

```
area  = PI * pow(r, 2.0)  // exponentiation via pow()
bits  = a | b              // bitwise OR
xor   = a ^ b              // bitwise XOR — not exponentiation
cube  = pow(x, 3.0)       // cube root: pow(x, 1.0/3.0)
```

```
import math
area  = math.pi * r ** 2   # ** is exponentiation
bits  = a | b              # bitwise OR
xor   = a ^ b              # bitwise XOR
cube  = x ** 3             # integer cube — exact
cube  = x ** (1/3)         # cube root as float
```

Upside ^ behaves as bitwise XOR in both loft and Python — no mismatch. Bitwise operator precedence matches: | (loose) → ^ → & → shifts → arithmetic (tight). The pow() function is explicit, avoiding any ambiguity about whether ^ means XOR or exponentiation.

Downside Python's ** operator works on integers, floats, and complex numbers and produces the exact type the operands imply. Loft's pow() operates on float and single only — integer exponentiation has

no built-in; compute it with a loop or cast to float first. Python's pow(base, exp, mod) three-argument form computes modular exponentiation efficiently; loft offers no equivalent.

5. Keywords

This page covers the building blocks that shape how your program makes decisions and repeats work: conditions, loops, breaks, and a few loop helpers that make common patterns shorter. Every example is verified with an assert so you can see the exact expected result.

```
fn main() {
```

5.0.1. Conditionals

'if' runs a block only when its condition is true. If the condition is false the block is skipped entirely — nothing else happens. 'panic' stops the program immediately with a message. During development it is a clear way to mark situations that should never occur.

```
  if 2 > 5 {
    panic("Incorrect test");
  }
```

Combine conditions with 'and' / 'or' (or their symbol equivalents '&&' / '||'). Note that '&' is the bitwise AND operator — it works on the individual bits of a number, which is different from the logical 'and' keyword. An 'if/else' chain picks exactly one branch to execute.

```
  a = 12;
  if a > 10 and a & 7 == 4 {
    a += 1;
  } else {
    a -= 1;
  }
```

Text enclosed in double quotes can contain expressions inside curly braces — the expression is evaluated and its result is inserted into the text. 'assert' checks that its first argument is true; if it is false the second argument is printed as an error message.

```
  assert(a == 13, "Incorrect value {a} != 13");
```

5.0.2. if as an expression

Every block in Loft produces a value — the last expression inside it. That means you can use 'if/else' on the right-hand side of an assignment, picking between two values based on a condition. No ternary operator needed.

```
  b = if a == 13 {
    "Correct"
  } else {
    "Wrong"
  };
  assert(b == "Correct", "Logic expression");
```

5.0.3. Iteration

‘for x in a..b’ loops over the integers starting at a and stopping before b (exclusive upper bound). Use ‘..=’ to include the upper bound. Here we sum 1 + 2 + 3 + 4 + 5 = 15.

```
t = 0;
for a in 1..6 {
  t += a;
}
assert(t == 15, "Total was {t} instead of 15");
```

‘rev(range)’ reverses the direction of any range. ‘1..=5’ is inclusive, so it visits 5, 4, 3, 2, 1 in that order. Multiplying each digit into a running total builds the number 54321.

```
t = 0;
for a in rev(1..=5) {
  t = t * 10 + a;
}
assert(t == 54321, "Result was {t} instead of 54321");
```

5.0.4. Nested loops and break

Loft has no ‘while’ or ‘loop’ keyword — all repetition uses ‘for’ with a range, so there is always a clear upper bound on how many iterations can occur. ‘break’ exits the nearest enclosing loop immediately. To exit an outer loop from inside an inner one, write ‘outerVar#break’. Here x#break leaves both loops when the product x*y reaches 16.

```
b = "";
for x in 1..5 {
  for y in 1..5 {
    if y > x {
      break;
```

this breaks the inner y loop

```
    }
    if x * y >= 16 {
      x# break;
    }
    if len(b) > 0 {
      b += "; ";
    }
    b += "{x}:{y}";
  }
}
assert(b == "1:1; 2:1; 2:2; 3:1; 3:2; 3:3; 4:1; 4:2; 4:3", "Incorrect sequence
'{b}'");
```

5.0.5. Loop helpers: #first and #count

Inside a loop body you have access to some useful metadata about the current iteration. ‘x#first’ is true only for the very first element that passes the filter. ‘x#count’ is a zero-based index counting only the elements that were not filtered out. An ‘if’ clause directly after ‘in range’ filters which values enter the loop — here we skip every value where x % 3 == 1, keeping 2, 3, 5, 6, 8, 9.

```
  b = "";
  for x in 1..=9 if x % 3 != 1 {
    if !x#first {
      b += ", ";
    }
    b += "{x#count}:{x}";
  }
  assert(b == "0:2, 1:3, 2:5, 3:6, 4:8, 5:9", "Sequence '{b}'");
```

5.0.6. Formatting a loop inline

A for loop can appear inside a format string. The results are collected into a bracketed, comma-separated list. A format specifier after the closing brace is applied to every element — ‘:02’ means at least 2 digits, zero-padded. This is handy for building compact representations on the fly.

```
  assert("a{for x in 1..7 {x*2}:02}b" == "a[02,04,06,08,10,12]b", "Format
range");
}
```

6. Texts

Text is one of the most frequently used types in any real program — for output, for reading input, for building messages. This page shows you how Loft stores and manipulates text, how to measure it, slice it, search it, and format it exactly the way you need.

The key thing to know upfront: Loft stores text as UTF-8, the same encoding used on the web and in most modern systems. Nearly all operations work in bytes rather than in Unicode characters. That makes them fast and simple, but you need to keep multi-byte characters in mind when you slice by position.

```
fn main() {
```

6.0.1. Joining and measuring text

Use '+' to join two pieces of text into one. The result is a new value — neither of the originals is modified. Placing a value name inside curly braces inside a format string inserts the value as text. You can also control the width: '{a:4}' pads the value on the right to at least 4 characters.

```
    a = "1" + "2";
    assert("'{a:4}'" == "'12  '", "Formatting text");
    assert(len(a + "123") == 5, "Text length");
```

'len()' counts bytes, not visible characters. An emoji like '😃' takes 4 bytes in UTF-8, a heart symbol '♥' takes 3, and every plain ASCII letter takes exactly 1. Keep this in mind whenever you use len() to check how "long" something is to a human reader.

```
    assert(len("😃") == 4, "Emoji byte length");
    assert(len("♥") == 3, "Heart byte length");
    assert(len("abc") == 3, "ASCII byte length");
```

6.0.2. Reading individual characters

Square brackets with a single byte index give you the character at that position. For plain ASCII text every byte is one character, so indexing by position is straightforward. Loft will return the full Unicode character even if it spans multiple bytes.

```
    s = "ABCDE";
    assert(s[0] == 'A', "Character at byte 0");
    assert(s[2] == 'C', "Character at byte 2");
```

6.0.3. Slicing text

A slice 's[start..end]' gives you the bytes from 'start' up to but not including 'end'. You can omit the start to begin at 0, or omit the end to go all the way to the last byte. Loft snaps offsets to valid character boundaries so you can never accidentally cut a multi-byte character in half.

```
    assert(s[0..2] == "AB", "Explicit start slice");
    assert(s[..2] == "AB", "Open-start slice");
```

```
assert(s[1..4] == "BCD", "Sub-string by byte range");
assert(s[3..] == "DE", "Open-ended sub-string");
```

A negative end index counts backwards from the end of the text. '−1' means "stop one byte before the very last byte". Combined with a start offset this lets you trim a known suffix without knowing the exact length.

```
txt = "12😊🙃45";
assert(txt[2..- 1] == "😊🙃4", "UTF-8 sub-string by byte range");
```

6.0.4. Iterating over characters

A 'for' loop over a text value visits one Unicode character at a time, even when characters span multiple bytes. Two loop helpers give you position information without any extra code: 'c#index' is the byte offset where the current character starts. 'c#next' is the byte offset immediately after the current character ends. This makes it easy to build a byte-offset map or to collect characters starting from a specific position.

```
result = "";
positions =[];
nexts =[];
for c in "Hi 😊!" {
  positions +=[c#index];
  nexts +=[c#next];
  if c#index >= 3 {
    result += c;
  }
}
assert(result == "😊!", "Character iteration was {result}");
assert("{positions}" == "[0,1,2,3,7]", "Character positions was {positions}");
assert("{nexts}" == "[1,2,3,7,8]", "Character nexts was {nexts}");
```

6.0.5. Searching inside text

These built-in functions answer common "does this text contain…?" questions. 'starts_with' and 'ends_with' check the boundaries. 'find' returns the byte offset of the first match, or null if not found. 'contains' is true if the needle appears anywhere in the text. All positions are byte offsets, consistent with 'len()' and slicing.

```
assert("something".starts_with("some"), "starts_with");
assert("something".ends_with("thing"), "ends_with");
assert("something".find("th") == 4, "find returns byte offset");
assert("a longer text".contains("longer"), "contains");
```

'trim()' removes spaces and other whitespace from both ends of a text value. It is useful when reading user input or parsing text from a file.

```
  assert(trim("  hello  ") == "hello", "trim");
```

6.0.6. Escaping braces in format strings

Inside a format string ‘{‘ and ‘}’ are special: they introduce an interpolated expression. To include a literal brace in the output, double it: ‘{{‘ produces ‘{‘ and ‘}}’ produces ‘}’.

```
  brace_inner = "cd";
  assert("ab{{cd}}e" == "ab{{{brace_inner}}}e", "Escaping braces");
```

6.0.7. Aligning text in a fixed-width field

The ‘:’ specifier controls alignment and width. ‘<’ left-aligns the value, ‘>’ right-aligns it (the default). The width can be a constant or a small arithmetic expression — here ‘2+3’ evaluates to 5, giving a field of width 5.

```
  vr = "abc";
  assert("1{vr:<2+3}2{vr}3{vr:6}4{vr:>7}" == "1abc  2abc3abc   4    abc", "Text
alignment");
}
```

7. Integers

Numbers are at the heart of almost every program. This page covers the integer types Loft provides, the arithmetic and bitwise operations you can perform on them, how to convert between numbers and text, and what Loft does when something goes wrong — such as dividing by zero.

The default number type is 'integer': a 32-bit signed whole number, the same as i32 in Rust. It can hold values from about −2 billion to +2 billion. For larger values Loft also has 'long' (64-bit), shown at the end of this page. For decimal (fractional) numbers, see the Float page.

```
fn main() {
```

7.0.1. Converting between numbers and text

Wrapping a value in '{...}' inside a string formats it as text. Going the other way, 'as integer' parses a text value into a number. If the text cannot be parsed, the result is null — not a crash.

```
  v = 4;
  assert("{v}" == "4", "Format integer as text");
  assert("123" as integer == 123, "Parse text to integer");
  assert(!("abc" as integer), "Unparseable text gives null");
```

7.0.2. Arithmetic and operator precedence

Loft follows standard mathematical precedence: '*' and '/' before '+' and '-'. Bitwise operators (<<, &, ^) have their own precedence — when mixing them with arithmetic, parentheses make your intent clear and avoid surprises. Note: '^' is XOR, not exponentiation. Use pow(base, exp) for powers.

```
  assert(1 + 2 * 4 == 9, "Multiplication before addition");
  assert(1 + 2 << 2 == 12, "Shift: (1+2) << 2 = 12");
  assert(0x0a8 & 15 == 8, "Bitwise AND masks low 4 bits");
  assert(42 ^ 0b111111 == 21, "Bitwise XOR");
  assert(105 % 100 == 5, "Modulus (remainder)");
  assert(pow(2.0, 3.0) == 8.0, "pow() for exponentiation");
```

'abs()' returns the absolute value — the distance from zero, always positive.

```
  assert(1 + abs(-2) == 3, "abs(-2) == 2");
```

7.0.3. Division by zero produces null, not a crash

Most languages crash or throw an exception on division by zero. Loft produces null instead, which behaves like false in conditions. This lets you handle missing or bad data gracefully with a simple '!'.

```
  a = 2 * 2;
  a -= 4;
```

a is now 0

```
  assert(!(12 / a), "Division by zero gives null");
```

7.0.4. Compile-time warning for constant zero divisor

When the divisor is a literal 0 in source code, loft emits a compile-time warning because a constant-zero divisor is almost certainly a bug: n / 0

7.0.5. Embedding integers in text

```
assert("a{12}b" == "a12b", "Integer in format string");
```

A full expression can appear inside '{...}', not just a variable name.

```
assert("a{1 + 2 * 3}b" == "a7b", "Expression in format string");
```

7.0.6. Number format specifiers

After a ':' inside '{...}' you can control how a number is displayed: #x — hexadecimal with '0x' prefix o — octal b — binary + — always show a sign (+ or -) N — minimum field width (space-padded on the left) 0N — minimum field width (zero-padded on the left)

```
assert("a{1+2+32:#x}b" == "a0x23b", "Hex format with 0x prefix");
assert("{12:o}" == "14", "Octal");
assert("{12:+4}" == " +12", "Sign and width");
assert("{1:03}" == "001", "Zero-padded width");
assert("{42:b}" == "101010", "Binary");
```

Hexadecimal literals in source code accept both lower and upper case digits.

```
assert(0xff == 255, "Lowercase hex literal");
assert(0xFF == 255, "Uppercase hex literal");
assert(0x2A == 42, "Uppercase hex digit");
```

7.0.7. The 'long' type for large numbers

'long' is a 64-bit signed integer — use it when values might exceed 2 billion. Write a long literal by appending 'l': '1l', '100000000000l'. Long values support the same arithmetic and format specifiers as integer. Convert a long back to an integer with 'as integer'.

```
big = 10000000000l * 5l;
assert(big == 50000000000l, "Long arithmetic");
assert("{1l + 1:+4}" == "  +2", "Long in format string");
assert(12l as integer == 12, "Long to integer conversion");
```

7.0.8. Common pitfall: integer overflow

If a 32-bit integer calculation overflows the max value (2 billion), the result wraps around silently. Use 'long' when you expect large values. For example, a score that multiplies two large numbers can silently wrap. Switching to 'long' avoids this: 'score = big_a as long * big_b as long'.

```
}
```

8. Boolean

A boolean value is either ‘true’ or ‘false’. Booleans appear naturally wherever you make a decision: in ‘if’ conditions, loop filters, and comparisons. Loft adds a third state called ‘null’ — meaning “no value” — which behaves like false whenever a boolean is expected.

This design means you rarely need to write a separate null-check: ‘!x’ is true both when x is false and when x is null.

```
fn main() {
```

A comparison produces a boolean result directly. ‘!’ flips a boolean: true ⟶ false, false ⟶ true.

```
    assert(!(3 > 2 + 4), "3 is not greater than 6");
    assert(true as text == "true", "Convert boolean to text");
```

8.0.1. Logical operators: and / or

‘and’ (also ‘&&’) is true only when both sides are true. ‘or’ (also ‘||’) is true when at least one side is true. Both use short-circuit evaluation: the right side is only evaluated if the left side does not already determine the result. This matters when the right side could produce null or has a side effect.

```
    assert(1 > 0 and 2 > 1, "Both conditions true");
    assert(1 > 2 or 3 > 2, "Second condition true");
    assert(!(1 > 2 && 3 > 2), "First false → whole 'and' is false");
    assert(!(1 > 2 || 3 > 4), "Both false → 'or' is false");
```

8.0.2. Null in boolean context

Division by zero (and other failed operations) produce null. Null in a boolean context is treated as false, so ‘!’ is true. This lets you write guard clauses without a separate null-check syntax: if !result { … handle missing value … }

```
    zero = 0;
    assert(!(12 / zero), "null from division-by-zero is false-like");
    assert(12 > 0, "positive integer is true");
```

8.0.3. Bitwise operators

While ‘and’/‘or’ work on true/false values, bitwise operators work on the individual bits of an integer. They are useful for flags, masks, and low-level data manipulation.

‘&’ — keeps only bits set in BOTH operands (AND) ‘|’ — keeps bits set in EITHER operand (OR) ‘<<’ — shift bits left N positions (×2^N) ‘>>’ — shift bits right N positions (÷2^N)

Tip: ‘&’ binds less tightly than comparison operators. Use parentheses when you mix bitwise and comparison expressions in the same condition.

```
    assert((0x0f & 0xa8) == 8, "Bitwise AND: keeps only bits in both");
    assert((0xf0 | 0x0f) == 0xff, "Bitwise OR: combines bits from either");
```

```
    assert(1 << 4 == 16, "Left shift: 1 × 2^4 = 16");
    assert(256 >> 3 == 32, "Right shift: 256 ÷ 2^3 = 32");
```

8.0.4. Negation

‘!’ is the logical NOT operator. It flips any boolean expression.

```
    assert(!false, "not false is true");
    assert(!(1 > 2), "not false comparison is true");
```

8.0.5. Formatting booleans

Booleans can be embedded in a format string like any other value. The ‘^’ alignment specifier centres the value in a field of given width. ‘<’ aligns left, ‘>’ aligns right (right-alignment is the default).

```
    assert("1{true:^7}2" == "1 true  2", "Centred boolean in field of width 7");
    assert("{false}" == "false", "Plain boolean format");
```

8.0.6. Common pitfall: ‘&’ vs ‘and’

‘&’ is bitwise AND on integers; ‘and’ (or ‘&&’) is logical AND on booleans. Writing ‘a & b’ when you mean ‘a and b’ usually compiles but gives wrong results because it operates on the numeric representation of the booleans. Always use ‘and’ / ‘&&’ for boolean logic.

```
    flag_a = 3 > 1;
```

true

```
    flag_b = 4 > 2;
```

true

```
    assert(flag_a and flag_b, "Correct: logical AND on two booleans");
}
```

9. Float

A ‘float’ stores a number with a decimal point, like 3.14 or −0.001. It uses 64-bit precision (the same as f64 in Rust), which gives you about 15 significant digits — enough for scientific work, games, and most real-world maths. When you write a number with a decimal point in Loft, it is automatically a float.

```
fn main() {
```

Write a decimal point in a literal and Loft treats the whole expression as a float. The usual arithmetic operators (+, -, *, /) all work the way you would expect.

```
  assert(1.5 + 0.5 == 2.0, "Float addition");
  assert(3.0 / 2.0 == 1.5, "Float division");
  assert(2.0 * 1.5 == 3.0, "Float multiplication");
```

The ‘f’ suffix selects single-precision (32-bit) floats. Single-precision takes half the memory of a regular float and is common in graphics and audio code where exact decimal values matter less than speed or size.

```
  x = 0.1f + 2 * 1.0f;
  assert(x == 2.1f, "Single precision float");
```

‘as float’ converts an integer to a float so you can mix them in calculations. Putting a float inside ‘{...}’ converts it to text for display. You can also go the other way: ‘“1.5” as float’ parses the text back to a number.

```
  assert(3 as float == 3.0, "Integer to float");
  assert("1.5" as float == 1.5, "Text to float");
  assert("{1.5}" == "1.5", "Float formatted as text");
```

9.0.1. Formatting Floats

Put a colon after the value inside ‘{...}’ to control how it looks. ‘{value:width.precision}’ — ‘width’ is the minimum number of characters printed (padded with spaces on the left); ‘precision’ fixes the number of decimal places. You can use either part on its own: ‘{value:.2}’ just fixes decimal places, ‘{value:5}’ just sets the minimum width.

```
  assert("{1.2:4.2}" == "1.20", "Float with width and precision");
  assert("{334.1:.2}" == "334.10", "Float with precision only");
  assert("{1.4:5}" == "  1.4", "Float with width only");
```

9.0.2. Math Functions

‘PI’ is a built-in constant (approximately 3.14159265358979). Multiplying it by 1000 and rounding gives 3142, which confirms the value is correct.

```
  assert(round(PI * 1000.0) == 3142.0, "PI constant");
```

‘pow(base, exponent)’ raises a number to a power — this is exponentiation. Note: ‘^’ in Loft is bitwise XOR, NOT exponentiation. Always use pow() for powers. ‘log(value, base)’ is the inverse: log(x, b) answers “b to the what power equals x?” Here: 4^5 = 1024, and log(1024, 2) = 10 because 2^10 = 1024.

```
  assert(log(pow(4.0, 5), 2) == 10.0, "log base 2 of 4^5");
```

‘sin’ and ‘cos’ work in radians, not degrees. A full circle is 2*PI radians; a half circle (180 degrees) is PI radians. sin(PI) is theoretically zero but floating-point gives a tiny rounding error, so we use ceil() to snap it up. cos(PI) is exactly −1, so the whole expression ceil(0 + −1 * 1000) lands at −1000.

```
  assert(ceil(sin(PI) + cos(PI) * 1000) == -1000.0, "sin and cos");
```

‘abs()’ returns the absolute value — the distance from zero, always non-negative. Useful any time you care about magnitude but not direction.

```
  assert(abs(-2.5) == 2.5, "Absolute value of float");
```

‘round()’ picks the nearest whole number (0.5 rounds up). ‘ceil()’ always rounds up to the next whole number, even for 2.01. ‘floor()’ always rounds down to the previous whole number, even for 2.99. All three give back a float, not an integer — so 3.0, not 3.

```
  assert(round(2.6) == 3.0, "round up");
  assert(round(2.4) == 2.0, "round down");
  assert(ceil(2.1) == 3.0, "ceil");
  assert(floor(2.9) == 2.0, "floor");
```

Single-precision floats work inside format strings just like regular floats.

```
  assert("a{0.1f + 2 * 1.0f}b" == "a2.1b", "Single-precision format");
}
```

10. Functions

Functions let you give a reusable piece of logic a name so you can call it from multiple places without copying code. This page covers: declaring functions, default argument values, reference parameters (so a function can modify the caller's variable), early return, const parameters, and type-based dispatch.

10.0.1. Declaring Functions

The keyword 'fn' starts a function definition. List parameters as 'name: type' separated by commas. '-> type' declares what the function gives back. The last expression in the body is returned automatically — no 'return' needed there. Default values let callers skip optional arguments.

```
fn greet(name: text, greeting: text = "Hello") -> text {
  greeting + ", " + name + "!"
}
```

10.0.2. Reference Parameters and Defaults

A function with no '-> type' is called for its side effects, not its result. Adding '&' before a parameter type makes it a reference: the function receives a direct link to the caller's variable and can read or write it. Without '&', Loft copies the value in, so changes inside the function stay local. Default values (written '= value' after the type) are substituted when the caller omits that argument.

```
fn add(a:  & integer, b: integer, c: integer = 0) {
  a += b + c;
}
```

10.0.3. Early Return

Use 'return value;' to exit a function before reaching the end. This is handy for guard clauses: handle the special cases first, then write the normal path without extra nesting.

```
fn classify(n: integer) -> text {
  if n < 0 {
    return "negative";
  }
  if n == 0 {
    return "zero";
  }
  "positive"
}
```

10.0.4. Const and Reference Parameters

'const' on a parameter tells the compiler "this function must never change this value." The compiler enforces it — any assignment to a const parameter is a compile error. '&' is the opposite promise: "this function will modify this value." Declaring '&' without actually writing to the parameter is also a compile error. These rules make it easy to read a function signature and know what it does to its inputs.

```
fn scale(a: const integer, factor: const integer) -> integer {
  a * factor
}
```

10.0.5. Type-Based Dispatch

You can define two functions with the same name as long as their parameter types differ. Loft picks the right one at compile time based on the type of the argument you pass.

```
fn describe_int(v: integer) -> text {
  "integer:{v}"
}
```

```
fn describe_text(v: text) -> text {
  "text:{v}"
}
```

10.0.6. Function References

'fn <name>' produces a compile-checked reference to a named function. The result has type 'fn(param_types) -> return_type' and can be:

- stored in a variable,
- called directly with 'f(args)', and
- passed as a parameter to another function.

The compiler resolves the name at the 'fn' expression and reports an error if the function does not exist or the name is not a function.

```
fn double_it(x: integer) -> integer {
  x * 2
}
```

```
fn negate_it(x: integer) -> integer {
  - x
}
```

```
fn apply_fn(f:  fn(integer) -> integer, x: integer) -> integer {
  f(x)
}
```

```
fn main() {
```

Passing both arguments explicitly overrides the default.

```
  assert(greet("World", "Hi") == "Hi, World!", "Explicit argument");
```

Leaving out the second argument causes Loft to use the default "Hello".

```
    assert(greet("World") == "Hello, World!", "Default argument");
```

'add' takes 'a' by reference, so every call updates the original variable 'v'. Watch how v accumulates: 1 -> 3 -> 8.

```
    v = 1;
    add(v, 2);
```

v = 1 + 2 = 3

```
    add(v, 4, 1);
```

v = 3 + 4 + 1 = 8

```
    assert(v == 8, "Reference parameter: {v}");
```

'classify' uses early returns to handle each case separately.

```
    assert(classify(-5) == "negative", "Classify negative");
    assert(classify(0) == "zero", "Classify zero");
    assert(classify(3) == "positive", "Classify positive");
```

'scale' marks both parameters 'const', so the compiler verifies they are not changed.

```
    assert(scale(3, 7) == 21, "scale(3,7)");
```

Loft selects the right function based on the type of the argument.

```
    assert(describe_int(42) == "integer:42", "Integer describe");
    assert(describe_text("hi") == "text:hi", "Text describe");
```

A function reference is stored in a variable with type 'fn(...) -> ...'. Calling it looks exactly like a regular function call.

```
    f = fn double_it;
    assert(f(5) == 10, "fn-ref stored and called: {f(5)}");
```

Pass function references as arguments to higher-order functions.

```
    assert(apply_fn(fn double_it, 7) == 14, "fn-ref as arg (double)");
    assert(apply_fn(fn negate_it, 3) == -3, "fn-ref as arg (negate)");
}
```

11. Vector

A vector is an ordered list of values that can grow and shrink while your program runs. Every element must have the same type — you cannot mix integers and text in one vector. Write a vector literal with square brackets: '[1, 2, 3]'. Under the hood Loft allocates exactly as much memory as needed, so vectors are efficient even when you do not know the size up front.

11.0.1. Higher-order functions: map, filter, reduce

'map', 'filter', and 'reduce' let you transform or summarise a vector by passing a function reference — written 'fn <name>' — as the first argument. map(v, fn f) — apply f to every element; returns a new vector filter(v, fn pred) — keep only elements for which pred returns true reduce(v, fn f, init) — fold all elements into a single value

```
fn triple(x: integer) -> integer {
  x * 3
}
```

```
fn is_even_n(x: integer) -> boolean {
  x % 2 == 0
}
```

```
fn sum_acc(acc: integer, x: integer) -> integer {
  acc + x
}
```

```
fn mul_acc(acc: integer, x: integer) -> integer {
  acc * x
}
```

```
fn main() {
```

Create a vector with a literal and loop over it with 'for'. Two special loop annotations help when you need to know where you are: 'v#first' is true only on the very first iteration — useful for skipping separators. 'v#index' holds the zero-based position of the current element.

```
  x =[1, 3, 6, 9];
  b = "";
  for v in x {
    if !v#first {
      b += " ";
    }
    b += "{v#index}:{v}"
  }
  assert(b == "0:1 1:3 2:6 3:9", "result {b}");
```

'+=' appends another vector to the end of an existing one. You can filter and delete elements in a single pass: Add 'if condition' after 'in vector' to visit only matching elements. Write 'v#remove' inside the

loop body to delete the current element. Here we keep only multiples of 3 by removing everything else. Note: you cannot append to a vector (v += [...]) while iterating over it — that is a compile error because the loop would then visit the new elements too, which could loop forever.

```
x +=[12, 14, 15];
for v in x if v % 3 != 0 {
  v#remove;
}
assert("{x}" == "[3,6,9,12,15]", "result {x}");
```

11.0.2. Clearing

'v.clear()' removes all elements, setting the length to 0. The underlying storage is kept so appending afterwards is efficient.

```
x.clear();
assert(x.len() == 0, "clear empties the vector");
x += [99];
assert(x[0] == 99, "append after clear works");
```

11.0.3. Slicing

A slice gives you a window into part of a vector without copying it. 'v[a..b]' contains elements at positions a, a+1, ..., b-1 (b is excluded). 'v[a..]' goes from position a to the very last element. 'v[..b]' goes from the beginning up to (but not including) position b.

```
pows =[1, 2, 4, 8, 16];
assert("{pows[1..3]}" == "[2,4]", "Sub-vector");
assert("{pows[3..]}" == "[8,16]", "Open-ended sub-vector");
assert("{pows[..3]}" == "[1,2,4]", "Open-start sub-vector");
```

Accessing an index that does not exist is safe: Loft returns null instead of crashing. You can check for null with '!' (logical not) because null is falsy.

```
assert(!pows[10], "Out-of-bounds access returns null");
```

11.0.4. Comprehensions

A comprehension is a concise way to build a new vector from a formula. Syntax: '[for variable in range { expression }]' Add 'if condition' to include only elements that satisfy the condition. This is much shorter than creating an empty vector and appending in a loop.

```
evens =[for n in 1..10 if n % 2 == 0 {
  n
}];
assert("{evens}" == "[2,4,6,8]", "Filtered comprehension");
doubled =[for n in 1..6 {
  n * 2
```

```
  }];
  assert("{doubled}" == "[2,4,6,8,10]", "Doubled comprehension");
```

Embedding a for loop directly inside a format string '{...}' produces a formatted list. The format specifier after ':' is applied to every element in the result. ':02' means "at least 2 digits wide, padded with zeros on the left".

```
  assert("{for n in 1..7 {n*2}:02}" == "[02,04,06,08,10,12]", "Formatted vector
loop");
```

11.0.5. Reverse Iteration

Wrap a range in 'rev()' to step through elements from the last index to the first. 'rev(0..=3)' covers indices 0, 1, 2, 3 — the '=' makes the upper end inclusive. Here we visit pows[3]=8, pows[2]=4, pows[1]=2, pows[0]=1, building 8421 digit by digit.

```
  c = 0;
  for e in pows[rev(0..=3)] {
    c = c * 10 + e;
  }
  assert(c == 8421, "Reverse sub-vector iteration");
```

You can fill a vector with many copies of the same value using '; count' syntax: [SomeStruct { field: value }; 16] This creates 16 identical copies in one expression. See 08-struct.loft for examples. To append to a vector inside a function and have the caller see the change, mark the parameter with '&': fn append_one(v: &vector<integer>, x: integer) { v += [x]; } Without '&', appending is local to the function and the caller's vector stays the same. Mutations to existing elements (e.g. v[0] = 99) are always visible to the caller even without '&', because the vector's storage is shared.

11.0.6. Higher-order functions

'map' applies a function to every element and returns a new vector.

```
  nums =[1, 2, 3, 4, 5];
  tripled = map(nums,  fn triple);
  assert("{tripled}" == "[3,6,9,12,15]", "map triple: {tripled}");
```

'filter' keeps only elements for which the predicate returns true.

```
  evens2 = filter(nums,  fn is_even_n);
  assert("{evens2}" == "[2,4]", "filter evens: {evens2}");
```

'reduce' folds all elements into a single value, starting from an initial accumulator. Argument order: reduce(vector, initial_value, fn combiner).

```
  total = reduce(nums, 0,  fn sum_acc);
  assert(total == 15, "reduce sum: {total}");
```

```
    product = reduce(nums, 1,  fn mul_acc);
    assert(product == 120, "reduce product: {product}");
```

You can chain these calls: filter first, then map, then reduce.

```
    result = reduce(map(filter(nums,  fn is_even_n),  fn triple), 0,  fn sum_acc);
    assert(result == 18, "filter+map+reduce: {result}");
}
```

12. Structs

A struct groups related values under one name. Instead of keeping a product's name, price, and stock count in three separate variables that can drift out of sync, you bundle them together into a Product struct. Each piece of data inside the struct is called a field. You read and write fields using dot notation: 'product.price'. Here is a simple product record. All four fields are plain integers or text.

```
struct Product {
  name: text,
  price: integer,
  stock: integer
}
```

12.0.1. Field Constraints

You can restrict what values a field may hold. 'limit(min, max)' rejects any value outside that range at runtime. 'not null' declares that zero is a meaningful value — without it, zero is treated as "no value" (null), which can cause surprising behaviour. Fields you omit in a constructor receive zero (null for nullable fields) by default. Here all three colour channels can be zero (black is a valid colour).

```
struct Colour {
  r: integer limit(0, 255) not null,
  g: integer limit(0, 255) not null,
  b: integer limit(0, 255) not null
}
```

12.0.2. Methods

A method is a function whose first parameter is named 'self'. Loft uses the type of 'self' to decide which struct the method belongs to. Call it with dot notation: 'c.to_hex()'. A method can read fields via 'self.field' and return any type.

```
fn to_hex(self: Colour) -> integer {
  self.r * 0x10000 + self.g * 0x100 + self.b
}
```

A method can also return a completely new struct value. Write '-> StructName' as the return type and construct the value in the body. The original struct is not modified — the caller gets a brand-new copy.

```
fn dimmed(self: Colour) -> Colour {
  Colour {r: self.r / 2, g: self.g / 2, b: self.b / 2 }
}
```

12.0.3. Computed Fields

A field can calculate its value from other fields in the same struct. Write '= expression' after the type to set a default evaluated at construction time. Inside that expression, '$' refers to the struct being built. The 'name_length' field is filled automatically whenever you create an Item.

```
struct Item {
  name: text,
  name_length: integer = len($.name)
}
```

12.0.4. Storing many structs in a vector

'Area' uses compact unsigned integer types (u16, u8) to keep each record small. This matters when you need millions of tiles in a game map.

```
struct Area {
  height: u16,
  terrain: u8,
  water: u8,
  direction: u8
}
```

```
fn main() {
```

Constructing a struct: name the fields you want to set. Fields you leave out default to zero (or null for nullable fields).

```
  apple = Product {name: "Apple", price: 120, stock: 50 };
  assert(apple.price == 120, "price field: {apple.price}");
  assert(apple.name == "Apple", "name field: {apple.name}");
```

You can read and write individual fields after construction.

```
  apple.stock -= 1;
  assert(apple.stock == 49, "stock after one sale: {apple.stock}");
```

A field omitted from the constructor gets zero as its default.

```
  col = Colour {r: 128, b: 128 };
  assert(col.g == 0, "omitted green channel defaults to zero");
```

Formatting a struct shows all fields compactly.

```
  assert("{col}" == "{{r:128,g:0,b:128}}", "Struct compact formatting");
```

':j' produces JSON output.

```
  assert("{col:j}" == "{{\"r\":128,\"g\":0,\"b\":128}}", "JSON format");
```

Call a method on a variable using dot notation.

```
  purple = Colour {r: 128, b: 128 };
  assert("{purple.to_hex():x}" == "800080", "hex method result");
```

You can call a method directly on a constructor expression.

```
  assert(Colour {r: 255, g: 0, b: 0 }.to_hex() == 0xff0000, "method on
constructor");
```

‘dimmed’ returns a new Colour with each channel halved. The original variable is unchanged.

```
  dark = purple.dimmed();
  assert(dark.r == 64, "dimmed r: {dark.r}");
  assert(dark.b == 64, "dimmed b: {dark.b}");
  assert(dark.g == 0, "dimmed g: {dark.g}");
```

Computed fields are filled automatically at construction time.

```
  it = Item {name: "hello" };
  assert(it.name_length == 5, "computed name_length: {it.name_length}");
```

Fill a vector with copies of the same struct using ‘; count’ syntax. This creates 16 Area records, all with the same initial values.

```
  map =[Area {height: 0, terrain: 1, water: 1, direction: 1 };
  16];
  assert("{map[3]}" == "{{height:0,terrain:1,water:1,direction:1}}", "tile
record");
  map[3].height = 200;
  assert(map[3].height == 200, "individual tile update");
```

12.0.5. sizeof

‘sizeof(Type)’ returns the packed byte size used when the type is stored as a struct field or vector element. Range-constrained integer types like u8 and u16 report their packed size, not the 4-byte stack slot size.

```
  assert(sizeof(integer) == 4, "integer: 4 bytes");
  assert(sizeof(u8) == 1, "u8: 1 byte (packed)");
  assert(sizeof(u16) == 2, "u16: 2 bytes (packed)");
  assert(sizeof(Colour) == 3, "Colour: 3 × u8 = 3 bytes");
  assert(sizeof(Area) == 5, "Area: u16 + 3 × u8 = 5 bytes");
}
```

13. Enums

An enum defines a fixed set of named choices. Instead of using raw numbers like 0 = north, 1 = east — which nobody can read six months later — you give each option a name. The compiler then checks every comparison and conversion, so a typo becomes a compile error instead of a silent wrong answer. Plain enums are just names that can be compared, ordered, and converted to and from text. Their order matches their declaration order.

```
enum Direction {
  North,
  East,
  South,
  West
}
```

13.0.1. Enums that carry data

Sometimes different choices have different shapes of data. A 'Circle' needs one number (radius); a 'Rect' needs two (width and height). Struct enums let each variant carry its own fields, keeping all the shape kinds in one type while still letting you work with each variant naturally.

```
enum Shape {
  Circle {radius: float },
  Rect {width: float, height: float }
}
```

13.0.2. Polymorphic methods

Write the same function name for each variant, each taking 'self' of that variant's type. Loft picks the right version at runtime based on the actual variant — this is called polymorphic dispatch. It lets you call shape.area() without knowing which kind of shape it is.

```
fn area(self: Circle) -> float {
  PI * pow(self.radius, 2)
}
```

```
fn area(self: Rect) -> float {
  self.width * self.height
}
```

Polymorphic methods can return text just as well as numbers. Each variant can produce its own human-readable description.

```
fn describe(self: Circle) -> text {
  "circle with radius {self.radius}"
}
```

```
fn describe(self: Rect) -> text {
  "rect {self.width} by {self.height}"
}
```

13.0.3. Enum methods

Plain enum variants can also have methods. The 'self' parameter carries the current direction value, and the method can return any type. Here 'opposite()' flips North↔South and East↔West.

```
fn opposite(self: Direction) -> Direction {
  if self == North {
    South
  } else if self == South {
    North
  } else if self == East {
    West
  } else {
    East
  }
}
```

```
fn main() {
```

Plain enum values are ordered by declaration position. North comes first (smallest), West comes last (greatest).

```
  d = East;
  assert(d == East, "Enum equality");
  assert(d != North, "Enum inequality");
  assert(d > North, "East declared after North, so it is greater");
  assert(West > South, "West declared last, so it is greatest");
```

Convert between text and enum in both directions.

```
  assert("{d}" == "East", "Format plain enum as text");
  assert("West" as Direction == West, "Parse text to enum");
```

Use a plain enum method like any other method.

```
  assert(d.opposite() == West, "opposite of East is West");
  assert(North.opposite() == South, "opposite of North is South");
```

Struct enum variants are constructed exactly like regular structs.

```
  c = Circle {radius: 1.0 };
  assert(c.radius == 1.0, "Circle radius field");
  r = Rect {width: 4.0, height: 5.0 };
  assert(r.width * r.height == 20.0, "Rect manual area");
```

Calling area() on a Circle runs the Circle version; on a Rect it runs the Rect version — chosen automatically at runtime.

```
  assert(round(c.area() * 1000) == round(PI * 1000), "Circle area = PI*r^2");
  assert(r.area() == 20.0, "Rect area = width*height");
```

describe() uses format strings with field access inside each variant's method.

```
  assert(c.describe() == "circle with radius 1", "describe circle:
{c.describe()}");
  assert(r.describe() == "rect 4 by 5", "describe rect: {r.describe()}");
```

13.0.4. Stubs for missing implementations

If a variant intentionally has no implementation of a method, the compiler emits a warning. Provide an empty-body stub to silence it: fn area(self: SomeVariant) -> float { } A stub returns null at runtime and suppresses the warning.

13.0.5. Match expressions on enums

Match picks a code path based on the active variant. You must handle every variant, or include a _ wildcard arm that catches the rest.

```
  axis = match d {
    North | South => "vertical",
    East | West => "horizontal"
  };
  assert(axis == "horizontal", "or-pattern: East matches East|West");
```

When a variant has fields, name them inside braces to use them in the arm body.

```
  label = match c {
    Circle { radius } => "r={radius}",
    Rect { width, height } => "{width}x{height}"
  };
  assert(label == "r=1", "field destructuring in match arm");
```

13.0.6. Guard clauses

An arm can have an `if` guard after the pattern. If the guard fails, matching falls through to the next arm. Because the guard can fail, a guarded arm alone does not prove the variant is handled — you still need a wildcard _ or an unguarded arm for that variant.

```
  area = match c {
    Circle { radius } if radius > 0.0 => PI * radius * radius,
    _ => 0.0
  };
  assert(round(area * 1000) == round(PI * 1000), "guarded match on Circle");
```

13.0.7. Scalar match

Match also works on integers, text, floats, booleans, and characters. Arms can be literals, ranges, `null`, or _.

```
  grade = match 85 {
    90..=100 => "A",
    80..90   => "B",
    _        => "C"
  };
  assert(grade == "B", "range pattern 80..90 matches 85");
```

Or-patterns work on scalars too.

```
  kind = match 2 {
    1 | 2 | 3 => "low",
    _         => "high"
  };
  assert(kind == "low", "scalar or-pattern");
```

A `null` pattern matches when the value is absent (e.g. division by zero).

```
  zero = 0;
  check = match 1 / zero {
    null => "absent",
    _    => "present"
  };
  assert(check == "absent", "null pattern matches div-by-zero");
```

Character literals work in match arms.

```
  vowel = match 'e' {
    'a' | 'e' | 'i' | 'o' | 'u' => true,
    _ => false
  };
  assert(vowel, "character or-pattern");
}
```

14. Sorted

A ‘sorted’ collection holds records in order by one or more key fields. You can look up a record by key instantly and iterate all records in key order. The collection stays sorted as you add new elements — no manual sorting needed. Declare the key fields inside angle brackets: ‘field’ sorts ascending, ‘-field’ descending.

```
struct Elm {
  key: text,
  value: integer
}
```

```
struct Db {
  map: sorted < Elm[-key] >
}
```

```
fn main() {
```

14.0.1. Adding Elements

You initialise a sorted collection the same way as a vector. The elements are automatically placed in the right position as you add them.

```
  db = Db {map: [
  Elm {key: "One", value: 1 },
  Elm {key: "Two", value: 2 },
  Elm {key: "Three", value: 3 },
  Elm {key: "Four", value: 4 }
  ] };
```

14.0.2. Looking Up by Key

Use square brackets with the key value to find a record instantly. If the key is not present the result is null — you can check it with ‘!’.

```
  assert(db.map["Two"].value == 2, "Key lookup: Two");
  assert(db.map["Three"].value == 3, "Key lookup: Three");
  assert(!db.map["Five"], "Missing key returns null");
  assert(!db.map[null], "Null key returns null");
```

Append new elements with ‘+=’; they are placed in the correct sorted position.

```
  db.map +=[Elm {key: "Zero", value: 0 }];
  assert(db.map["Zero"].value == 0, "Newly added element");
```

14.0.3. Iterating in Order

A ‘for’ loop over a sorted collection visits elements in key order. Here the key is ‘-key’ (descending text), so the order is: Zero, Two, Three, One, Four.

```
sum = 0;
for v in db.map {
  sum = sum * 10 + v.value;
}
```

Zero(0), Two(2), Three(3), One(1), Four(4) ⟶ 0*10+2=2, *10+3=23, *10+1=231, *10+4=2314

```
assert(sum == 2314, "Sorted iteration total: {sum}");
```

14.0.4. Iterating in Reverse

Wrap the collection in rev() to visit elements from last to first key order. Here the collection is sorted by '-key' (descending text), so the stored order is Zero, Two, Three, One, Four. rev() visits them in the opposite order.

```
rev_sum = 0;
for v in rev(db.map) {
  rev_sum = rev_sum * 10 + v.value;
}
```

Four(4), One(1), Three(3), Two(2), Zero(0) ⟶ 4*10+1=41, *10+3=413, *10+2=4132, *10+0=41320

```
assert(rev_sum == 41320, "Reverse iteration total: {rev_sum}");
```

14.0.5. Loop Helpers: #first, #count, and #remove

'v#first' is true for the very first element visited. 'v#count' is a running index starting at 0. 'v#remove' removes the current element while iterating.

```
first_key = "";
labels = "";
for v in db.map {
  if v#first {
    first_key = v.key
  }
  if !v#first {
    labels += ","
  }
  labels += "{v#count}:{v.key}"
}
assert(first_key == "Zero", "First element: {first_key}");
assert(labels == "0:Zero,1:Two,2:Three,3:One,4:Four", "Labels: {labels}");
```

Remove elements with value > 2 while iterating.

```
for v in db.map if v.value > 2 {
  v#remove
}
sum2 = 0;
```

```
  for v in db.map {
    sum2 += v.value
  }
```

Remaining: Zero(0), Two(2), One(1) ⟶ sum = 3

```
  assert(sum2 == 3, "Sum after remove: {sum2}");
```

14.0.6. Removing by Key

Assigning null to a sorted subscript removes the element with that key. Removing a key that is not present is a safe no-op.

```
  db2 = Db {map: [
  Elm {key: "A", value: 10 },
  Elm {key: "B", value: 20 },
  Elm {key: "C", value: 30 }
  ] };
  db2.map["B"] = null;
  assert(!db2.map["B"], "B was removed");
  assert(db2.map["A"].value == 10, "A still present");
  assert(db2.map["C"].value == 30, "C still present");
  db2.map["missing"] = null;
```

no-op; does not panic

```
  sum3 = 0;
  for v in db2.map {
    sum3 += v.value
  }
  assert(sum3 == 40, "Sum after key removal: {sum3}");
```

Note: #index is not available on sorted — use #count for a sequential counter.

```
}
```

filling and finding values

15. Index

An ‘index’ lets you find records instantly by key and iterate over ranges of keys in order. It supports multi-part keys: you can sort by a primary key and break ties with a secondary key. Declare the key fields inside angle brackets: ‘field’ sorts ascending, ‘-field’ descending. Note: each record can only belong to one index at a time.

```
struct Elm {
  nr: integer,
  key: text,
  value: integer
}
```

```
struct Db {
  map: index < Elm[nr, -key] >
}
```

```
fn main() {
```

15.0.1. Adding and Looking Up Records

Fill the index just like a vector. Elements are automatically kept in key order. This index is sorted first by ‘nr’ (ascending), then by ‘key’ (descending) for ties.

```
  db = Db {map: [
  Elm {nr: 101, key: "One", value: 1 },
  Elm {nr: 92, key: "Two", value: 2 },
  Elm {nr: 83, key: "Three", value: 3 },
  Elm {nr: 83, key: "Four", value: 4 },
  Elm {nr: 83, key: "Five", value: 5 },
  Elm {nr: 63, key: "Six", value: 6 }
  ] };
```

Provide all key fields together inside brackets to find a record. Supply fewer fields to match all records that share a prefix.

```
  assert(db.map[101, "One"].value == 1, "Key lookup");
  assert(!db.map[12, ""], "Missing key returns null");
  assert(!db.map[83, "One"], "Wrong secondary key returns null");
```

15.0.2. Iterating in Order

A ‘for’ loop visits all elements in key order.

```
  total = 0;
  for r in db.map {
    total += r.value;
  }
  assert(total == 21, "Sum of all values: {total}");
```

15.0.3. Range Queries

Provide a range in the first key position to visit only the matching slice. '[83..92, "Two"]' means: nr from 83 up to (not including) 92, and key from "Two" down (since the key is sorted descending, "Two" is the start of that descending segment).

```
  sum = 0;
  for v in db.map[83..92, "Two"] {
    sum = sum * 10 + v.value;
  }
```

Elements matched: (83, Three, 3), (83, Four, 4), (83, Five, 5)

```
  assert(sum == 345, "Range iteration result: {sum}");
```

15.0.4. Loop Helpers: #first and #count

'r#first' is true for the very first element visited. 'r#count' is a running counter starting at 0. Note: #index is not meaningful on index collections — use #count instead.

```
  first_nr = 0;
  count_total = 0;
  for r in db.map {
    if r#first {
      first_nr = r.nr
    }
    count_total = r#count
  }
  assert(first_nr == 63, "First element by key: {first_nr}");
  assert(count_total == 5, "Total elements: {count_total}");
```

15.0.5. Removing by Key

Assigning null to an index subscript removes the element with that exact key combination. Removing a key that is not present is a safe no-op.

```
  db.map[92, "Two"] = null;
  assert(!db.map[92, "Two"], "element (92, Two) was removed");
  assert(db.map[101, "One"].value == 1, "element (101, One) still present");
  db.map[999, "Z"] = null;
```

no-op; does not panic

```
  total2 = 0;
  for r in db.map {
    total2 += r.value
  }
  assert(total2 == 19, "Sum after key removal: {total2}");
}
```

16. Hash

A ‘hash’ lets you find a record by its key in constant time, regardless of how many records you have. Unlike ‘sorted’ and ‘index’, a hash has no meaningful order — you use it purely for fast lookups. List the key fields in brackets: ‘hash<Type[field]>’. Combine a hash with a vector when you want both fast lookup and a stable iteration order.

```
struct Keyword {
  name: text
}
```

```
struct Data {
  h: hash < Keyword[name] >
}
```

16.0.1. Combining Hash and Vector

Pairing a hash with a vector on the same record type gives you the best of both worlds: the hash for instant lookup by key, and the vector for iterating in insertion order.

```
struct Count {
  t: text,
  v: integer
}
```

```
struct Counting {
  entries: vector < Count >,
  lookup: hash < Count[t] >
}
```

```
fn fill(c: Counting) {
  c.entries =[
 {t: "One", v: 1 },
 {t: "Two", v: 2 },
 {t: "Three", v: 3 },
 {t: "Four", v: 4 }
  ]
}
```

```
fn main() {
```

16.0.2. Basic Lookup

Fill a hash the same way you fill a vector. Look up by key with square brackets. A found record is truthy; a missing key returns null.

```
  d = Data { };
  d.h =[ {name: "one" }, {name: "two" }];
```

```
  d.h +=[ {name: "three" }, {name: "four" }];
  assert(d.h["three"], "Key exists");
  assert(!d.h["None"], "Missing key returns null");
```

16.0.3. Hash + Vector Together

Here 'entries' is the vector (keeps insertion order) and 'lookup' is the hash (fast access). Both fields point to the same records — adding to one automatically updates the other.

```
  c = Counting { };
  fill(c);
  assert(c.lookup["Three"].v == 3, "Hash lookup: Three");
  assert(c.lookup["One"].v == 1, "Hash lookup: One");
  assert(!c.lookup["Five"], "Missing key returns null");
```

Iterate in insertion order via the vector; look up by name via the hash. The vector also supports #first, #count, and #remove inside the loop.

```
  add = 0;
  for item in c.entries {
    add += item.v;
  }
  assert(add == 10, "Sum via vector iteration: {add}");
```

16.0.4. Removing a Key

Assigning null to a hash subscript removes that element. Removing a key that is not present is a safe no-op.

```
  d.h["three"] = null;
  assert(!d.h["three"], "three was removed");
  assert(d.h["one"], "one still present");
  d.h["missing"] = null;
```

no-op; does not panic

16.0.5. Why you cannot iterate a hash directly

A hash has no stable element order — hash bucket positions depend on key hashes and the internal load factor, so there is no meaningful #index or sequential position. 'for item in c.lookup' is therefore a compile error. Always pair a hash with a vector when you need both fast lookup and ordered iteration.

```
}
```

17. File

A file handle lets you read and write files without worrying about when the OS opens or closes them. The file opens on the first read or write and closes automatically when the handle goes out of scope.

17.0.1. Inspecting the File System

`file(path)` creates a File handle without opening anything yet. `f\#format` tells you what kind of path you are looking at: TextFile, LittleEndian, BigEndian, Directory, or NotExists. `lines()` reads a text file and returns it as a vector of lines. `exists(path)` is a convenience shorthand for checking that a path is not NotExists. `delete(path)` removes a file and returns false if it was not there. Clean up any leftover files from a previous interrupted run.

```
fn cleanup() {
  delete("test.bin");
  delete("test2.bin");
  delete("buffer.bin");
}
```

```
struct Buffer {
  size: i32,
  data: vector < single >
}
```

```
fn main() {
  cleanup();
```

Asking for the format of a directory path returns Directory, not a file format. `lines()` reads the named text file and splits it on newlines.

```
  ex = file("tests/example");
  assert(ex#format == Directory, "example is a directory");
  c = file("tests/example/config/terrain.txt").lines();
  assert(c[1] == "    terrain = [", "Line was '{c[1]}'");
```

`exists` and `delete` are safe to call when the file is absent. `delete` returns false rather than crashing when nothing is there.

```
  assert(!exists("test.bin"), "File should not exist before the test.");
  assert(!delete("nonexistent_xyz.bin"), "delete on missing file returns false");
```

17.0.2. Writing a Text or Binary File

Wrapping the file handle in a block ensures the file closes the moment the block ends. Set `f\#format` to LittleEndian or BigEndian before writing binary data. Use `f += value` to append the raw bytes of any scalar value (u8, u16, i32, long, single, float, or text).

```
{f = file("test.bin");
 assert(f#format == NotExists, "File should not exist yet.");
 f#format = BigEndian;
 f += 0 as u8;
 f += 1 as u8;
 f += 0x203 as u16;
 f += 0x4050607;
 f += 0x8090a0b0c0d0e0fl;
```

`f\#size` returns the total number of bytes written so far.

```
  assert(f#size == 16l, "Should have written 16 bytes.");
```

Text is written as raw UTF-8 bytes with no length prefix.

```
  f += "Hello world!";
  assert(f#size == 28l, "Size should be 28 bytes (16 + 12).");
}  // The file closes when the handle goes out of scope.
```

`move(src, dst)` renames a file. It refuses if the destination already exists or if either path would leave the project directory.

```
  assert(exists("test.bin"), "File should exist after writing.");
  assert(!move("test.bin", "../test.bin"), "Should refuse to move outside the
project.");
  assert(move("test.bin", "test2.bin"), "Could not move the file.");
```

17.0.3. Reading Back What You Wrote

When you open an existing file the default format is TextFile. Set `f\#format` to match the format used when writing before you read any bytes. `f\#read(n) as T` reads exactly n bytes and interprets them as type T. `f\#index` is the byte offset where the last read started. `f\#next` is the current read position; assign to it to seek anywhere.

```
{f = file("test2.bin");
 assert(f#format == TextFile, "The default format is TextFile.");
 f#format = LittleEndian;
```

The file was written as BigEndian, so reading the same 4 bytes as LittleEndian produces a byte-swapped value — that is intentional here and confirms that the bytes were stored in the order you chose.

```
  v = f#read(4) as i32;
  assert(v == 0x3020100, "BigEndian bytes 0..3 read as LittleEndian i32.");
  assert(f#index == 0l, "Last read started at byte 0.");
  assert(f#next == 4l, "Next read starts at byte 4.");
```

Seek to byte 16 to skip past the integers and read the text directly.

```
  f#next = 16l;
  s = f#read(5) as text;
  assert(s == "Hello", "Partial text read from offset 16.");
  assert(f#index == 16l, "Last read started at byte 16.");
  assert(f#next == 21l, "Next position after 5-byte text read.");
```

Requesting more bytes than remain simply returns whatever is left.

```
  rest = f#read(100) as text;
  assert(rest == " world!", "Read continues to end of file.");
  assert(f#next == f#size, "Position should be at end of file.");
```

You can seek back to any position and re-read.

```
  f#next = 0l;
  assert(f#read(4) as i32 == 0x3020100, "Seek-and-reread matches original.");
}
  assert(delete("test2.bin"), "Could not remove the test file.");
```

17.0.4. Working with Vectors and Struct Data

`f += vector\<T\>` writes every element in sequence as raw bytes, which is the fastest way to dump a whole collection to disk. Setting `f\#size = n` truncates or zero-extends the file to exactly n bytes — useful for discarding the tail after you have finished writing.

```
{f = file("buffer.bin");
 f#format = LittleEndian;
 ints =[1, 2, 3, 4];
 f += ints;
 assert(f#size == 16l, "Four i32 values = 16 bytes");
```

Truncate to the first two integers.

```
  f#size = 8l;
  assert(f#size == 8l, "Truncated to 8 bytes");
}
  assert(delete("buffer.bin"), "Could not remove buffer.bin after vector write
test.");
```

Write a count followed by individual float values. `sizeof(T)` returns the byte width of a type, so you can compute the correct read length without hard-coding magic numbers.

```
{buf = file("buffer.bin");
 buf#format = LittleEndian;
 buf += 4 as i32;
 buf += 1.1f;
 buf += 1.2f;
 buf += 2.1f;
```

```
  buf += 2.2f;
 }
```

Read the count, then use it to read exactly that many floats directly into a struct field. This pattern lets you serialise and deserialise structs with variable-length data cleanly.

```
 {b = Buffer { };
  f = file("buffer.bin");
  f#format = LittleEndian;
  n = f#read(4) as i32;
  b.data = f#read(n * sizeof(single));
 }
  assert(delete("buffer.bin"), "Could not remove buffer.bin.");
}
```

18. Image

Loft has a built-in Image type for loading PNG files and inspecting their pixels. Loading a PNG takes one function call; after that you can read every pixel, check the dimensions, and iterate over the whole image with a for loop.

18.0.1. Loading an Image

Pass a File handle to the 'png()' function to load a PNG into memory. The entire file is read and decoded at this point. After the call you can access pixels as many times as you like with no further I/O.

img = png(file("photo.png"))

If the file does not exist or is not a valid PNG, 'img' will be null and you can check for that with '!' before proceeding.

18.0.2. Checking Dimensions

Once loaded, 'img#width' and 'img#height' give you the pixel dimensions. These are read-only attributes — you cannot resize an image by writing to them.

w = img#width h = img#height println("image is {w} x {h} pixels — {w * h} pixels total")

18.0.3. Accessing Individual Pixels

Index the image with two integer coordinates [x, y] to get a pixel value. x = 0 is the left column; y = 0 is the top row. Each pixel carries four channels, all in the range 0–255:

px = img[x, y] r = px#red

A pixel from outside the image bounds is null.

18.0.4. Iterating Over All Pixels

A 'for' loop over an image visits every pixel from top-left to bottom-right, row by row. This is the easiest way to scan or analyse the whole image.

bright_red = 0; for px in img { if px#red > 200 and px#green < 50 and px#blue < 50 { bright_red += 1; } } println("found {bright_red} bright-red pixels")

18.0.5. Practical example: average brightness

Here is how you would compute the average brightness of an image. Brightness is often approximated as (r + g + b) / 3, or the more perceptually accurate formula (0.299*r + 0.587*g + 0.114*b).

total = 0; count = 0; for px in img { total += px#red + px#green + px#blue; count += 1; } if count > 0 { avg = total / (count * 3); println("average brightness: {avg} / 255") } Note: PNG test files are not included in the standard test suite, so the examples above are shown as comments rather than runnable assertions. Paste them into your own project and point them at a real PNG file to try.

```
fn main() {
}
```

19. Lexer

The lexer library breaks a text into tokens so your program can understand its structure. Tokens are the smallest meaningful pieces: numbers, identifiers, operators, and string literals. You tell the lexer which multi-character sequences count as single tokens and which words are reserved, and it handles the rest.

19.0.1. Setting Up the Lexer

Create a `lexer::Lexer` and register your language's rules before you parse anything. `set_tokens` ensures operators like `+=` or `\>\>` are scanned as one token instead of two separate characters. `set_keywords` prevents reserved words from being returned as plain identifiers — the lexer will report them exactly as written so your parser can treat them specially.

```
use lexer;
fn main() {
  l = lexer::Lexer { };
  l.set_tokens(["+=", "*=", "-=", "<=", ">=", "!=", "==", ">>", "<<", "->", "=>",
">>>", "..", "..=", "&&", "||"]);
  l.set_keywords(["for", "in", "if", "else", "fn", "pub", "use", "struct",
"enum", "match", "and", "or"]);
```

19.0.2. Reading Tokens

`parse_string(name, source)` feeds source text into the lexer. The name is used in error messages and position reports. After that, call the typed reader functions one by one to consume tokens in order.

`int()` consumes and returns the next integer token, or null if the current token is not an integer. `long_int()` does the same for integers suffixed with `l`. `matches(s)` consumes the next token only when it equals s and returns true; otherwise it leaves the token in place and returns false. `peek()` returns the next token as text without consuming it. `position()` returns the current location as `file:line:col`.

```
  l.parse_string("Tokens", "12 += -2 * 3l >> 4");
  assert(l.int() == 12, "Integer");
  assert(!l.matches("+"), "Incorrect plus");
  assert(l.peek() != "+", "Incorrect plus");
  assert(l.matches("+="), "Incorrect plus_is");
  assert(l.int() == -2, "Second integer");
  assert(l.matches("*"), "Incorrect multiply");
  assert(l.int() != 3, "Third number");
  assert(l.long_int() == 3, "Incorrect long");
  assert(l.position() == "Tokens:1:15", "Incorrect position {l.position()}");
  assert(!l.matches(">"), "Incorrect higher");
  assert(l.matches(">>"), "Incorrect logical shift");
  assert(l.position() == "Tokens:1:18", "Incorrect position {l.position()}");
```

19.0.3. String Literals and Comments

`constant_text()` reads a double-quoted string literal and unescapes any escape sequences inside it. `constant_character()` reads a single-quoted character literal and returns it as text.

```
  l.parse_string("Texts", "\"123\" + '4'");
  assert(l.constant_text() == "123", "Incorrect text literal");
  assert(l.matches("+"), "Incorrect add");
  assert(l.constant_character() == "123", "Incorrect text literal");
```

The lexer collects `//` comments automatically as it scans. You do not need to handle them yourself. `last_comment()` returns the accumulated comment text since the last consumed token. When multiple comment lines appear in a row they are joined with newlines into a single string. `comment_behind()` is true when the comment appeared on the same line as the preceding token rather than on its own line above. `is_finished()` returns true once every token has been consumed.

```
  l.parse_string("Comments", "// starting comments\n123 // same line comment\n//
extra comment\n4");
  assert(!l.comment_behind(), "Initial comment not behind");
  assert(l.last_comment() == "starting comments", "Initial comment");
  assert(l.int() == 123, "Content integer");
  assert(l.comment_behind(), "Second comment is behind");
  assert(l.last_comment() == "same line comment\nextra comment", "Second
comment");
  assert(!l.is_finished(), "Not Ready");
  assert(l.int() == 4, "Second integer");
  assert(l.last_comment() == "", "No remaining comment");
  assert(l.is_finished(), "Ready");
```

19.0.4. Embedded Format Expressions

Loft string literals can embed expressions with `{expr}`. The lexer exposes a protocol that lets you parse these yourself. When `constant_text()` reaches a `{`, it returns the literal text before it and sets `is_formatting()` to true. At that point call `set_formatting(false)` and parse the embedded expression normally using the usual token readers. When the expression is done, call `set_formatting(true)` and consume the closing `}}`. Then `constant_text()` continues with the next segment of the string.

```
  l.parse_string("Formatting", "\"abc{{12 + 34}}def\"");
  assert(l.constant_text() == "abc", "Before formatting");
  assert(l.is_formatting(), "Formatting");
  l.set_formatting(false);
  assert(l.int() == 12, "First integer");
  assert(l.matches("+"), "Incorrect plus");
  assert(l.int() == 34, "Second integer");
  l.set_formatting(true);
  assert(l.matches("}}"), "Incorrect closing brace");
  assert(l.constant_text() == "def", "After formatting");
  assert(!l.is_formatting(), "Formatting");
}
```

20. Parser

The ‘parser’ library lets a Loft program read and validate other Loft source code at runtime. This is useful when you want to: * validate configuration files written in the Loft syntax * build tools that inspect or transform Loft source * write a test that checks whether a generated code snippet is syntactically correct

‘parse(name, source)’ is the single entry point. It takes a display name (used in error messages) and a Loft source string. The name does not need to correspond to a real file — it is only used when reporting parse errors.

If the source is invalid, parse() emits a diagnostic error and the call returns without producing a value.

20.0.1. What the parser understands

The parser handles the complete Loft grammar: * ‘struct Name { field: type [= default] }’ — named-field aggregates * ‘enum Name { Variant [{ field: type }] }’ — tagged unions with optional fields * ‘fn name(params) [-> type] { body }’ — functions with a block body * ‘fn name(params) [-> type]; #rust “template”’ — operator templates backed by Rust * ‘use module;’ — module imports * Expressions: binary operators with precedence, function calls, field access, index expressions, if/else, for loops, blocks, and formatted string literals * Type expressions: plain names, generic types like ‘vector<T>’, keyed collections (sorted/hash/index), and integer ranges with ‘limit(min, max)’

```
use parser;
fn main() {
```

20.0.2. Parsing a minimal function

The double braces ‘{{‘ and ‘}}’ produce literal ‘{‘ and ‘}’ inside a Loft format string — needed here because the source code itself contains braces.

```
  parser::parse("hello", "fn main() {{ println(\"Hello World\"); }}");
```

20.0.3. Parsing a struct and function together

The parser validates the entire source snippet as one unit. Both the struct definition and the function body are checked.

```
  parser::parse("point", "struct Point {{ x: integer, y: integer }} fn origin() -
> Point {{ Point {{ x: 0, y: 0 }} }}");
```

20.0.4. Parsing an enum

Both a plain variant and a struct variant (with fields) are recognised.

```
  parser::parse("shape", "enum Shape {{ Circle {{ radius: float }}, Rectangle
{{ w: float, h: float }} }}");
```

20.0.5. Parsing a for loop and arithmetic

This exercises the expression parser and range syntax.

```
  parser::parse("loop", "fn sum(n: integer) -> integer {{ t = 0; for i in 1..=n
{{ t += i; }} t }}");
```

20.0.6. Practical use: validating user-supplied code

If your application lets users write Loft snippets (for scripting or configuration), you can parse them before executing:

snippet = read_user_input(); parser::parse(“user_code”, snippet);

Combine this with the lexer (see the Lexer page) when you need to extract individual tokens from the source rather than validate the whole grammar.

```
  println("parser test passed");
}
```

21. Libraries

A library is a `.loft` file you can share across projects. Place it in a `lib/` directory, import it with `use name;` at the top of your file, and then refer to its types and functions using the `name::` prefix. Everything in the library is always accessible — there is no private/public barrier. `use` statements must come before any `fn` or `struct` definition in your file. Putting a `use` after a definition is a syntax error.

```
use testlib;
```

21.0.1. Constants and Enums

Library constants and enum variants are accessed with the `libname::` prefix. You can compare, order, and convert enum values exactly as you would with enums defined in the same file.

21.0.2. Struct Construction and Field Access

Construct a library struct with its full namespace prefix. Omitting the prefix is a parse error. Once you have a value, field access uses the plain dot notation with no prefix needed.

21.0.3. Calling Library Methods

Methods defined in the library are called on the value directly — no prefix on the call site. Free functions (not methods) do need the `libname::` prefix.

21.0.4. Extending a Library Type

You can add new methods to a library type from your own file. Write the `self` parameter type with the `::` separator and the compiler treats the function as a method on that type.

```
fn shifted(self: testlib::Point, dx: float, dy: float) -> testlib::Point {
  testlib::Point {x: self.x + dx, y: self.y + dy }
}
```

```
fn main() {
```

Constants from a library require the library prefix.

```
  assert(testlib::MAX_SIZE == 100, "library constant MAX_SIZE");
  assert(testlib::MIN_SIZE == 1, "library constant MIN_SIZE");
```

Library enum variants: comparison and ordering work as normal.

```
  s = testlib::Ok;
  assert(s == testlib::Ok, "enum equality");
  assert(s < testlib::Error, "enum ordering");
  assert("{s}" == "Ok", "enum to text: {s}");
  assert("Warning" as testlib::Status == testlib::Warning, "text to enum");
```

Library enum values can be passed to library free functions.

```
  assert(testlib::status_text(testlib::Error) == "err", "free fn with enum arg");
```

Structs are constructed with namespace prefix and accessed by field.

```
p = testlib::Point {x: 3.0, y: 4.0 };
assert(p.x == 3.0, "field access: x={p.x}");
assert(p.y == 4.0, "field access: y={p.y}");
```

Methods defined in the library are called without a prefix.

```
dist = p.distance();
assert(round(dist) == 5.0, "method call: distance={dist}");
```

User-defined method on a library type (self: testlib::Point) works.

```
assert(p.shifted(1.0, 2.0).x == 4.0, "shifted x");
assert(p.shifted(0.0, 2.0).y == 6.0, "shifted y");
```

Scalar fields can be mutated directly.

```
b = testlib::Bag {label: "test", count: 0 };
b.label = "updated";
assert(b.label == "updated", "direct scalar field mutation");
```

Methods that mutate scalar fields work.

```
b.bump();
assert(b.count == 1, "method scalar mutation: count={b.count}");
b.bump();
assert(b.count == 2, "method scalar mutation: count={b.count}");
```

Free functions: called with the library prefix.

```
assert(testlib::add(3, 4) == 7, "free function add");
assert(testlib::add(0, -5) == -5, "free function negative");
```

Library types work as function parameter types when written with their full namespace prefix in the function signature. A struct with a vector field can be initialised via a struct literal, including elements that are themselves library structs.

```
bag2 = testlib::Bag {label: "full", count: 3, items: [
testlib::Point {x: 1.0, y: 0.0 },
testlib::Point {x: 0.0, y: 1.0 }
] };
assert(len(bag2.items) == 2, "initialized vector field: {len(bag2.items)}");
assert(bag2.items[0].x == 1.0, "vector field element access");
```

Appending a struct variable with `+= \[var\]` and a whole vector with `+= vec` both work.

```
  extra = testlib::Point {x: 7.0, y: 8.0 };
  bag2.items +=[extra];
  assert(len(bag2.items) == 3, "append via var: len={len(bag2.items)}");
  assert(bag2.items[2].x == 7.0, "append via var: x={bag2.items[2].x}");
  more_pts =[testlib::Point {x: 9.0, y: 10.0 }, testlib::Point {x: 11.0, y:
12.0 }];
  bag2.items += more_pts;
  assert(len(bag2.items) == 5, "append vector: len={len(bag2.items)}");
  assert(bag2.items[3].x == 9.0, "append vector[3].x={bag2.items[3].x}");
```

Importing the same library twice is silently ignored. A library can itself import other libraries using `use`, so dependency chains work.

21.0.5. Package Layout and loft.toml

A library can be distributed as a directory package instead of a single flat file. The packaged directory layout is:

mylib/ loft.toml optional manifest src/ mylib.loft library source (default entry)

When the interpreter searches a lib directory and finds `\<dir\>/mylib/` it looks for `\<dir\>/mylib/src/mylib.loft` automatically.

The optional `loft.toml` manifest supports two settings:

[package] loft = ">=1.0" minimum interpreter version required

[library] entry = "src/mylib.loft" override the default entry path

If the interpreter version is below the stated minimum, loading the library produces a fatal compile error describing the version mismatch. If no manifest is present, the default entry `src/\<name\>.loft` is used.

21.0.6. Limitations

These are the current rough edges to keep in mind. `use` must appear before all definitions. If you write a function first and then a `use`, the compiler reports a syntax error: fn foo() {} use testlib;

```
}
```

22. Store Locks

Loft gives you two separate ways to protect a value from accidental changes:

1. **Compile-time const**: mark a variable or parameter with 'const' and the compiler refuses to compile any code that tries to reassign it. The check happens before the program even runs — zero runtime cost.
2. **Runtime lock**: the '#lock' attribute on a reference lets you lock a store at runtime so that any write attempt panics immediately, even across function boundaries. This is useful for debugging: turn it on when you suspect an unexpected mutation.

```
struct Counter {
  value: integer
}
```

22.0.1. const parameters

'const' on a parameter is a compile-time promise: "this function will not modify this value." The compiler enforces it — any assignment to a const parameter is a compile error, caught before you run anything.

```
fn read_value(self: const Counter) -> integer {
  self.value
}
```

A non-const parameter leaves the store unlocked so the function can write.

```
fn increment(self: Counter) {
  self.value += 1
}
```

```
fn main() {
```

22.0.2. const local variables

Declare a local variable with 'const' to signal that it will not change after its first assignment. The compiler rejects any later assignment to it — reassigning or appending to a const variable is a compile error.

This is handy for configuration values or lookup tables that should never be overwritten by accident deep inside a long function.

```
  const limit = 100;
  assert(limit == 100, "const integer is readable");
```

const also works for struct references.

```
  const cfg = Counter {value: 42 };
  assert(cfg.value == 42, "const reference is readable");
```

Passing a const reference to a const parameter is always allowed.

```
  assert(read_value(cfg) == 42, "const passed to const param");
```

22.0.3. Calling methods on const references

A non-const method can still be called on a non-const variable even after you have manually locked the store; the lock is a runtime check.

```
  c = Counter {value: 10 };
  increment(c);
  assert(c.value == 11, "increment modified c");
```

22.0.4. Runtime store locks with #lock

‘#lock’ is an attribute on any reference variable. Setting it to true turns on a runtime guard: any write to that store will panic immediately, wherever it happens. A freshly created reference starts unlocked.

```
  d = Counter {value: 5 };
  assert(!d#lock, "new store starts unlocked");
  d#lock = true;
  assert(d#lock, "store is locked after assignment");
```

You can still read from a locked store — only writes are blocked.

```
  assert(read_value(d) == 5, "locked store is still readable");
```

22.0.5. When to use each approach

* Use ‘const’ on parameters and locals to express your design intent and get compile-time safety at zero cost. * Use ‘#lock = true’ when you want a runtime tripwire: you suspect some code path is mutating a value it should not touch, and you want the program to panic with a precise location rather than corrupt silently. get_store_lock() is the function form of the #lock attribute. Both return the same boolean.

```
  e = Counter {value: 99 };
  assert(get_store_lock(e) == e#lock, "function form matches attribute before
lock");
  e#lock = true;
  assert(get_store_lock(e) == e#lock, "function form matches attribute after
lock");
}
```

23. Parallel execution

The `par(b=worker_call, threads)` clause on a `for` loop runs a function on every element of a vector in parallel and gives you the results one by one in the loop body. Use it when you have a large collection and a CPU-intensive per-element calculation: the work is spread across the requested number of threads and the results come back in the original order.

The full syntax is: for a in <vector> par(b=<worker_call>, <threads>) { body }

Two worker call forms are supported. Form 1 calls a global or user-defined function with the loop element as its argument: for a in items par(b=my_func(a), 4) { ... }

Form 2 calls a method on the element itself: for a in items par(b=a.my_method(), 4) { ... }

The worker function must take a `const` reference to the element type and return a single primitive value (integer, float, or boolean). ── Shared struct definitions ──────────────────────────────────────

```
struct Score {
  value: integer
}
```

```
struct ScoreList {
  items: vector < Score >
}
```

```
struct Range {
  lo: integer,
  hi: integer
}
```

```
struct RangeList {
  items: vector < Range >
}
```

── Worker functions ──────────────────────────────────────

Global functions (Form 1)

```
fn double_score(r: const Score) -> integer {
  r.value * 2
}
```

```
fn span(r: const Range) -> integer {
  r.hi - r.lo
}
```

```
fn score_as_float(r: const Score) -> float {
  r.value as float
}
```

```
fn score_positive(r: const Score) -> boolean {
  r.value > 0
}
```

Methods on Score (Form 2)

```
fn get_value(self: const Score) -> integer {
  self.value
}
```

```
fn is_positive(self: const Score) -> boolean {
  self.value > 0
}
```

Helpers

```
fn make_scores() -> ScoreList {
  q = ScoreList { };
  q.items +=[Score {value: 10 }, Score {value: 20 }, Score {value: 30 }];
  q
}
```

```
fn make_ranges() -> RangeList {
  q = RangeList { };
  q.items +=[Range {lo: 0, hi: 10 }, Range {lo: 5, hi: 12 }, Range {lo: -3, hi:
7 }];
  q
}
```

```
fn main() {
```

23.0.1. Running a Global Function in Parallel

Each Score's value is doubled by `double_score`. With 1 thread the work is sequential; with 4 threads it runs concurrently. Both must produce the same total because results are delivered in the original order.

```
  q = make_scores();
  sum = 0;
  for a in q.items par(b = double_score(a), 1) {
    sum += b;
  }
  assert(sum == 120, "form-1 integer (1 thread): sum == 120");
```

integer return, 4 threads

```
q2 = make_scores();
sum2 = 0;
for a in q2.items par(b = double_score(a), 4) {
  sum2 += b;
}
assert(sum2 == 120, "form-1 integer (4 threads): sum == 120");
```

`span` works on a two-field struct. The element size is inferred from the struct layout so you do not need to specify it.

```
r = make_ranges();
span_sum = 0;
for a in r.items par(b = span(a), 2) {
  span_sum += b;
}
assert(span_sum == 27, "form-1 Range span: sum == 27");
```

Workers can return float. Here we just count iterations to confirm every element was processed.

```
q3 = make_scores();
fcount = 0;
for a in q3.items par(b = score_as_float(a), 1) {
  fcount += 1;
}
assert(fcount == 3, "form-1 float return: 3 elements processed");
```

Workers can return boolean. Use `if b` in the body to act on the result.

```
q4 = make_scores();
pos = 0;
for a in q4.items par(b = score_positive(a), 1) {
  if b {
    pos += 1;
  }
}
assert(pos == 3, "form-1 boolean: all 3 positive");
```

An empty vector is safe: the loop body simply never executes.

```
empty = ScoreList { };
empty_sum = 0;
for a in empty.items par(b = double_score(a), 1) {
  empty_sum += b;
}
assert(empty_sum == 0, "form-1 empty: body executes 0 times");
```

23.0.2. Running a Method in Parallel

Form 2 calls the worker as a method on each element. The syntax `b=a.get_value()` tells the compiler to dispatch `get_value` on every element in parallel and bind each result to `b` in the loop body.

```
  q5 = make_scores();
  sum3 = 0;
  for a in q5.items par(b = a.get_value(), 1) {
    sum3 += b;
  }
  assert(sum3 == 60, "form-2 integer (1 thread): sum == 60");
```

integer return, 4 threads

```
  q6 = make_scores();
  sum4 = 0;
  for a in q6.items par(b = a.get_value(), 4) {
    sum4 += b;
  }
  assert(sum4 == 60, "form-2 integer (4 threads): sum == 60");
```

boolean return — count positives via method

```
  q7 = make_scores();
  pos2 = 0;
  for a in q7.items par(b = a.is_positive(), 1) {
    if b {
      pos2 += 1;
    }
  }
  assert(pos2 == 3, "form-2 boolean: all 3 positive");
}
```

24. Logging

Printing everything to the console is fine during development, but in a real application you usually want more control: write to a file, filter by severity, and keep the output even after the terminal session ends. Loft's logging functions give you that without changing your source code.

24.0.1. The four severity levels

Choose the level that matches how serious the event is:

* 'log_info' — routine progress; fine to see during development but often silenced in production to keep log files small. Example: "processing file X", "connected to database".

* 'log_warn' — something unexpected happened but the program recovered. Example: "config key missing, using default", "retrying after timeout".

* 'log_error' — something went wrong and you need to investigate, but the program can continue (perhaps degraded). Example: "failed to save record", "unexpected null value".

* 'log_fatal' — a condition so serious that normal operation is impossible. Example: "cannot open database", "required config file not found".

24.0.2. Configuring the log destination

By default, log calls are silent no-ops — no file is written, no console output is produced. To switch logging on, place a 'log.conf' file in the same directory as your '.loft' file, or pass '–log-conf path/to/log.conf' on the command line.

Generate a documented template with all defaults by running: loft –generate-log-config

A minimal 'log.conf' looks like this:

[log] file = log.txt # write messages here (relative to the .loft file) level = info # minimum level to record; choices: info warn error fatal

[rotation] max_size_mb = 500 # rotate after this many megabytes daily = true # also rotate at midnight UTC max_files = 10 # keep at most this many log files

[rate_limit] per_site = 5 # suppress messages from the same source line after 5/minute

[levels] # Override the global level for a specific file: # "debug_tool.loft" = info # "src/" = error

24.0.3. Production mode

When 'production = true' is set in log.conf: * 'panic()' becomes a fatal log entry instead of aborting the process. * A failing 'assert()' becomes an error log entry instead of aborting. The program keeps running and the problem is captured in the log — useful for long-running services where a single error should not bring everything down.

24.0.4. Using format strings in log messages

Log messages are plain text, but you can embed any expression using the same '{...}' format syntax as everywhere else in Loft. The interpolation happens only when the message is actually going to be written; if the configured level is higher than the call, the string is never evaluated (no performance cost for suppressed messages).

```
fn main() {
```

Without a log.conf these are all silent no-ops — the tests below pass even though no output is produced.

```
  log_info("starting up");
  log_warn("this is a warning");
  log_error("something went wrong");
  log_fatal("critical failure");
```

A true assert never logs anything — it is only the false case that logs.

```
  assert(true, "this should never fail");
```

24.0.5. Typical usage pattern

In a real program you would write something like:

fn process(item: Item) { log_info(“processing item {item.id}”); result = do_work(item); if !result { log_error(“work failed for item {item.id}”); return; } log_info(“finished item {item.id} successfully”); }

The log messages give you a record of exactly what the program did and where it went wrong, without cluttering the terminal during normal runs.

```
  println("logging test passed");
}
```

25. Random

Loft provides three functions for randomness:

rand(lo, hi) — a random integer in [lo, hi] inclusive. rand_seed(seed) — seed the random number generator for reproducible results. rand_indices(n) — a vector of n integers [0..n-1] in a random order.

The generator is a fast PCG64 algorithm. Without an explicit seed the generator starts from a fixed default seed, so results are reproducible across runs unless you seed with a time-based value.

```
fn main() {
```

25.0.1. Basic random integers

‘rand(lo, hi)’ returns a uniformly distributed integer in [lo, hi]. Call ‘rand_seed’ first to choose the sequence.

```
  rand_seed(42);
  r = rand(1, 6);
```

simulated die roll: 1..6

```
  assert(r >= 1 && r <= 6, "die roll out of range: {r}");
```

The same seed always produces the same sequence.

```
  rand_seed(0);
  a = rand(0, 999);
  rand_seed(0);
  assert(rand(0, 999) == a, "seeded rand must be reproducible");
```

‘rand’ returns null when lo > hi.

```
  assert(!rand(10, 5), "invalid range returns null");
```

25.0.2. Random ordering: rand_indices

‘rand_indices(n)’ returns a vector containing 0, 1, …, n-1 in a random order. Use the indices to visit another collection in random order without copying it.

```
  rand_seed(7);
  order = rand_indices(5);
  assert(len(order) == 5, "size must be 5");
```

Every value 0..4 appears exactly once.

```
  items =["apple", "banana", "cherry", "date", "elderberry"];
  seen =[for i in 0..5 {
    false
  }];
```

```
  for idx in order {
    seen[idx] = true;
  }
  all_seen = true;
  for s in seen {
    if !s {
      all_seen = false
    }
  }
  assert(all_seen, "rand_indices must cover all positions");
```

25.0.3. Sampling without replacement

Pick k distinct items from a list by taking the first k indices.

```
  rand_seed(1);
  indices = rand_indices(len(items));
```

Take the first 3 items in random order.

```
  picked = "";
  for i in 0..3 {
    if i > 0 {
      picked += ", "
    }
    picked += items[indices[i]]
  }
```

‘picked’ now contains 3 distinct fruit names in a random order.

```
  assert(len(picked) > 0, "should have picked some items: {picked}");
}
```

26. Time

Loft provides two time functions:

now() — milliseconds since the Unix epoch (wall-clock time). ticks() — microseconds elapsed since program start (monotonic clock).

Both return a 'long'.

Use 'now()' for timestamps, log entries, and date calculations. Use 'ticks()' for benchmarks and frame timing — it is unaffected by system clock changes or NTP adjustments.

```
fn main() {
```

26.0.1. Wall-clock time: now()

'now()' returns the current time as milliseconds since 1970-01-01T00:00:00 UTC. The value is always positive and grows over time.

```
  t = now();
  assert(t > 0l, "now() must be positive");
```

Two successive calls return non-decreasing values.

```
  t2 = now();
  assert(t2 >= t, "now() must be non-decreasing");
```

26.0.2. Elapsed time: ticks()

'ticks()' measures microseconds since the program started. It uses a monotonic clock so it never jumps backward.

```
  start = ticks();
  assert(start >= 0l, "ticks() must be non-negative");
  end = ticks();
  assert(end >= start, "ticks() must be monotonically non-decreasing");
```

26.0.3. Measuring elapsed time

Subtract two 'ticks()' values to get the duration in microseconds. Divide by 1000 to convert to milliseconds.

elapsed_us = end_ticks - start_ticks elapsed_ms = elapsed_us / 1000l

Example: measure how long a loop takes.

```
  t_before = ticks();
  sum = 0;
  for i in 0..1000 {
    sum += i
  }
  t_after = ticks();
```

```
    elapsed = t_after - t_before;
    assert(elapsed >= 0l, "elapsed time must be non-negative: {elapsed}");
    assert(sum == 499500, "loop produced wrong sum: {sum}");
```

26.0.4. Common patterns

Timestamp a log entry (seconds since epoch): seconds = now() / 1000l

Seed the random number generator with the current time: rand_seed(now() as integer)

Simple stopwatch: start = ticks() … do work … log_info("Done in {(ticks() - start) / 1000l} ms")

```
}
```

27. Safety

Loft catches many errors at compile time, but a few surprises remain at runtime. This page catalogues every known trap so you can write confident code from day one. Each section includes a live example that proves the described behavior.

```
fn main() {
```

27.0.1. Null sentinels

Every type uses a special in-band value to represent null. The value depends on the type:

- `boolean` uses `false`
- `integer` uses `i32::MIN` (-2 147 483 648)
- `long` uses `i64::MIN`
- `float` and `single` use `NaN`
- `character` uses the NUL character
- `text` uses an internal null pointer
- `reference` uses record 0
- plain `enum` uses byte 255 (limits enums to 255 variants)

This means there is one value per type that you cannot distinguish from null. For integers, that value is `i32::MIN`. Division by zero also produces `i32::MIN`, so both paths look the same to your code:

```
  zero = 0;
  n = 1 / zero;
  assert(!n, "div-by-zero is null");
  assert(n != 0, "null sentinel is not zero; it is i32::MIN");
  assert(n < 0, "i32::MIN is the most negative 32-bit integer");
```

Arithmetic on null propagates: null plus anything is null.

```
  assert(!(n + 1), "null + 1 is still null");
```

Mitigation: Use `long` when you need the full 32-bit range, or declare struct fields as `not null` to reclaim the sentinel value.

27.0.2. Integer overflow wraps silently

32-bit integers wrap when they exceed roughly 2 billion. There is no runtime overflow check and no exception. The result is a small or negative number and the program continues as if nothing happened. The following would wrap silently in a release build: big = 2000000000; big + big ⟶ negative number Mitigation: Use `long` (64-bit) when multiplying or summing large values: `big as long + big as long` avoids the wrap.

```
  big = 2000000000;
  assert(big as long + big as long == 4000000000l, "long avoids wrap");
```

27.0.3. Bitwise operators with zero

All bitwise operators (AND, OR, XOR, shift) work correctly with zero. Zero is the identity element for OR, XOR, and shift; zero for AND.

```
assert(0b1010 & 0 == 0, "AND with 0: zero");
assert(0b1010 | 0 == 0b1010, "OR with 0: identity");
assert(0b1010 ^ 0 == 0b1010, "XOR with 0: identity");
assert(5 << 0 == 5, "shift left by 0: identity");
assert(5 >> 0 == 5, "shift right by 0: identity");
```

27.0.4. Float null is NaN

Floats represent null as `NaN` (Not a Number). Null floats behave consistently with other null types: null is not equal to anything (including itself), and null is not-equal to everything.

```
bad = 0.0 / 0.0;
assert(!bad, "NaN is falsy — this is how you detect null floats");
assert(!(bad == bad), "null == null is false");
assert(bad != bad, "null != null is true");
assert(bad != 0.0, "null != 0.0 is true");
```

Use `!f` or `f ?? default` to check for null floats.

27.0.5. Text length counts bytes, not characters

`len()` on text returns the number of UTF-8 bytes, not the number of visible characters. Multi-byte characters (accented letters, emoji, CJK) each occupy 2-4 bytes.

```
emoji = "Hi 😊!";
assert(len(emoji) == 8, "5 visible chars but 8 bytes (emoji is 4 bytes)");
```

Slicing and indexing also use byte offsets. Slicing in the middle of a multi-byte character is an error. Mitigation: Use `for c in text` to iterate by character. Use `c\#index` and `c\#next` to get the byte boundaries of each character.

```
count = 0;
for c in emoji { count += 1; }
assert(count == 5, "for-loop iterates by character, not byte");
```

27.0.6. `\#index` means different things on text and vectors

On a text loop, `c\#index` is the byte offset of the current character. On a vector loop, `v\#index` is the 0-based element position. Both are called `\#index` but represent different units.

```
v = [10, 20, 30];
idx = 0;
for x in v { idx = x#index; }
assert(idx == 2, "vector #index: element position (0-based)");
```

Text #index is a byte offset, not a character count:

```
t = "aé";
byte_pos = 0;
for c in t { byte_pos = c#index; }
assert(byte_pos == 1, "text #index: byte offset of 'é' (byte 1, not char 1)");
```

27.0.7. ?? evaluates the left side twice for complex expressions

The null-coalescing operator `??` checks if the left side is null and returns the right side if so. For a simple variable this is fine, but for a function call or complex expression the left side is evaluated once for the null check and once for the result. Mitigation: Assign complex expressions to a temporary variable first. For example, instead of `result = expensive_call() ?? default` (which calls the function twice), write `temp = expensive_call()` on one line and then `result = temp ?? default` on the next.

27.0.8. Text indexing and slicing return different types

`txt\[i\]` returns a `character` (a single Unicode scalar value). `txt\[i..j\]` returns `text` (a UTF-8 string). These are different types.

```
txt = "hello";
ch = txt[0];
slice = txt[0..1];
assert(ch == 'h', "indexing returns a character");
assert(slice == "h", "slicing returns text");
```

Building text from characters requires format interpolation:

```
result = "";
for c in "abc" { result += "{c}"; }
assert(result == "abc", "characters must be formatted into text");
```

27.0.9. Format strings: braces are always interpreted

Every string literal in loft is a format string. Literal braces must be escaped as `{{` and `}}`:

```
n = 42;
assert("{n}" == "42", "single braces: format expression");
assert(len("{{}}") == 2, "double braces produce literal brace chars");
```

Forgetting to escape braces in expected output is a common mistake in assertions and comparisons.

27.0.10. Hash collections cannot be iterated

Hashes are lookup structures, not ordered collections. You cannot write `for item in my_hash { }`. If you need both fast lookup and ordered iteration, keep a vector and a hash pointing at the same record type. See the Hash documentation page for the recommended pattern.

27.0.11. Mutation guard blocks appending during iteration

The compiler prevents `v += \[x\]` inside `for e in v`. This protects against infinite loops. The guard also catches field access: `for e in db.items { db.items += \[x\]; }` is blocked too. The only allowed mutation is `e\#remove` inside a filtered loop.

27.0.12. If-expression requires else when used as a value

Using `if` as a value expression without an `else` clause is a compile error. This prevents accidental null values from missing branches. If-statements (where the body has no value) do not need else. For example, `x = if cond { 1 }` is an error; write `x = if cond { 1 } else { 0 }`.

27.0.13. Match guards do not prove a variant is handled

A guarded arm like `Red if cond =\> ...` does not count as handling the `Red` variant because the guard can fail at runtime. Even if every variant has a guarded arm, you still need a wildcard _ or an unguarded arm so the compiler knows every case is covered.

27.0.14. Ref-parameter semantics

Without &, appending to a vector parameter is local — the caller's vector does not grow. With &, the caller sees the new elements. Field-level mutations (e.g. `v\[i\].field = x`) are always visible to the caller because both sides share the same underlying database reference. Rule of thumb: Use `&vector\<T\>` when the function needs to grow the vector. Use plain `vector\<T\>` when the function only reads or modifies existing elements.

27.0.15. File I/O assumes UTF-8

All file reading in loft assumes the file content is valid UTF-8. Reading a binary file or a file in a different encoding (e.g. Latin-1) will crash the program at runtime. There is currently no way to read raw bytes. Mitigation: Only read files you know to be UTF-8. If you need to process binary data, convert it to UTF-8 externally before passing it to loft.

27.0.16. XOR is ^, not exponentiation

Unlike some languages where ^ means "power", in loft ^ is bitwise XOR. Use the `pow()` function for exponentiation.

```
  assert((0b1010 ^ 0b1100) == 0b0110, "^ is XOR");
  assert(pow(2.0, 3.0) == 8.0, "use pow() for power");
}
```

28. Standard Library

28.1. Types

```
pub type boolean size(1)
```

Primitive types built into lav. True or false value.

```
pub type integer size(4)
```

32-bit signed integer.

```
pub type long size(8)
```

64-bit signed integer. Use when values exceed 2 billion.

```
pub type single size(4)
```

32-bit floating-point. Good for graphics and performance-sensitive math.

```
pub type float size(8)
```

64-bit floating-point. Use when precision matters.

```
pub type text size(4)
```

UTF-8 string.

```
pub type character size(4)
```

A single Unicode code point.

```
pub type u8 = integer limit(0, 255) size(1)
```

Integer subtypes for compact storage in struct fields. Behave as integer in expressions. 0 – 255, 1 byte.

```
pub type i8 = integer limit(-128, 127) size(1)
```

–128 – 127, 1 byte.

```
pub type u16 = integer limit(0, 65535) size(2)
```

0 – 65535, 2 bytes.

```
pub type i16 = integer limit(-32768, 32767) size(2)
```

–32768 – 32767, 2 bytes.

```
pub type i32 = integer size(4)
```

Full 32-bit integer range, 4 bytes.

28.2. Math

Functions for numeric computation. All trigonometric functions work in radians. Both single and float variants exist for every function — choose single for speed, float for precision. Integer operations

```
pub fn abs(both: integer) -> integer
```

Absolute value. Removes the sign from a negative integer.

```
pub fn abs(both: long) -> long
```

Absolute value for long integers.

```
pub fn abs(both: single) -> single
```

Absolute value for single-precision floats.

```
pub fn cos(both: single) -> single
```

Cosine. Use for circular motion: x = r * cos(angle).

```
pub fn sin(both: single) -> single
```

```
pub fn tan(both: single) -> single
```

```
pub fn acos(both: single) -> single
```

```
pub fn asin(both: single) -> single
```

```
pub fn atan(both: single) -> single
```

```
pub fn ceil(both: single) -> single
```

```
pub fn floor(both: single) -> single
```

```
pub fn round(both: single) -> single
```

```
pub fn sqrt(both: single) -> single
```

```
pub fn atan2(both: single, v2: single) -> single
```

Arc tangent of y/x, preserving the correct quadrant. Use instead of atan when you have separate x/y components.

```
pub fn log(both: single, v2: single) -> single
```

```
pub fn pow(both: single, v2: single) -> single
```

Raises base to the power exp. Use for exponential growth curves and scaling.

```
pub fn abs(both: float) -> float
```

Absolute value for double-precision floats.

```
pub PI = OpMathPiFloat()
```

The ratio of a circle's circumference to its diameter (3.14159...).

```
pub E = OpMathEFloat()
```

Euler's number, the base of natural logarithms (2.71828...).

```
pub fn cos(both: float) -> float
```

Cosine. Use for circular motion: x = r * cos(angle).

```
pub fn sin(both: float) -> float
```

```
pub fn tan(both: float) -> float
```

```
pub fn acos(both: float) -> float
```

```
pub fn asin(both: float) -> float
```

```
pub fn atan(both: float) -> float
```

```
pub fn ceil(both: float) -> float
```

```
pub fn floor(both: float) -> float
```

```
pub fn round(both: float) -> float
```

```
pub fn sqrt(both: float) -> float
```

```
pub fn atan2(both: float, v2: float) -> float
```

Arc tangent of y/x, preserving the correct quadrant. Use instead of atan when you have separate x/y components.

```
pub fn log(both: float, v2: float) -> float
```

```
pub fn pow(both: float, v2: float) -> float
```

Raises base to the power exp. Use for exponential growth curves and scaling.

28.3. exp / ln / log2 / log10

Raises E (2.71828…) to the power v. Use for exponential growth models.

```
pub fn exp(both: single) -> single
```

```
pub fn exp(both: float) -> float
```

```
pub fn ln(both: single) -> single
```

```
pub fn ln(both: float) -> float
```

```
pub fn log2(both: single) -> single
```

```
pub fn log2(both: float) -> float
```

```
pub fn log10(both: single) -> single
```

```
pub fn log10(both: float) -> float
```

28.4. min / max / clamp

```
pub fn min(both: integer, b: integer) -> integer
```

Returns null if either argument is null (consistent with all binary arithmetic operations). Smallest of two integer values.

```
pub fn max(both: integer, b: integer) -> integer
```

```
pub fn clamp(both: integer, lo: integer, hi: integer) -> integer
```

```
pub fn min(both: long, b: long) -> long
```

```
pub fn max(both: long, b: long) -> long
```

```
pub fn clamp(both: long, lo: long, hi: long) -> long
```

```
pub fn min(both: single, b: single) -> single
```

```
pub fn max(both: single, b: single) -> single
```

```
pub fn clamp(both: single, lo: single, hi: single) -> single
```

```
pub fn min(both: float, b: float) -> float
```

```
pub fn max(both: float, b: float) -> float
```

```
pub fn clamp(both: float, lo: float, hi: float) -> float
```

28.5. Text

Functions for working with text (UTF-8 strings) and character values. Read the value of a variable and put a reference to it on the stack

```
pub fn len(both: text) -> integer
```

Number of bytes in the text. Use for bounds checks and iteration limits.

```
pub fn len(both: character) -> integer
```

Byte length of the character's UTF-8 encoding (1–4).

Splits self on every occurrence of separator and returns the parts as a vector. Use to parse CSV lines or space-separated tokens.

```
pub fn split(self: text, separator: character) -> vector < text >
```

```
pub fn starts_with(self: text, value: text) -> boolean
```

Functions for searching, transforming, and classifying text and character values. Character classification functions return true only if every character in the text satisfies the condition. The single-character variants test one code point. Returns true if self begins with value. Use for prefix matching (e.g., protocol detection).

```
pub fn ends_with(self: text, value: text) -> boolean
```

Returns true if self ends with value. Use for suffix matching (e.g., file extension checks).

```
pub fn trim(both: text) -> text[both]
```

Removes leading and trailing whitespace. Use when processing user input or file content.

```
pub fn trim_start(self: text) -> text[self]
```

Removes leading whitespace only.

```
pub fn trim_end(self: text) -> text[self]
```

Removes trailing whitespace only.

```
pub fn find(self: text, value: text) -> integer
```

Returns the byte index of the first occurrence of value, or null if not found. Use to locate substrings before slicing.

```
pub fn rfind(self: text, value: text) -> integer
```

Returns the byte index of the last occurrence of value, or null if not found. Use to find file extensions or the last path separator.

```
pub fn contains(self: text, value: text) -> boolean
```

Returns true if value appears anywhere in self. Simpler than find when you only need a yes/no answer.

```
pub fn replace(self: text, value: text, with: text) -> text
```

Returns a copy of self with every occurrence of value replaced by with.

```
pub fn to_lowercase(self: text) -> text
```

Returns a lowercase copy. Use for case-insensitive comparisons.

```
pub fn to_uppercase(self: text) -> text
```

Returns an uppercase copy.

```
pub fn is_lowercase(self: text) -> boolean
```

True if all characters are lowercase letters.

```
pub fn is_lowercase(self: character) -> boolean
```

True if the character is a lowercase letter.

```
pub fn is_uppercase(self: text) -> boolean
```

True if all characters are uppercase letters.

```
pub fn is_uppercase(self: character) -> boolean
```

True if the character is an uppercase letter.

```
pub fn is_numeric(self: text) -> boolean
```

True if all characters are numeric digits (Unicode numeric, not just ASCII 0–9).

```
pub fn is_numeric(self: character) -> boolean
```

True if the character is a numeric digit.

```
pub fn is_alphanumeric(self: text) -> boolean
```

True if all characters are letters or digits. Use to validate identifiers or tokens.

```
pub fn is_alphanumeric(self: character) -> boolean
```

True if the character is a letter or digit.

```
pub fn is_alphabetic(self: text) -> boolean
```

True if all characters are alphabetic.

```
pub fn is_alphabetic(self: character) -> boolean
```

True if the character is alphabetic.

```
pub fn is_whitespace(self: text) -> boolean
```

True if all characters are whitespace. Use to detect blank lines.

```
pub fn is_control(self: text) -> boolean
```

True if all characters are control characters.

```
pub fn join(parts: vector < text >, sep: text) -> text
```

Joins parts with sep between each consecutive pair. Returns “” for an empty vector. Use to build comma-separated lists, path segments, or any delimited output.

28.6. Collections

Operations on vector<T> — the primary ordered collection type. Vectors are grown by appending with += and elements are accessed by index. All structures are passed by reference instead of by value

```
pub fn len(both: vector) -> integer
```

Number of elements in the vector. Use in loop bounds: for i in 0..v.len().

```
pub fn clear(both: vector)
```

Remove all elements from the vector, setting its length to 0.

28.7. Output and Diagnostics

```
pub fn assert(test: boolean, message: text, file: text, line: integer)
```

Panics with message if test is false. Use to verify invariants during development. In production mode (–production), logs an error instead of aborting. The file and line are injected by the compiler; do not pass them manually.

```
pub fn panic(message: text, file: text, line: integer)
```

Immediately terminates execution with message. Use for unrecoverable error states. In production mode (–production), logs a fatal entry instead of aborting. The file and line are injected by the compiler; do not pass them manually.

```
pub fn log_info(message: text, file: text, line: integer)
```

Write a structured log record at the chosen severity. The file and line are injected by the compiler; do not pass them manually.

```
pub fn log_warn(message: text, file: text, line: integer)
```

```
pub fn log_error(message: text, file: text, line: integer)
```

```
pub fn log_fatal(message: text, file: text, line: integer)
```

```
pub fn print(v1: text)
```

Writes v to standard output without a newline. Use for progress output and building up a line incrementally.

```
pub fn println(v1: text)
```

28.8. Parallel

Internal: run a loft function over every element of a vector in parallel. Not part of the public API — use the `par(b=worker(a), N)` for-loop clause instead.

28.9. File System

```
pub struct EnvVariable {
  name: text,
  value: text
}
```

Types and functions for reading and writing files. A File value is obtained via file() and carries the path, format, and an internal reference. An environment variable as a name/value pair.

```
pub struct Pixel {
  r: integer limit(0, 255) not null,
  g: integer limit(0, 255) not null,
  b: integer limit(0, 255) not null
}
```

```
pub struct Image {
  name: text,
  width: integer,
  height: integer,
  data: vector < Pixel >
}
```

```
pub enum Format {
  TextFile,
  LittleEndian,
  BigEndian,
  Directory,
  NotExists
}
```

```
pub struct File {
  path: text,

  size: long,

  format: Format,

  ref: i32,

  current: long,

  next: long
}
```

```
pub fn value(self: Pixel) -> integer
```

Returns the pixel colour as a packed 24-bit integer (0xRRGGBB). Use for fast colour comparison or storage.

```
pub fn content(self: File) -> text
```

```
pub fn lines(self: File) -> vector < text >
```

Returns the platform path separator character: '\' on Windows, '/' elsewhere. Detected once at startup from the runtime filesystem.

```
pub fn path_sep() -> character
```

```
pub fn file(path: text) -> File
```

```
pub fn exists(path: text) -> boolean
```

```
pub fn delete(path: text) -> boolean
```

```
pub fn move(from: text, to: text) -> boolean
```

```
pub fn mkdir(path: text) -> boolean
```

```
pub fn mkdir_all(path: text) -> boolean
```

```
pub fn set_file_size(self: File, size: long) -> boolean
```

```
pub fn files(self: File) -> vector < File >
```

```
pub fn write(self: File, v: text)
```

```
pub fn png(self: File) -> Image
```

Decodes a PNG file and returns an Image. Returns null if the file is not in text format. Use to load sprite sheets or textures.

28.10. Environment

Returns all environment variables as a vector of EnvVariable records (fields: name, value). Use to inspect or forward the full environment.

```
pub fn env_variables() -> vector < EnvVariable >
```

```
pub fn env_variable(name: text) -> text
```

Returns the value of the environment variable name, or null if it is not set. Use to read configuration from the shell environment.

Functions for interacting with the host operating system. Returns the command-line arguments passed to the program. The first element is typically the program name.

```
pub fn arguments() -> vector < text >
```

```
pub fn directory(v:  & text = "") -> text
```

Returns the current working directory, optionally with v appended as a subpath. Use to construct absolute paths relative to where the program was launched.

```
pub fn user_directory(v:  & text = "") -> text
```

Returns the current user's home directory, optionally with v appended. Use for storing user-specific data or configuration.

```
pub fn program_directory(v:  & text = "") -> text
```

Returns the directory containing the running executable, optionally with v appended. Use to locate assets bundled alongside the program.

28.11. Random

```
pub fn rand(lo: integer, hi: integer) -> integer
```

Returns a random integer in [lo, hi] (inclusive). Returns null when lo > hi or when either argument is null.

```
pub fn rand_seed(seed: long)
```

Seeds the thread-local random number generator with the given value. Call this before rand() or rand_indices() to get reproducible sequences.

```
pub fn rand_indices(n: integer) -> vector < integer >
```

Returns a vector of n integers [0, 1, ..., n-1] in a random order. Useful for random iteration, shuffling, or sampling without replacement. Returns an empty vector when n is 0 or null.

28.12. Time

```
pub fn now() -> long
```

Returns the current wall-clock time as milliseconds since the Unix epoch (1970-01-01T00:00:00 UTC). Use for timestamps and logging.

```
pub fn ticks() -> long
```

Returns microseconds elapsed since program start (monotonic clock). Unaffected by system clock adjustments. Use for benchmarks and frame timing.

29. Roadmap

This page describes the planned development path for Loft and loft. Features marked **planned** are not yet available — the code examples show the intended syntax once the feature ships.

29.0.1. Current status — version 0.x

The language is under active development. All features documented on the language pages work correctly and are stable enough for real use, but the stability guarantee (no breaking changes) does not apply until version 1.0.

What works today:

- All primitive types: `integer`, `long`, `float`, `boolean`, `text`
- Structs and struct-enums with methods
- Collections: `vector`, `sorted`, `index`, `hash`
- Functions, default parameters, reference parameters (`&`), `const` parameters
- Function references (`fn name`) and higher-order functions (`map`, `filter`, `reduce`)
- File I/O: text, binary, directory listing
- Parallel execution: `par(…)` for-loop clause and `parallel_for()`
- Logging framework
- Library system (`use mylib;`)
- Null safety: every type is nullable; `not null` annotation on struct fields
- Format strings with expression interpolation: `"{x * 2 + 1}"`

29.0.2. Version 1.0 — Language stability

Version 1.0 is a stability contract: any program that works on 1.0 will compile and run identically on all future 1.x releases. The language surface — syntax, type system, documented standard library, command-line flags — is frozen at 1.0.

The primary gate items are correctness fixes (no panics on valid programs) and infrastructure (correct project identity, CHANGELOG, release binaries). See the internal planning documents for the full gate list.

The following features are target items for 1.0 — they are planned for the 1.0 release but will ship in 1.1 if they are not ready in time:

29.0.2.1. Match expressions planned

A `match` expression dispatches on an enum value with compiler-checked exhaustiveness. If every variant is not covered and no wildcard arm is present, the compiler reports an error.

```
// Plain enum dispatch
direction = North;
label = match direction {
    North => "north"
    East  => "east"
    South => "south"
    West  => "west"
};
assert(label == "north", "match direction");
```

```
// Struct-enum: each arm binds the variant's fields
area = match shape {
    Circle { radius }       => PI * radius * radius
    Rect   { width, height } => width * height
};
```

```
// Wildcard arm catches anything not listed above
description = match code {
    200 => "ok"
    404 => "not found"
    _   => "unknown"
};
```

Today the same logic requires an if/else chain or a dispatched method. Match makes it shorter and adds exhaustiveness checking.

29.0.2.2. Wildcard and selective imports planned

The current `use mylib;` import requires prefixing every name with `mylib::`. Two shorter forms are planned:

```
use mylib::*;              // bring all public names into scope
use mylib::Point, add;    // bring specific names into scope
```

The compiler will report a collision error if a wildcard-imported name shadows a local definition.

29.0.2.3. Code formatter planned

A canonical formatter enforces one consistent Loft style — 2-space indentation, opening braces on the same line, 80-column line wrapping. No configuration.

```
loft --format my_program.loft       # format in-place
loft --format-check my_program.loft  # exit 1 if formatting differs (CI)
loft --format src/                   # format every .loft file in a directory
```

The formatter is a token-stream pass that preserves all comments and produces a deterministic output. It does not require the program to type-check successfully.

29.0.3. Version 1.1 — Ergonomics

29.0.3.1. Lambda expressions planned

Today, passing a function to `map` or `filter` requires a named top-level function. Lambda expressions let you write the function body inline without giving it a name:

```
nums = [1, 2, 3, 4, 5];

// Today: requires a named function
fn double(x: integer) -> integer { x * 2 }
doubled = map(nums, fn double);
```

```
// With lambdas: inline, no top-level declaration needed
doubled = map(nums, fn(x: integer) -> integer { x * 2 });
evens   = filter(nums, fn(x: integer) -> boolean { x % 2 == 0 });
total   = reduce(nums, 0, fn(acc: integer, x: integer) -> integer { acc + x });
```

Lambdas do not capture surrounding variables in the first version — context must be passed explicitly. Closure capture is a longer-term item.

29.0.3.2. Interactive mode (REPL) planned

Running `loft` with no arguments will enter an interactive session where each expression is evaluated immediately and its result printed:

```
$ loft
> x = 42
> "{x * 2}"
84
> struct Point { x: float, y: float }
> p = Point { x: 1.0, y: 2.0 }
> p.x + p.y
3.0
```

Variable and type definitions persist across lines for the duration of the session. A parse error discards the failed line and continues the session.

29.0.4. Web IDE — independent track

A fully serverless browser-based IDE for Loft is being developed in parallel with the interpreter. It will run the full loft interpreter as a WebAssembly module — no installation, no account, no server.

Planned features:

- Syntax highlighting with the CodeMirror 6 editor
- Run button ⟶ console output and problems panel
- Go-to-definition and find-usages (Ctrl+click)
- Multiple projects stored locally in the browser (IndexedDB)
- Documentation and example browser without leaving the IDE
- One-click ZIP export with a structure ready for local `loft` development
- Works offline after first load (PWA service worker)

The web IDE does not depend on interpreter version 1.0 and will be released on its own timeline.

29.0.5. Following progress

Development is tracked in the GitHub repository. The internal planning documents describe each item in detail, including implementation notes and effort estimates.