
# The IRIS Programming Language

A Complete Guide

---

IRIS Language Project

Version 0.2.0 — 2025

*A practical guide to writing fast, expressive programs in IRIS —*
*from your first hello world to neural networks and native binaries.*

# 1 The IRIS Programming Language

## 1.1 A Complete Guide

---

*A practical guide to writing fast, expressive programs in IRIS — from your first hello world to neural networks and native binaries.*

---

## 1.2 Table of Contents



---

## 1.3 Foreword

IRIS is a systems and machine-learning DSL built with performance, expressiveness, and clarity in mind. It compiles through LLVM to native machine code, meaning the programs you write are fast — not scripted, not interpreted at runtime. At the same time, IRIS gives you high-level conveniences that feel at home in a modern programming language: type inference, closures, algebraic data types, pattern matching, channels for concurrency, and built-in automatic differentiation.

**Who is this book for?**

This book is for programmers who have some experience in at least one other language (Python, Rust, C, Go, or similar). You do not need to have written a compiler or worked in

ML before. The early chapters cover fundamentals; the later chapters tackle advanced topics like tensors, gradient descent, and native binary compilation.

### 1.3.1 How to read this book

Read chapters 1 through 5 in order — they build on each other. After that, the chapters are largely independent. If you are specifically interested in ML, jump to chapters 10 and 11 after chapter 5. If you want to compile native binaries, chapter 12 stands alone.

Every chapter has working code examples you can run immediately, a "Try It Yourself" section with exercises, and a "Common Mistakes" sidebar covering the pitfalls that trip up new IRIS programmers.

Let's get started.

---

## 1.4 Chapter 1: Getting Started

### 1.4.1 1.1 What Is IRIS?

IRIS is a statically-typed compiled language. Its pipeline looks like this:

```
.iris source file
       |
    Lexer  (text → tokens)
       |
    Parser (tokens → AST)
       |
    Lowerer (AST → IR)
       |
  Optimization passes
       |
  LLVM IR backend
       |
  clang/gcc linker
       |
  Native binary (.exe on Windows)
```

The same source can also be run directly by the built-in tree-walking interpreter (for quick development), or compiled all the way to a native binary for production.

### 1.4.2 1.2 Installation (Windows)

IRIS ships as a single executable. After downloading the installer or the standalone `iris.exe`:

1. Copy `iris.exe` to a folder on your `PATH` (for example, `C:\tools\iris\`).
2. Open a new terminal and verify the installation:

```
iris --version
```

You should see output like:

```
```

```
iris 0.2.0
```

For native binary compilation, IRIS requires LLVM/clang 17+ and MSYS2 MinGW GCC (ucrt64). Install LLVM from https://releases.llvm.org/ (to `C:\Program Files\LLVM`) and MSYS2 from https://www.msys2.org/ (to `C:\msys64`), then run `pacman -S mingw-w64-ucrt-x86_64-gcc` in the MSYS2 terminal.

### 1.4.3 1.3 Hello, World

Create a file named `hello.iris`:

```
def main() -> i64 {
    print("Hello, World!");
    0
}
```

Run it:

```
iris run hello.iris
```

Output:

```
Hello, World!
```

Let's look at what each line does:

- `def main() -> i64` — defines a function named `main` with no parameters that returns an `i64` (64-bit integer).
- `print("Hello, World!");` — calls the built-in `print` function. The semicolon is required because this is a non-tail statement.
- `0` — the last expression in a function is its return value. `main` returns `0`, meaning success.

> **Note:** IRIS functions return the value of their last expression. You rarely need an explicit `return` statement. The last line `0` is the return value — it has no semicolon because it is the tail expression.

### 1.4.4 1.4 The REPL

IRIS comes with an interactive REPL (Read-Eval-Print Loop) that is great for exploring the language:

```
iris repl
```

You will see:

```
```

```
IRIS 0.2.0 REPL
  :help for commands · :quit to exit · Ctrl+D to exit

>>
```

Try typing expressions:

```
>> 1 + 2
3
>> "hello"
hello
>> 42 * 10
420
```

You can define functions and use them immediately:

```
>> def square(x: i64) -> i64 { x * x }
>> square(7)
49
```

**REPL commands:**

| Command | Description |
|---|---|
| `:help` | Show available commands |
| `:env` | List all active bindings and definitions |
| `:type <expr>` | Show the inferred type of an expression |
| `:bring std.math` | Load a standard library module |
| `:reset` | Clear all session state |
| `:quit` | Exit the REPL |

### 1.4.5 1.5 IDE Setup

IRIS includes a Language Server Protocol (LSP) server. For Visual Studio Code:

1. Install the `vscode-iris` extension from the `vscode-iris/` folder in the IRIS distribution.
2. Open a `.iris` file — the extension automatically starts the LSP server.

Features provided:

- Syntax highlighting
- Hover documentation
- Go to definition
- Error diagnostics (underlines)
- Auto-completion
- Outline view
- Signature help
- Document formatting

To start the LSP server manually (for other editors):

```
iris lsp
```

### 1.4.6 Try It Yourself

1. Write a program that prints your name.
2. Write a program that prints the result of `100 * 100`.
3. Open the REPL and use `:type` to check the type of `3.14`.

---

## 1.5 Chapter 2: Values and Types

### 1.5.1 2.1 Primitive Types

IRIS has five primitive scalar types:

| Type | Description | Example |
|---|---|---|
| `i64` | 64-bit signed integer | `42`, `-7`, `0` |
| `i32` | 32-bit signed integer | `42 to i32` |
| `f64` | 64-bit floating-point | `3.14 to f64` |
| `f32` | 32-bit floating-point | `3.14`, `1.0` |
| `bool` | Boolean | `true`, `false` |
| `str` | String (UTF-8) | `"hello"` |

> **Note:** Float literals like `3.14` are `f32` by default. To get an `f64`, write `3.14 to f64`.

### 1.5.2 2.2 Bindings: `val` and `var`

Use `val` to create an immutable binding and `var` for a mutable one:

```
def demo() -> i64 {
    val x = 10         // immutable — cannot reassign
    var y = 20         // mutable — can be reassigned
    y = y + 5          // ok: y is now 25
    x + y              // returns 35
}
```

If you try to reassign a `val`, you get a compile error:

```
def bad() -> i64 {
    val x = 10
    x = 20   // ERROR: cannot assign to immutable binding 'x'
```

```
    x
}
```

### 1.5.3 2.3 Type Inference

IRIS infers types from context, so you rarely need to annotate them explicitly:

```
def inferred() -> i64 {
    val a = 10        // inferred as i64
    val b = 20        // inferred as i64
    a + b             // returns i64
}
```

You can add a type annotation after the binding name with `:`:

```
def annotated() -> f32 {
    val pi: f32 = 3.14159
    pi
}
```

### 1.5.4 2.4 Literals and Casts

Integer literals are `i64` by default. Float literals are `f32` by default.

To convert between types, use the `to` keyword:

```
def casts() -> f64 {
    val n: i64 = 42
    val f: f32 = 3.14
    val big: f64 = f to f64     // f32 -> f64
    val also: f64 = n to f64    // i64 -> f64
    also + big
}
```

Common cast patterns:

```
def cast_examples() -> i64 {
    val x: f64 = 9.7 to f64
    val rounded: i64 = floor(x) to i64     // 9
    rounded
}
```

### 1.5.5 2.5 Constants

Use `const` to define a module-level constant. Constants are always typed explicitly:

```
const MAX_SIZE: i64 = 1000
const PI: f64 = 3.14159265358979 to f64
const APP_NAME: str = "MyApp"

def show_constants() -> i64 {
```

```
    print(APP_NAME);
    MAX_SIZE
}
```

Constants are evaluated at compile time and can be used anywhere in the module.

### 1.5.6 2.6 Type Aliases

Use `type` to create an alias for an existing type:

```
type Index = i64
type Score = f64
type Name = str

def greet(name: Name) -> str {
    concat("Hello, ", name)
}
```

Type aliases are purely cosmetic — `Index` and `i64` are the same type to the compiler.

### 1.5.7 2.7 The Type System Overview

IRIS uses a structural, nominal type system:

- **Scalar types** (`i64`, `f32`, etc.) are value types — they are copied when assigned.
- **Records** (structs) are nominal — two records with the same fields but different names are different types.
- **Enums** (`choice`) are tagged unions — each variant can carry data.
- **Options** (`option<T>`) wrap a value that may or may not be present.
- **Results** (`result<T, E>`) represent success or failure.
- **Lists** (`list<T>`) are dynamic arrays.
- **Maps** (`map<K, V>`) are hash maps.
- **Tensors** (`tensor<f32, [M, N]>`) are N-dimensional arrays for ML.

### 1.5.8 Try It Yourself

1. Write a function that takes an `f32` and returns its square root as `f64`. (Hint: use `sqrt` then cast.)
2. Define a constant `GRAVITY: f64` for Earth's gravitational acceleration (9.81 m/s²) and write a function that computes the force on a given mass.
3. Create a type alias `Celsius = f64` and write a function that converts Celsius to Fahrenheit.

> **Common Mistakes:**
>
> - Forgetting that `3.14` is `f32`, not `f64`. If you pass it to a function expecting `f64`, you must write `3.14 to f64`.
> - Trying to reassign a `val` binding. Use `var` when you need mutation.
> - Mixing `i64` and `f64` in arithmetic without a cast. IRIS does not implicitly promote types.

---

# 1.6 Chapter 3: Functions

## 1.6.1 3.1 Defining Functions

The `def` keyword introduces a function:

```
def add(a: i64, b: i64) -> i64 {
    a + b
}
```

Every parameter has a name and a type separated by `:`. The return type follows `->`. The function body is enclosed in `{ }`.

## 1.6.2 3.2 Tail Expressions (No `return` Needed)

The last expression in a function body is implicitly returned. This is the *tail expression*:

```
def multiply(a: i64, b: i64) -> i64 {
    val product = a * b
    product          // tail expression — this is the return value
}
```

Most of the time, you do not need an explicit `return`. Note that `val product = a * b` has no semicolon? That's wrong — it's a statement, not a tail expression. Let's be precise:

```
def multiply(a: i64, b: i64) -> i64 {
    val product = a * b;   // statement — needs semicolon
    product                // tail expression — no semicolon
}
```

Actually, `val` bindings do need a semicolon. Let me show the complete picture:

- **Statements** (non-tail): need `;` at the end — `val x = 5;`, `print("hi");`, `f();`
- **Tail expression** (last line): no `;` — this is the return value

```
def example() -> i64 {
    val a = 10;          // statement: val binding
    val b = 20;          // statement: val binding
    print("computing");  // statement: side effect call
    a + b                // tail expression: no semicolon, this is returned
}
```

## 1.6.3 3.3 Early Return

Sometimes you want to return early from a function. Use `return`:

```
def safe_divide(a: i64, b: i64) -> i64 {
    if b == 0 {
        return 0;
    } else {
```

```
        a / b
    }
}
```

> **Note:** `return` exits the function immediately. It must be inside a statement context (followed by `;` if not at the end of a block).

### 1.6.4 3.4 Recursive Functions

Recursion works naturally in IRIS. Here's factorial:

```
def factorial(n: i64) -> i64 {
    if n <= 1 {
        1
    } else {
        n * factorial(n - 1)
    }
}

def main() -> i64 {
    print(factorial(10));
    0
}
```

And Fibonacci:

```
def fib(n: i64) -> i64 {
    if n <= 1 {
        n
    } else {
        fib(n - 1) + fib(n - 2)
    }
}

def main() -> i64 {
    print(fib(20));
    0
}
```

> **Note:** Recursive functions in IRIS are not tail-call optimized by default. For very deep recursion, prefer iterative approaches.

Here is an iterative Fibonacci that avoids stack overflow:

```
def fib_iter(n: i64) -> i64 {
    if n <= 1 {
```

```
        n
    } else {
        var a = 0;
        var b = 1;
        var i = 2;
        while i <= n {
            val tmp = a + b;
            a = b;
            b = tmp;
            i = i + 1
        }
        b
    }
}
```

### 1.6.5 3.5 Public Functions

By default, functions are private to their module. Use `pub def` to make a function accessible from other modules:

```
// In mylib.iris
pub def greet(name: str) -> str {
    concat("Hello, ", name)
}

// Private helper — not exported
def helper() -> i64 {
    42
}
```

When another file brings in `mylib`, only `greet` is accessible.

### 1.6.6 3.6 Default Parameters

Functions can have default parameter values:

```
def repeat_char(c: str, n: i64 = 3) -> str {
    repeat(c, n)
}

def main() -> i64 {
    print(repeat_char("-"));      // uses default n=3: "---"
    print(repeat_char("*", 5));   // overrides: "*****"
    0
}
```

### 1.6.7 3.7 Functions as First-Class Values

Functions can be stored and passed around using function types. The type `(i64, i64) -> i64` describes a function that takes two `i64` arguments and returns an `i64`:

```

```

```
def apply(f: (i64) -> i64, x: i64) -> i64 {
    f(x)
}

def double(x: i64) -> i64 {
    x * 2
}

def main() -> i64 {
    val result = apply(double, 21);
    print(result);   // prints 42
    0
}
```

### 1.6.8 Try It Yourself

1. Write a recursive function `power(base: i64, exp: i64) -> i64` that computes `base` raised to `exp`.
2. Write a function `clamp_score(score: f32, lo: f32 = 0.0, hi: f32 = 100.0) -> f32` with default bounds.
3. Write a function `sum_to(n: i64) -> i64` using a `while` loop (we will cover loops in chapter 4, but give it a try).

> **Common Mistakes:**
>
> - Forgetting the semicolon after non-tail statements inside a function. `print("hi")` without `;` will be parsed as the tail expression, making the function return type `unit` instead of what you intended.
> - Naming a function parameter the same as a built-in (like `len` or `print`). This shadows the built-in.
> - Confusing `return expr` (early exit) with just `expr` (tail expression). Both work, but `return` at the tail position is redundant and slightly less idiomatic.

---

## 1.7 Chapter 4: Control Flow

### 1.7.1 4.1 `if / else`

The `if` expression in IRIS **always** requires an `else` branch:

```
def abs_val(x: i64) -> i64 {
    if x < 0 { 0 - x } else { x }
}
```

`if` is an expression, meaning it produces a value. Both branches must produce the same type:

```
def classify(n: i64) -> str {
    if n > 0 {
        "positive"
    } else {
        if n < 0 {
            "negative"
        } else {
            "zero"
        }
    }
}
```

You can use `if` inline in expressions:

```
def max_of(a: i64, b: i64) -> i64 {
    val bigger = if a > b { a } else { b };
    bigger
}
```

> **Common Mistakes:** Omitting `else`. This is the single most common error for beginners. IRIS requires `else` because `if` produces a value — without `else`, the type of the expression is undefined.

### 1.7.2 4.2 `while` Loops

`while` loops repeat as long as a condition is true:

```
def count_down(from: i64) -> i64 {
    var n = from;
    while n > 0 {
        print(n);
        n = n - 1
    }
    0
}
```

Compute the sum of 1..100:

```
def sum_one_to_hundred() -> i64 {
    var total = 0;
    var i = 1;
    while i <= 100 {
        total = total + i;
        i = i + 1
    }
    total
}
```

### 1.7.3 4.3 `for` Range Loops

The `for i in start..end` loop iterates over the half-open range `[start, end)`:

```
def print_range() -> i64 {
    for i in 0..5 {
        print(i)
    }
    0
}
// prints: 0 1 2 3 4
```

For loops are clean and idiomatic when you need a counter:

```
def sum_range(n: i64) -> i64 {
    var total = 0;
    for i in 1..n + 1 {
        total = total + i
    }
    total
}
```

> **Note:** The range `start..end` is exclusive of `end`. To include `end`, write `start..end + 1`.

### 1.7.4 4.4 `loop` with `break`

The `loop` construct runs forever until you explicitly `break`:

```
def find_first_even(start: i64) -> i64 {
    var n = start;
    var result = 0;
    loop {
        if (n - (n / 2) * 2) == 0 {
            result = n;
            break
        } else {
            n = n + 1
        }
    }
    result
}
```

### 1.7.5 4.5 `break` and `continue`

Inside any loop, `break` exits the loop and `continue` skips the rest of the current iteration:

```
def skip_multiples_of_3(limit: i64) -> i64 {
    var printed = 0;
    for i in 1..limit + 1 {
```

```
        if (i - (i / 3) * 3) == 0 {
            continue
        } else {
            print(i);
            printed = printed + 1
        }
    }
    printed
}
```

### 1.7.6 4.6 Nested Loops

Loops can be nested freely:

```
def multiplication_table(n: i64) -> i64 {
    for i in 1..n + 1 {
        for j in 1..n + 1 {
            val product = i * j;
            print(product)
        }
    }
    0
}
```

### 1.7.7 4.7 Logical Operators

IRIS supports short-circuit logical operators:

- `&&` — logical AND (short-circuits: if left is false, right is not evaluated)
- `||` — logical OR (short-circuits: if left is true, right is not evaluated)

```
def check(x: i64, y: i64) -> bool {
    x > 0 && y > 0
}

def either_positive(x: i64, y: i64) -> bool {
    x > 0 || y > 0
}
```

### 1.7.8 Try It Yourself

1. Write a function `is_prime(n: i64) -> bool` using a `for` loop and `break`.
2. Write a function `collatz_steps(n: i64) -> i64` that counts how many steps the Collatz sequence takes to reach 1 (if n is even, divide by 2; if odd, multiply by 3 and add 1).
3. Write a function that prints a triangle of asterisks of height `h` (row 1 has one `*`, row 2 has two, etc.) using nested `for` loops.

> **Common Mistakes:**

> - Writing `if cond { body }` without an `else`. Always add `else { ... }`, even if it just returns `0`.
> - Off-by-one in ranges. `for i in 0..n` gives `n` iterations (0 through n-1), not n+1.
> - Using `%` for modulo — that is correct in IRIS. Do not look for `mod` keyword.

---

# 1.8 Chapter 5: Data Structures

## 1.8.1 5.1 Records

Records are named collections of fields, similar to structs in C or Rust:

```
record Point {
    x: f64,
    y: f64
}

def make_point(x: f64, y: f64) -> Point {
    Point { x: x, y: y }
}

def distance(p: Point) -> f64 {
    sqrt((p.x * p.x) + (p.y * p.y)) to f64
}

def main() -> i64 {
    val p = Point { x: 3.0 to f64, y: 4.0 to f64 };
    val d = distance(p);
    print(d);   // 5.0
    0
}
```

Records can contain any types, including other records:

```
record Color {
    r: f32,
    g: f32,
    b: f32
}

record Pixel {
    x: i64,
    y: i64,
    color: Color
}

def make_red_pixel(x: i64, y: i64) -> Pixel {
    Pixel {
        x: x,
```

```
        y: y,
        color: Color { r: 1.0, g: 0.0, b: 0.0 }
    }
}
```

### 1.8.2 5.2 Enums (`choice`)

Enums define a type with a fixed set of variants. Use `choice` to declare them:

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

def opposite(d: Direction) -> Direction {
    when d {
        Direction.North => Direction.South,
        Direction.South => Direction.North,
        Direction.East  => Direction.West,
        Direction.West  => Direction.East
    }
}
```

### 1.8.3 5.3 Pattern Matching with `when`

The `when` expression matches a value against its variants:

```
choice Shape {
    Circle,
    Square,
    Triangle
}

def sides(s: Shape) -> i64 {
    when s {
        Shape.Circle   => 0,
        Shape.Square   => 4,
        Shape.Triangle => 3
    }
}

def describe(s: Shape) -> str {
    when s {
        Shape.Circle   => "A round shape with no sides",
        Shape.Square   => "A four-sided shape",
        Shape.Triangle => "A three-sided shape"
    }
}
```

### 1.8.4 5.4 Tuples

Tuples are ordered collections of values with potentially different types. Access elements with `.0`, `.1`, `.2`, etc.:

```
def make_pair(a: i64, b: str) -> (i64, str) {
    (a, b)
}

def main() -> i64 {
    val pair = make_pair(42, "hello");
    val num = pair.0;   // 42
    val text = pair.1;  // "hello"
    print(num);
    print(text);
    0
}
```

Tuples are great for returning multiple values from a function:

```
def min_max(a: i64, b: i64, c: i64) -> (i64, i64) {
    val lo = if a < b { if a < c { a } else { c } } else { if b < c { b } else
{ c } };
    val hi = if a > b { if a > c { a } else { c } } else { if b > c { b } else
{ c } };
    (lo, hi)
}

def main() -> i64 {
    val result = min_max(7, 2, 9);
    print(result.0);   // 2
    print(result.1);   // 9
    0
}
```

### 1.8.5 5.5 Fixed Arrays

Arrays have a compile-time fixed size. The type `[T; N]` is an array of `N` elements of type `T`:

```
def sum_array() -> i64 {
    val nums: [i64; 5] = [10, 20, 30, 40, 50];
    var total = 0;
    for i in 0..5 {
        total = total + nums[i]
    }
    total
}
```

Arrays support element assignment (they are mutable by default):

```
```

```
def zero_fill(size: i64) -> i64 {
    val arr: [i64; 4] = [0, 0, 0, 0];
    for i in 0..4 {
        arr[i] = i * 2
    }
    arr[3]   // returns 6
}
```

### 1.8.6 5.6 Dynamic Lists

Lists are resizable arrays. `list()` creates an empty `list<i64>`. Use `list<T>()` for other element types:

```
def build_list() -> i64 {
    val nums = list();           // list<i64>
    push(nums, 10);
    push(nums, 20);
    push(nums, 30);
    print(list_len(nums));       // 3
    print(list_get(nums, 1));    // 20
    list_len(nums)
}
```

List operations:

```
def list_demo() -> i64 {
    val items = list();
    push(items, 5);
    push(items, 3);
    push(items, 8);
    push(items, 1);

    // Access by index
    val first = list_get(items, 0);     // 5

    // Modify by index
    list_set(items, 2, 100);

    // Length
    val n = list_len(items);       // 4

    // Pop from end
    val last = list_pop(items);    // returns last element

    n
}
```

For lists of strings, specify the type explicitly:

```
def string_list() -> i64 {
    val names = list();
```

```
    push(names, "Alice");
    push(names, "Bob");
    push(names, "Charlie");
    print(list_get(names, 0));   // Alice
    list_len(names)
}
```

### 1.8.7 5.7 Maps

Maps store key-value pairs. The `map<K, V>` type associates keys of type `K` with values of type `V`:

```
def word_count() -> i64 {
    val counts = map();
    map_set(counts, "apple", 3);
    map_set(counts, "banana", 7);
    map_set(counts, "cherry", 2);

    val found = map_get(counts, "banana");
    if is_some(found) {
        print(unwrap(found))   // prints 7
    } else {
        print(0)
    }
}
```

Map operations:

| Operation | Description |
|---|---|
| `map_set(m, k, v)` | Insert or update key `k` with value `v` |
| `map_get(m, k)` | Returns `option<V>` — some if key exists, none if not |
| `map_contains(m, k)` | Returns `bool` — true if key exists |
| `map_remove(m, k)` | Remove key `k` |
| `map_len(m)` | Number of entries |
| `map_keys(m)` | Returns `list<str>` of all keys |

### 1.8.8 5.8 Options

Options represent a value that may or may not be present. `option<T>` is either `some(v)` (contains a value) or `none` (no value):

```
def safe_head(lst: list<i64>) -> option<i64> {
    if list_len(lst) == 0 {
        none
    } else {
        some(list_get(lst, 0))
```

```
    }
}

def main() -> i64 {
    val lst = list();
    push(lst, 42);
    val head = safe_head(lst);
    if is_some(head) {
        print(unwrap(head))   // prints 42
    } else {
        print(-1)
    }
}
```

Common option functions:

| Function | Description |
|---|---|
| `some(v)` | Wrap value `v` in an option |
| `none` | The absent option value |
| `is_some(opt)` | Returns `bool` — is the option present? |
| `unwrap(opt)` | Extract the value (panics if none) |

> **Note:** `find(s, sub)` returns `option<i64>`. Always use `is_some()` to check before calling `unwrap()`. Do not compare the result with `< 0` — that is for C's `strstr`, not IRIS.

### 1.8.9 5.9 Results

Results represent either success or failure. `result<T, E>` is either `ok(v)` (success with value `v`) or `err(e)` (failure with error `e`):

```
def parse_age(s: str) -> result<i64, str> {
    val parsed = parse_i64(s);
    if is_some(parsed) {
        val age = unwrap(parsed);
        if age < 0 || age > 150 {
            err("age out of range")
        } else {
            ok(age)
        }
    } else {
        err("not a number")
    }
}

def main() -> i64 {
```

```
    val r = parse_age("25");
    if is_ok(r) {
        print(unwrap(r))   // 25
    } else {
        print("error")
    }
}
```

### 1.8.10 Try It Yourself

1. Define a `record Rectangle { width: f64, height: f64 }` and write functions `area(r: Rectangle) -> f64` and `perimeter(r: Rectangle) -> f64`.
2. Define a `choice Season { Spring, Summer, Autumn, Winter }` and write a function that returns the average temperature for each season.
3. Write a function that takes a `list<i64>` and returns a tuple `(i64, i64)` containing the minimum and maximum values.
4. Write a function `lookup(m: map<str, i64>, key: str, default: i64) -> i64` that returns the map value or a default if the key is missing.

> **Common Mistakes:**
>
> - Calling `unwrap()` on a `none` option. Always check `is_some()` first.
> - Forgetting that `list()` creates a `list<i64>`. For other types, use `list<str>()`, etc.
> - Mutating a `val`-bound list. Lists are reference types — even a `val` binding can mutate the list's contents. Use `val` when the binding itself won't change (you won't point it at a different list), and `var` when you might reassign the binding to a completely new list.

---

## 1.9 Chapter 6: Closures and Higher-Order Functions

### 1.9.1 6.1 Closure Syntax

A closure is an anonymous function that can capture values from its surrounding scope. The syntax is `|param: Type| expr`:

```
def main() -> i64 {
    val double = |x: i64| x * 2;
    val add_ten = |x: i64| x + 10;
    print(double(21));    // 42
    print(add_ten(32));   // 42
    0
}
```

For closures with multiple statements, use a block:

```
def main() -> i64 {
    val clamp_to_100 = |x: i64| {
        if x < 0 {
            0
        } else {
            if x > 100 {
                100
            } else {
                x
            }
        }
    };
    print(clamp_to_100(150));   // 100
    print(clamp_to_100(50));    // 50
    print(clamp_to_100(-5));    // 0
    0
}
```

### 1.9.2 6.2 Passing Closures as Arguments

Closures can be passed to functions using function type notation `(ParamType) -> ReturnType`:

```
def apply_twice(f: (i64) -> i64, x: i64) -> i64 {
    f(f(x))
}

def main() -> i64 {
    val triple = |x: i64| x * 3;
    print(apply_twice(triple, 2));   // 3*(3*2) = 18
    0
}
```

### 1.9.3 6.3 Implementing Map

Here is a `map` operation over a list — applies a function to every element:

```
def list_map(lst: list<i64>, f: (i64) -> i64) -> list<i64> {
    val result = list();
    val n = list_len(lst);
    for i in 0..n {
        push(result, f(list_get(lst, i)))
    }
    result
}

def main() -> i64 {
    val nums = list();
    push(nums, 1);
    push(nums, 2);
    push(nums, 3);
    push(nums, 4);
```

```
    push(nums, 5);

    val doubled = list_map(nums, |x: i64| x * 2);
    for i in 0..list_len(doubled) {
        print(list_get(doubled, i))
    }
    0
}
// prints: 2 4 6 8 10
```

### 1.9.4 6.4 Implementing Filter

```
def list_filter(lst: list<i64>, pred: (i64) -> bool) -> list<i64> {
    val result = list();
    val n = list_len(lst);
    for i in 0..n {
        val item = list_get(lst, i);
        if pred(item) {
            push(result, item)
        } else {
            0
        }
    }
    result
}

def main() -> i64 {
    val nums = list();
    push(nums, 1);
    push(nums, 2);
    push(nums, 3);
    push(nums, 4);
    push(nums, 5);
    push(nums, 6);

    val evens = list_filter(nums, |x: i64| (x - (x / 2) * 2) == 0);
    for i in 0..list_len(evens) {
        print(list_get(evens, i))
    }
    0
}
// prints: 2 4 6
```

### 1.9.5 6.5 Implementing Reduce / Fold

```
def list_reduce(lst: list<i64>, init: i64, f: (i64, i64) -> i64) -> i64 {
    var acc = init;
    val n = list_len(lst);
    for i in 0..n {
        acc = f(acc, list_get(lst, i))
    }
    acc
```

```
}

def main() -> i64 {
    val nums = list();
    push(nums, 1);
    push(nums, 2);
    push(nums, 3);
    push(nums, 4);
    push(nums, 5);

    val total = list_reduce(nums, 0, |acc: i64, x: i64| acc + x);
    val product = list_reduce(nums, 1, |acc: i64, x: i64| acc * x);
    print(total);    // 15
    print(product);  // 120
    0
}
```

### 1.9.6 6.6 Capture by Value

Closures capture variables from the surrounding scope by value at the point of closure creation:

```
def make_adder(n: i64) -> (i64) -> i64 {
    |x: i64| x + n
}

def main() -> i64 {
    val add5 = make_adder(5);
    val add10 = make_adder(10);
    print(add5(3));    // 8
    print(add10(3));   // 13
    0
}
```

This is a classic higher-order function pattern — `make_adder` returns a closure that "remembers" the `n` it was created with.

### 1.9.7 Try It Yourself

1. Write a `list_any(lst: list<i64>, pred: (i64) -> bool) -> bool` function that returns `true` if any element satisfies the predicate.
2. Write a `list_all(lst: list<i64>, pred: (i64) -> bool) -> bool` function that returns `true` if all elements satisfy the predicate.
3. Write a `compose(f: (i64) -> i64, g: (i64) -> i64) -> (i64) -> i64` function that returns a closure computing `f(g(x))`.

---

## 1.10 Chapter 7: String Processing

### 1.10.1 7.1 String Literals and Escapes

String literals are enclosed in double quotes. Escape sequences:

| Escape | Meaning |
|---|---|
| `\n` | Newline |
| `\t` | Tab |
| `\r` | Carriage return |
| `\\` | Literal backslash |
| `\"` | Literal double quote |

```
def main() -> i64 {
    print("Line one\nLine two");
    print("Tab\there");
    print("She said \"hello\"");
    0
}
```

### 1.10.2 7.2 F-Strings (String Interpolation)

F-strings let you embed expressions directly in strings using `{expr}`:

```
def greet(name: str, age: i64) -> str {
    f"Hello, {name}! You are {age} years old."
}

def main() -> i64 {
    val msg = greet("Alice", 30);
    print(msg);   // Hello, Alice! You are 30 years old.
    0
}
```

F-strings automatically convert embedded values to strings.

### 1.10.3 7.3 Built-in String Functions

| Function | Signature | Description |
|---|---|---|
| `len(s)` | `str -> i64` | Number of bytes in the string |
| `concat(a, b)` | `(str, str) -> str` | Concatenate two strings |
| `contains(s, sub)` | `(str, str) -> bool` | Does `s` contain `sub`? |
| `starts_with(s, p)` | `(str, str) -> bool` | Does `s` start with `p`? |
| `ends_with(s, s)` | `(str, str) -> bool` | Does `s` end with `s`? |
| `to_upper(s)` | `str -> str` | Uppercase |
| `to_lower(s)` | `str -> str` | Lowercase |
| `trim(s)` | `str -> str` | Strip leading/trailing whitespace |
| `repeat(s, n)` | `(str, i64) -> str` | Repeat `s` `n` times |
| `to_str(v)` | `T -> str` | Convert any value to string |
| `split(s, delim)` | `(str, str) -> list<str>` | Split by delimiter |
| `join(parts, delim)` | `(list<str>, str) -> str` | Join list with delimiter |
| `slice(s, start, end)` | `(str, i64, i64) -> str` | Substring |
| `find(s, sub)` | `(str, str) -> option<i64>` | Index of first occurrence |
| `str_replace(s, old, new)` | `(str, str, str) -> str` | Replace all occurrences |
| `parse_i64(s)` | `str -> option<i64>` | Parse integer |
| `parse_f64(s)` | `str -> option<f64>` | Parse float |

### 1.10.4 7.4 String Building Patterns

Building a string incrementally by concatenation:

```
def repeat_greeting(name: str, times: i64) -> str {
    var result = "";
    var i = 0;
    while i < times {
        result = concat(result, concat("Hello, ", concat(name, "! ")));
        i = i + 1
    }
    result
}
```

Using `join` with `split`:

```

```

```
def capitalize_words(s: str) -> str {
    val words = split(s, " ");
    val n = list_len(words);
    val out = list();
    for i in 0..n {
        val word = list_get(words, i);
        if len(word) == 0 {
            push(out, word)
        } else {
            val first = to_upper(slice(word, 0, 1));
            val rest = slice(word, 1, len(word));
            push(out, concat(first, rest))
        }
    }
    join(out, " ")
}

def main() -> i64 {
    print(capitalize_words("hello world from iris"));
    // Hello World From Iris
    0
}
```

### 1.10.5 7.5 Working with Split and Join

Split a CSV line and process fields:

```
def parse_csv_line(line: str) -> i64 {
    val fields = split(line, ",");
    val n = list_len(fields);
    val n_str = to_str(n);
    print(concat("Found ", concat(n_str, " fields")));
    for i in 0..n {
        val field = trim(list_get(fields, i));
        val i_str = to_str(i);
        print(concat("  [", concat(i_str, concat("] = ", field))))
    }
    n
}

def main() -> i64 {
    parse_csv_line("Alice, 30, Engineer, London")
}
```

Build a delimited string from a list:

```
def list_to_csv(items: list<str>) -> str {
    join(items, ",")
}

def main() -> i64 {
    val parts = list();
```

```
    push(parts, "name");
    push(parts, "age");
    push(parts, "city");
    print(list_to_csv(parts));   // name,age,city
    0
}
```

### 1.10.6 7.6 String Searching

```
def find_and_extract(text: str, marker: str) -> str {
    val pos = find(text, marker);
    if is_some(pos) {
        val idx = unwrap(pos);
        val after = slice(text, idx + len(marker), len(text));
        after
    } else {
        ""
    }
}

def main() -> i64 {
    val result = find_and_extract("key=value", "=");
    print(result);   // value
    0
}
```

### 1.10.7 Try It Yourself

1. Write a function `count_occurrences(text: str, target: str) -> i64` that counts how many times `target` appears in `text`.
2. Write a function `reverse_words(s: str) -> str` that reverses the order of words in a sentence.
3. Write a simple template engine: given a template like `"Hello, {name}!"` and a list of `(str, str)` substitutions, replace each `{key}` with its value.

> **Common Mistakes:**
>
> - Using `find(s, sub)` result directly as a number without checking `is_some()`. `find` returns `option<i64>`, not `i64`.
> - Confusing `len` (bytes) with character count. For ASCII strings, they are the same. For UTF-8 text with non-ASCII characters, `len` counts bytes.
> - Forgetting that `split` returns `list<str>`, not `list<i64>`. Use `list<str>()` type annotation when creating lists to hold the results.

---

# 1.11 Chapter 8: Error Handling

## 1.11.1 8.1 The `result<T, E>` Type

IRIS uses `result<T, E>` to represent operations that can fail. A result is either:

- `ok(v)` — success, containing value `v` of type `T`
- `err(e)` — failure, containing error `e` of type `E`

This pattern forces you to explicitly handle both success and failure cases, making errors visible and impossible to ignore accidentally.

## 1.11.2 8.2 Creating and Checking Results

```
def divide(a: f64, b: f64) -> result<f64, str> {
    if b == 0.0 {
        err("division by zero")
    } else {
        ok(a / b)
    }
}

def main() -> i64 {
    val r1 = divide(10.0 to f64, 2.0 to f64);
    val r2 = divide(10.0 to f64, 0.0 to f64);

    if is_ok(r1) {
        print(unwrap(r1))     // 5.0
    } else {
        print("error")
    };

    if is_ok(r2) {
        print(unwrap(r2))
    } else {
        print("error: division by zero")
    }
}
```

## 1.11.3 8.3 The `?` Operator

The `?` operator provides a shorthand for propagating errors. Inside a function that returns `result<T, E>`, writing `expr?` means: if `expr` is `err(e)`, return `err(e)` immediately; if it is `ok(v)`, continue with `v`.

```
def read_positive(s: str) -> result<i64, str> {
    val parsed = parse_i64(s);
    if is_some(parsed) {
        val n = unwrap(parsed);
        if n > 0 {
            ok(n)
        } else {
            err("must be positive")
        }
```

```
    } else {
        err("not a valid integer")
    }
}

def compute(a_str: str, b_str: str) -> result<i64, str> {
    val a = read_positive(a_str)?;
    val b = read_positive(b_str)?;
    ok(a + b)
}

def main() -> i64 {
    val r = compute("10", "20");
    if is_ok(r) {
        print(unwrap(r))   // 30
    } else {
        print("error")
    }
}
```

### 1.11.4 8.4 Pattern Matching Results with `when`

You can use `when` to match on a result and handle both cases expressively:

```
def process_file(path: str) -> str {
    val r = file_read_all(path);
    when r {
        ok(content) => concat("File contents: ", content),
        err(msg)    => concat("Failed to read file: ", msg)
    }
}
```

### 1.11.5 8.5 Chaining Operations

Results can be chained when each step depends on the previous success:

```
def parse_and_double(s: str) -> result<i64, str> {
    val parsed = parse_i64(s);
    if is_some(parsed) {
        ok(unwrap(parsed) * 2)
    } else {
        err(concat("cannot parse: ", s))
    }
}

def parse_two_and_add(a: str, b: str) -> result<i64, str> {
    val x = parse_and_double(a)?;
    val y = parse_and_double(b)?;
    ok(x + y)
}

def main() -> i64 {
    val good = parse_two_and_add("5", "3");
```

```
    val bad  = parse_two_and_add("5", "abc");

    if is_ok(good) { print(unwrap(good)) } else { print("failed") };
    // 16

    if is_ok(bad) { print(unwrap(bad)) } else { print("failed") }
    // failed
}
```

### 1.11.6 8.6 Combining Options and Results

Options and results often appear together. A common pattern is to convert `option<T>` into `result<T, E>`:

```
def option_to_result(opt: option<i64>, msg: str) -> result<i64, str> {
    if is_some(opt) {
        ok(unwrap(opt))
    } else {
        err(msg)
    }
}

def safe_parse(s: str) -> result<i64, str> {
    option_to_result(parse_i64(s), concat("not a number: ", s))
}
```

### 1.11.7 8.7 Panicking with `panic` and `assert`

For truly unrecoverable situations, use `panic` to abort with a message:

```
def must_be_positive(n: i64) -> i64 {
    if n <= 0 {
        panic(f"expected positive, got {n}")
    } else {
        n
    }
}
```

Use `assert` for debugging invariants:

```
def safe_sqrt(x: f64) -> f64 {
    assert(x >= 0.0 to f64);
    sqrt(x)
}
```

`assert(cond)` panics with a generic message if `cond` is false.

### 1.11.8 Try It Yourself

1. Write a `safe_list_get(lst: list<i64>, i: i64) -> result<i64, str>` that returns an error if the index is out of bounds.

2. Write a `parse_point(s: str) -> result<(i64, i64), str>` that parses a string like `"3,7"` into a tuple of two integers, returning an error if the format is wrong.
3. Chain three operations that can each fail using the `?` operator.

---

# 1.12 Chapter 9: Concurrency

## 1.12.1 9.1 Channels

IRIS provides channels for communicating between concurrent tasks. A channel is a typed queue: one task sends values, another receives them.

```
def main() -> i64 {
    val ch = channel();       // creates a channel<i64>
    send(ch, 42);
    val value = recv(ch);
    print(value);             // 42
    0
}
```

> **Note:** `channel()` creates an unbuffered, blocking channel. `send` blocks until the receiver is ready; `recv` blocks until a value is available.

## 1.12.2 9.2 Spawning Tasks with `spawn`

The `spawn` block runs its body as a concurrent task:

```
def main() -> i64 {
    val ch = channel();
    spawn {
        send(ch, 0);
        send(ch, 1);
        send(ch, 2);
        send(ch, 3);
        send(ch, 4)
    }
    for i in 0..5 {
        val v = recv(ch);
        print(v)
    }
    0
}
// prints: 0 1 2 3 4
```

## 1.12.3 9.3 Parallel For Loops

`par for` runs loop iterations in parallel using a thread pool:

```
def heavy_work(i: i64) -> i64 {
    // simulate work
```

```
    i * i
}

def main() -> i64 {
    val results: [i64; 10] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
    par for i in 0..10 {
        results[i] = heavy_work(i)
    }
    for i in 0..10 {
        print(results[i])
    }
    0
}
```

> **Note:** `par for` is ideal for embarrassingly parallel workloads where iterations do not depend on each other. The order of execution is not guaranteed.

### 1.12.4 9.4 Atomics: Thread-Safe Counters

When multiple concurrent tasks share a value, use atomics to avoid data races:

```
def main() -> i64 {
    val counter = atomic_new(0);

    par for i in 0..2000 {
        atomic_add(counter, 1)
    }

    val total = atomic_load(counter);
    print(total)
}
```

Atomic operations:

| Function | Description |
|---|---|
| `atomic_new(v)` | Create an atomic with initial value `v` |
| `atomic_load(a)` | Read the current value |
| `atomic_store(a, v)` | Write a new value |
| `atomic_add(a, v)` | Atomically add `v` and return the new value |

### 1.12.5 9.5 Producer-Consumer Pattern

A classic concurrency pattern where one task produces work and another consumes it:

```
def main() -> i64 {
    val ch = channel();
```

```
    // Producer: send squares 0..9 then sentinel -1
    spawn {
        for i in 0..10 {
            send(ch, i * i)
        }
        send(ch, -1)
    }
    // Consumer: accumulate until sentinel
    var total = 0;
    var running = true;
    while running {
        val v = recv(ch);
        if v < 0 {
            running = false
        } else {
            total = total + v
        }
    }
    print(total);   // sum of squares: 0+1+4+9+...+81 = 285
    0
}
```

### 1.12.6 9.6 Time Functions

For timing and delays:

```
def main() -> i64 {
    val t0 = time_now_ms();
    for i in 0..1000000 {
        val _ = i * i
    }
    val t1 = time_now_ms();
    val elapsed = t1 - t0;
    print(f"elapsed: {elapsed} ms");
    0
}
```

`sleep_ms(ms)` suspends the current task for the given milliseconds.

### 1.12.7 Try It Yourself

1. Write a program that uses `spawn` and a channel to compute the sum of a large list in two halves concurrently.
2. Use `par for` to fill a list with the first 100 Fibonacci numbers.
3. Create a bounded work queue: a channel through which tasks are sent, and three worker tasks that each receive and process them.

---

# 1.13 Chapter 10: Automatic Differentiation

## 1.13.1 10.1 Dual Numbers with `grad`

IRIS has built-in support for forward-mode automatic differentiation. The `grad_of` function computes the derivative of a closure at a given point:

```
def main() -> i64 {
    // Derivative of f(x) = x at x=3.0 is 1.0
    val g = grad_of(|x: f32| x, 3.0);
    print(g);   // 1.0
    0
}
```

`grad_of` takes a closure `(f32) -> f32` and a point, and returns the derivative as `f32`.

## 1.13.2 10.2 Computing Gradients

The power of `grad_of` comes from computing derivatives automatically through any computation built from `f32` operations:

```
// Compute df/dx of f(x) = x^2 at x=3 → 6.0
def main() -> i64 {
    val deriv = grad_of(|x: f32| x * x, 3.0);
    print(deriv);    // 6.0 (derivative of x^2 at x=3 is 2*3=6)
    0
}
```

For a function `f(x) = x^3 + 2*x + 1`, the derivative is `3*x^2 + 2`:

```
def poly_deriv(x_val: f64) -> f64 {
    // f(x) = x^3 + 2x + 1
    // f'(x) = 3x^2 + 2
    val x_sq = x_val * x_val;
    3.0 to f64 * x_sq + 2.0 to f64
}

def main() -> i64 {
    // At x = 2: f'(2) = 3*4 + 2 = 14
    print(poly_deriv(2.0 to f64));   // 14.0
    0
}
```

## 1.13.3 10.3 Simple Gradient Descent

Gradient descent uses derivatives to minimize a function. Here we minimize `f(x) = (x - 3)^2`, which has minimum at `x = 3`:

```
def f(x: f64) -> f64 {
    val diff = x - 3.0 to f64;
    diff * diff
}
```

```
def f_prime(x: f64) -> f64 {
    // Derivative of (x-3)^2 = 2*(x-3)
    2.0 to f64 * (x - 3.0 to f64)
}

def gradient_descent(start: f64, lr: f64, steps: i64) -> f64 {
    var x = start;
    for i in 0..steps {
        val grad_val = f_prime(x);
        x = x - lr * grad_val
    }
    x
}

def main() -> i64 {
    val result = gradient_descent(10.0 to f64, 0.1 to f64, 50);
    print(result);   // approximately 3.0
    0
}
```

### 1.13.4 10.4 Neural Network Gradient Descent

A more realistic example — linear regression with gradient descent:

```
// Linear regression: minimize (y - (w*x + b))^2 over w and b
def predict(w: f64, b: f64, x: f64) -> f64 {
    w * x + b
}

def loss(w: f64, b: f64, x: f64, y: f64) -> f64 {
    val diff = y - predict(w, b, x);
    diff * diff
}

// Compute gradients numerically (finite differences)
def grad_w(w: f64, b: f64, x: f64, y: f64) -> f64 {
    val h = 0.0001 to f64;
    (loss(w + h, b, x, y) - loss(w - h, b, x, y)) / (2.0 to f64 * h)
}

def grad_b(w: f64, b: f64, x: f64, y: f64) -> f64 {
    val h = 0.0001 to f64;
    (loss(w, b + h, x, y) - loss(w, b - h, x, y)) / (2.0 to f64 * h)
}

def train(epochs: i64) -> i64 {
    var w = 0.0 to f64;
    var b = 0.0 to f64;
    val lr = 0.01 to f64;

    // Training data: y = 2*x + 1
    val xs: [f64; 4] = [1.0 to f64, 2.0 to f64, 3.0 to f64, 4.0 to f64];
```

```
    val ys: [f64; 4] = [3.0 to f64, 5.0 to f64, 7.0 to f64, 9.0 to f64];

    for epoch in 0..epochs {
        var dw = 0.0 to f64;
        var db = 0.0 to f64;
        for i in 0..4 {
            dw = dw + grad_w(w, b, xs[i], ys[i]);
            db = db + grad_b(w, b, xs[i], ys[i])
        }
        w = w - lr * (dw / 4.0 to f64);
        b = b - lr * (db / 4.0 to f64)
    }

    print(f"w = {w}, b = {b}");
    // Should converge to w ≈ 2.0, b ≈ 1.0
    0
}

def main() -> i64 {
    train(200)
}
```

### 1.13.5 Try It Yourself

1. Use the gradient descent framework to minimize `f(x) = x^4 - 4*x^2`. Find both minima by starting from different initial points.
2. Implement a simple perceptron that learns the XOR function.

---

## 1.14 Chapter 11: Tensors and ML

### 1.14.1 11.1 Tensor Types

Tensors are the primary data structure for ML workloads. The type `tensor<f32, [M, K]>` describes a 2D tensor with `M` rows and `K` columns:

```
// A 3x3 identity-like matrix (type signature demonstration)
def make_identity(t: tensor<f32, [3, 3]>) -> tensor<f32, [3, 3]> {
    // Tensor operations use einsum notation
    einsum("ij->ij", t)
}
```

Tensor dimensions can be:

- **Integer literals**: `[3, 3]` — fixed size
- **Symbolic names**: `[M, K]` — size known at runtime, tracked symbolically

### 1.14.2 11.2 The `einsum` Intrinsic

Einstein summation notation provides a concise way to express tensor contractions. The first argument must be a string literal describing the operation:

```
```

```
// Matrix multiplication: C[m,n] = sum_k A[m,k] * B[k,n]
def matmul(a: tensor<f32, [M, K]>, b: tensor<f32, [K, N]>) -> tensor<f32, [M,
N]> {
    einsum("mk,kn->mn", a, b)
}

// Dot product: scalar = sum_i a[i] * b[i]
def dot(a: tensor<f32, [N]>, b: tensor<f32, [N]>) -> tensor<f32, []> {
    einsum("i,i->", a, b)
}

// Batch matrix multiply: C[b,m,n] = sum_k A[b,m,k] * B[b,k,n]
def batch_matmul(
    a: tensor<f32, [B, M, K]>,
    b: tensor<f32, [B, K, N]>
) -> tensor<f32, [B, M, N]> {
    einsum("bmk,bkn->bmn", a, b)
}
```

Common einsum patterns:

| Notation | Operation |
|---|---|
| `"ij,jk->ik"` | Matrix multiplication |
| `"i,i->"` | Dot product (scalar result) |
| `"ij->ji"` | Transpose |
| `"ii->"` | Trace (sum of diagonal) |
| `"ij->i"` | Row sum |
| `"ij->j"` | Column sum |

### 1.14.3 11.3 Building a Neural Network Layer

A linear (dense) layer computes `output = input @ weights + bias`:

```
record LinearLayer {
    weights: tensor<f32, [OUT, IN]>,
    bias: tensor<f32, [OUT]>
}

def linear_forward(
    lyr: LinearLayer,
    x: tensor<f32, [IN]>
) -> tensor<f32, [OUT]> {
    einsum("oi,i->o", lyr.weights, x)
    // Note: bias addition would be a separate step
}
```

### 1.14.4 11.4 Activation Functions

Activation functions are applied element-wise using built-in tensor operations:

```
// ReLU: f(x) = max(0, x)
def relu(x: tensor<f32, [N]>) -> tensor<f32, [N]> {
    einsum("i->i", x)
}

// Sigmoid: f(x) = 1 / (1 + exp(-x))
def sigmoid(x: tensor<f32, [N]>) -> tensor<f32, [N]> {
    einsum("i->i", x)
}

// Tanh activation
def tanh_act(x: tensor<f32, [N]>) -> tensor<f32, [N]> {
    einsum("i->i", x)
}
```

> **Note:** In practice, IRIS applies activation functions via runtime kernels. The `einsum` identity pass-through shown here demonstrates the type signatures. The compiler recognizes well-known activation patterns and emits optimized code.

### 1.14.5 11.5 A Simple Training Loop

Putting it together with a minimal training loop structure:

```
// Two-layer MLP for classification
record MLP {
    w1: tensor<f32, [H, IN]>,
    b1: tensor<f32, [H]>,
    w2: tensor<f32, [OUT, H]>,
    b2: tensor<f32, [OUT]>
}

def forward(mdl: MLP, x: tensor<f32, [IN]>) -> tensor<f32, [OUT]> {
    // Hidden layer
    val h = einsum("hi,i->h", mdl.w1, x);
    val h_act = einsum("i->i", h);
    // Output layer
    einsum("oh,h->o", mdl.w2, h_act)
}
```

> **Note:** In practice, training a neural network in IRIS involves loading data, computing losses, and applying gradient updates. The `einsum` operations form the computational graph; gradient computation can be done via the `grad` mechanism described in chapter 10 or via numerical finite differences.

### 1.14.6 11.6 Sparse Tensors

For data with many zero values, IRIS supports sparse representations:

```
def process_sparse(data: tensor<f32, [N]>) -> i64 {
    val sparse = sparsify(data);      // convert to sparse
    val dense = densify(sparse);      // convert back to dense
    0
}
```

Sparse tensors save memory and speed up operations when the data is predominantly zero (e.g., embeddings, adjacency matrices).

### 1.14.7 Try It Yourself

1. Write a function `softmax(x: tensor<f32, [N]>) -> tensor<f32, [N]>` that computes the softmax of a vector. (Hint: compute exp of each element, then divide by the sum.)
2. Write a function to compute the Frobenius norm of a matrix (square root of sum of squared elements) using `einsum`.
3. Design a three-layer MLP record type and write its `forward` function.

---

## 1.15 Chapter 12: Native Compilation

### 1.15.1 12.1 Building a Native Binary

The `iris build` command compiles your IRIS source to a native executable:

```
iris build myapp.iris -o myapp.exe
```

After building, run it directly:

```
myapp.exe
```

Or use `iris run` which compiles and runs in one step:

```
iris run myapp.iris
```

### 1.15.2 12.2 How the Compiler Pipeline Works

When you run `iris build`, the following steps happen:

1. **Parse**: The `.iris` source is tokenized and parsed into an AST.
2. **Lower**: The AST is compiled to IRIS IR (a block-parameter SSA form similar to MLIR).
3. **Optimize**: Several passes run:
   - `ValidatePass` — checks SSA invariants
   - `TypeInferPass` — ensures type consistency
   - `ConstFoldPass` — folds constant expressions
   - `DcePass` — dead code elimination
   - `CsePass` — common subexpression elimination
4. **LLVM IR**: The IR is translated to LLVM IR text.

5. **Compile**: `clang` compiles the LLVM IR to an object file.
6. **Link**: `gcc` links the object file with the IRIS C runtime to produce the final executable.

You can inspect the IR at each stage:

```
iris --emit ir myapp.iris        # print IRIS IR
iris --emit llvm myapp.iris      # print LLVM IR text
```

### 1.15.3 12.3 Inspecting LLVM IR

The `--emit llvm` flag prints the LLVM IR that will be compiled:

```
iris --emit llvm hello.iris
```

This is useful for debugging performance issues or understanding what the compiler generates.

### 1.15.4 12.4 Calling C Libraries with `extern`

IRIS can call C functions using `extern def` declarations:

```
// Declare C standard library functions
extern def strlen(s: str) -> i64
extern def puts(s: str) -> i64

def main() -> i64 {
    val msg = "Hello from C!";
    puts(msg);
    0
}
```

The `extern def` declaration tells IRIS the C function's name and signature. At link time, the function must be available in a linked library.

A more complete FFI example — calling a C math function:

```
// C's pow function from libm
extern def pow_c(base: f64, exp: f64) -> f64

def compute_power() -> i64 {
    val result = pow_c(2.0 to f64, 10.0 to f64);
    print(result);   // 1024.0
    0
}
```

### 1.15.5 12.5 The C Runtime

IRIS programs link against a small C runtime (`iris_runtime.c`) that provides:

- Memory allocation for lists, maps, channels, and other heap objects
- String operations
- Channel and threading primitives (using pthreads)
- Atomic operations

- I/O functions

You do not need to manage memory manually — the runtime handles allocation and a reference-counting scheme for heap objects.

### 1.15.6 12.6 Performance Tips

**Use fixed arrays for hot data paths**: `[T; N]` arrays are allocated on the stack and have no overhead. `list<T>` involves heap allocation.

```
// Fast: stack-allocated array
def sum_fixed() -> i64 {
    val data: [i64; 8] = [1, 2, 3, 4, 5, 6, 7, 8];
    var total = 0;
    for i in 0..8 {
        total = total + data[i]
    }
    total
}
```

**Minimize allocations in hot loops**: Avoid creating new lists or maps inside tight loops.

**Use `par for` for embarrassingly parallel workloads**: When iterations are independent, `par for` can use multiple CPU cores.

**Profile with timing**: Use `time_now_ms()` to measure how long sections of code take:

```
def benchmark() -> i64 {
    val t0 = time_now_ms();
    // ... work ...
    val t1 = time_now_ms();
    val elapsed = t1 - t0;
    print(f"elapsed: {elapsed}ms");
    0
}
```

### 1.15.7 Try It Yourself

1. Write a program, build it as a native binary, and run it. Measure the time to compute the 40th Fibonacci number both recursively and iteratively.
2. Declare and call a C function from your program (for example, `rand()` from the C standard library to generate random numbers).
3. Use `iris --emit ir` and `iris --emit llvm` to see what code a simple function generates.

---

## 1.16 Chapter 13: The Standard Library

IRIS ships with a standard library of `.iris` files that you can bring into your programs. Use the `bring` statement at the top of your file:

```
bring std.math
bring std.string
```

```
bring std.fmt
bring std.fs
```

### 1.16.1 13.1 `std.math` — Extended Math Functions

```
bring std.math

def main() -> i64 {
    print(gcd(48, 18));           // 6
    print(lcm(4, 6));             // 12
    print(abs_i64(-42));          // 42
    print(is_even(7));            // false
    print(is_odd(7));             // true
    print(clamp_i64(150, 0, 100)); // 100
    0
}
```

Available functions from `std.math`:

| Function | Description |
|---|---|
| `gcd(a, b)` | Greatest common divisor |
| `lcm(a, b)` | Least common multiple |
| `abs_i64(n)` | Absolute value for integers |
| `clamp_i64(x, lo, hi)` | Clamp integer to range |
| `min_i64(a, b)` | Integer minimum |
| `max_i64(a, b)` | Integer maximum |
| `sign_i64(n)` | Sign function: −1, 0, or 1 |
| `is_even(n)` | True if n is divisible by 2 |
| `is_odd(n)` | True if n is not divisible by 2 |

### 1.16.2 13.2 `std.string` — String Utilities

```
bring std.string

def main() -> i64 {
    val padded = pad_left("42", 6, "0");    // "000042"
    val trimmed = trim_start("  hello  ");   // "hello  " (only left)
    val ws = words("hello world foo");       // list<str>

    print(padded);
    print(len(ws));   // 3

    val joined = str_join(ws, "-");
    print(joined);    // "hello-world-foo"
```

```
    0
}
```

Available functions from `std.string`:

| Function | Description |
|---|---|
| `trim_start(s)` | Trim leading whitespace |
| `trim_end(s)` | Trim trailing whitespace |
| `pad_left(s, width, ch)` | Left-pad to width with character |
| `pad_right(s, width, ch)` | Right-pad to width with character |
| `words(s)` | Split on spaces, returns `list<str>` |
| `lines(s)` | Split on `\n`, returns `list<str>` |
| `str_join(parts, delim)` | Join list with delimiter |
| `is_empty(s)` | True if `len(s) == 0` |
| `str_repeat(s, n)` | Repeat string `n` times |

### 1.16.3 13.3 `std.fmt` — Formatting

The `fmt` module provides printf-style string formatting:

```
bring std.fmt

def main() -> i64 {
    // sprintf takes a format string and list<str> of pre-stringified args
    val args = list();
    push(args, to_str(42));
    push(args, to_str(3.14159));

    val s = sprintf("%05d %.2f", args);
    print(s);   // "00042 3.14"

    // Pad integers for table output
    val n = pad_int(7, 4);
    print(n);   // "   7"

    val z = zero_pad_int(42, 6);
    print(z);   // "000042"
    0
}
```

Available functions from `std.fmt`:

| Function | Description |
|---|---|
| `sprintf(fmt, args)` | Printf-style format string |
| `pad_int(n, width)` | Right-align integer in field |
| `zero_pad_int(n, width)` | Zero-pad integer |
| `left_align(s, width)` | Left-align string in field |
| `right_align(s, width)` | Right-align string in field |

Format specifiers: `%d`, `%s`, `%f`, `%g`, `%x`, `%i`, `%%` (literal `%`), with optional width (`%5d`), zero-padding (`%05d`), left-align (`%-8s`), and precision (`%.3f`).

### 1.16.4 13.4 `std.fs` — File System

```
bring std.fs

def main() -> i64 {
    // Write a file
    val ok = write_text("output.txt", "Hello, IRIS!\n");
    if ok {
        print("wrote file")
    } else {
        print("failed to write")
    };

    // Read a file
    val content = read_text("output.txt");
    print(content);

    // Check existence
    if path_exists("output.txt") {
        print("file exists")
    } else {
        print("no file")
    };

    // Read lines
    val lns = read_lines("output.txt");
    print(list_len(lns));
    0
}
```

Available functions from `std.fs`:

| Function | Description |
|---|---|
| `read_text(path)` | Read file as string (empty on error) |
| `write_text(path, content)` | Write string to file, returns `bool` |
| `path_exists(path)` | Check if file or directory exists |
| `read_lines(path)` | Read file as `list<str>` of lines |

### 1.16.5 13.5 Using `bring` in the REPL

In the REPL, use `:bring` to load a stdlib module:

```
>> :bring std.math
loaded: std.math
>> gcd(48, 18)
6
```

### 1.16.6 Try It Yourself

1. Use `std.fmt` to format a table of numbers with aligned columns.
2. Use `std.fs` to write a program that reads a text file, counts its words, and reports the result.
3. Use `std.string` to write a function that normalizes a string: trim whitespace, convert to lowercase, and replace multiple spaces with a single space.

---

## 1.17 Chapter 14: Tooling

### 1.17.1 14.1 The REPL in Depth

The IRIS REPL is a persistent interactive session. It supports multi-line input when you open a brace:

```
>> def greet(name: str) -> str {
...   concat("Hello, ", name)
... }
>> greet("World")
Hello, World
```

The REPL maintains state across inputs — definitions and bindings persist:

```
>> val x = 42
>> val y = 100
>> x + y
142
```

**REPL commands:**

Every command accepts a short alias shown in parentheses.

| Command | Alias | Description |
|---|---|---|
| `:help` | `:h` | Show the full command reference |
| `:env` | `:e` | List all active definitions and bindings |
| `:type <expr>` | `:t <expr>` | Show the inferred type of an expression |
| `:bring <mod>` | `:b <mod>` | Load a stdlib module (e.g. `:bring std.math`) |
| `:time` | | Show elapsed wall-clock time of the last evaluation |
| `:history` | | Show numbered input history for this session |
| `:clear` | | Clear the terminal screen |
| `:ir <expr>` | | Show the compiled IRIS IR for an expression |
| `:reset` | | Clear all session state and start fresh |
| `:quit` | `:q` | Exit the REPL (also Ctrl+D or Ctrl+C) |

**Commands in detail:**

`:help` / `:h` — Print the table of all available commands and their aliases.

`:env` / `:e` — List all active definitions and bindings in the current session:

```
>> def square(x: i64) -> i64 { x * x }
>> val n = 7
>> :env
  Definitions:
    def square(x: i64) -> i64 { x * x }
  Bindings:
    val n: i64 = 7
```

`:type <expr>` / `:t <expr>` — Discover the type of an expression without evaluating it:

```
>> :type 3 + 4
: i64
>> :type "hello"
: str
>> :type 3.14
: f32
>> :t true
: bool
```

`:bring <mod>` / `:b <mod>` — Load a stdlib module into the current session:

```
```

```
>> :bring std.math
loaded: std.math
>> gcd(12, 8)
4
```

`:time` — Show how long the last evaluation took:

```
>> val fib = 100000
>> :time
last evaluation took 0.124ms
```

`:history` — Show every input entered so far this session, numbered:

```
>> :history
  [1] val x = 42
  [2] val y = 100
  [3] x + y
```

`:clear` — Clear the terminal screen (sends ANSI escape codes).

`:ir <expr>` — Compile an expression and show the resulting IRIS IR:

```
>> :ir 2 + 3
function __eval_0() -> i64 {
  block0:
    %0 = const 2 : i64
    %1 = const 3 : i64
    %2 = add %0, %1 : i64
    return %2
}
```

`:reset` — Clear all session state and start fresh:

```
>> :reset
session cleared
```

`:quit` / `:q` — Exit the REPL (also Ctrl+D or Ctrl+C).

### 1.17.2 14.2 LSP Features

The IRIS Language Server Protocol implementation provides a rich editing experience in any LSP-compatible editor. Start the server with:

```
iris lsp
```

The server communicates over stdin/stdout using JSON-RPC (Language Server Protocol v3.17).

#### 1.17.2.1 Core Features

**Hover documentation**: Hover over a function call to see its signature and type information.

**Error diagnostics**: Errors appear as red/yellow underlines as you type. Each diagnostic carries a machine-readable code (e.g. `E0001`, `E0100`) for easy lookup. Hover to see the full error message with suggestions.

**Go to definition**: Ctrl+Click (or F12) on a function name to jump to where it is defined.

**Auto-completion**: Press Ctrl+Space to see completions for function names, field names, keywords, and bring-accessible stdlib symbols.

**Outline view**: The sidebar shows all functions and definitions in the current file.

**Signature help**: When you type a `(` after a function name, the parameter list and expected types appear.

**Document formatting**: Run "Format Document" to auto-format the current file.

#### 1.17.2.2 Code Actions (Quick Fixes)

When the editor underlines an error, a lightbulb icon appears with one-click fixes:

- **Add missing `bring`**: If you call `gcd(12, 8)` without importing `std.math`, the code action inserts `bring std.math` at the top of the file.
- **Prefix unused variable**: If a variable is declared but never used, the code action renames it with an `_` prefix to suppress the warning.
- **Insert closing brace**: If a block is left unterminated, the code action inserts the missing `}`.
- **Extract to variable**: Select an expression and extract it into a `val` binding.

#### 1.17.2.3 Inlay Hints

The LSP server can display inline type annotations next to `val` and `var` bindings that omit explicit types:

```
def example() -> i64 {
    val x = 42;          // inlay hint: `: i64`
    var name = "IRIS";   // inlay hint: `: str`
    0
}
```

Enable or disable this in your editor's settings.

#### 1.17.2.4 Find All References

Right-click an identifier and choose "Find All References" (or Shift+F12) to see every location in the current file where that name is used — definitions, calls, and assignments.

#### 1.17.2.5 Rename Symbol

Press F2 on a function or variable name to rename it everywhere it appears. The LSP server computes all occurrences and applies the rename atomically.

#### 1.17.2.6 Diagnostic Codes

Every error and warning carries a diagnostic code for quick reference:

| Code Range | Category |
|---|---|
| `E0001` – `E0006` | Parse errors (unexpected character, unterminated string, invalid literal, etc.) |
| `E0100` – `E0107` | Lowering errors (undefined variable, type mismatch, duplicate function, etc.) |
| `E0200` – `E0205` | Pass errors (use-before-def, multiple definition, type error, shape mismatch, etc.) |
| `E0300` | Code generation errors |
| `E0400` | Interpreter errors |
| `E0500` | I/O errors |

See Appendix E for detailed descriptions and fixes.

## 1.17.3 14.3 The Step Debugger (DAP)

IRIS implements the Debug Adapter Protocol (DAP), which integrates with VS Code's debugging panel and other compatible debuggers.

Start the debug adapter:

```
iris dap
```

From VS Code with the IRIS extension, press F5 to start a debugging session.

### 1.17.3.1 Core Debugging

- **Breakpoints**: Click in the gutter to set a breakpoint on a line.
- **Step over (F10)**: Execute the current line and move to the next.
- **Step into (F11)**: Step into a function call.
- **Step out (Shift+F11)**: Run until the current function returns.
- **Continue (F5)**: Resume execution until the next breakpoint.
- **Variables panel**: See all local variables and their current values.

### 1.17.3.2 Advanced Features

- **Step back**: Reverse one step to the previous statement. Useful for inspecting a value you just passed — press the step-back button in VS Code's debug toolbar or use the `stepBack` command.
- **Hover evaluation**: Hover over any variable or expression in the source while paused to see its current value in a tooltip. The debugger evaluates the expression in the current scope context.
- **Debug Console evaluation**: Type arbitrary IRIS expressions in the Debug Console to evaluate them in the current scope. Supports arithmetic, variable lookup, and simple function calls.
- **Call stack**: The Call Stack panel shows the full chain of function calls leading to the current position, with source locations for each frame.

- **Loaded sources**: View which source files the debugger has loaded via the "Loaded Sources" panel.
- **Exception info**: When a runtime error occurs, the debugger reports exception details including the error description and break mode so you can inspect the program state at the point of failure.

## 1.17.4 14.4 The VS Code Extension

The official IRIS VS Code extension (`iris-lang`) bundles the LSP client, DAP client, and additional editor features.

### 1.17.4.1 Installation

```
code --install-extension iris-lang-0.2.0.vsix
```

Or install from the Extensions panel in VS Code by searching for "IRIS Language".

### 1.17.4.2 Features

- **Syntax highlighting**: Full TextMate grammar for `.iris` files — keywords, types, strings, comments, numbers, and operators.
- **Error diagnostics**: Real-time error and warning underlines powered by the LSP server.
- **Code actions**: Lightbulb quick fixes appear automatically for common errors.
- **Inlay hints**: Inline type annotations for `val` / `var` bindings.
- **Go to definition, Find References, Rename**: Standard IDE navigation.
- **Debugging**: Press F5 to launch a debug session with full breakpoint, step, and variable inspection support.
- **Server status**: The status bar shows the IRIS language server state. Click to see options:
  - *Restart Server* — restart the LSP server without reloading the window.
  - *Stop Server* — stop the language server.
  - *Show Output* — view the server's log output channel.
- **Execution timing**: After running or building an IRIS file, the output channel shows the elapsed time.

### 1.17.4.3 Extension Settings

| Setting | Default | Description |
|---|---|---|
| `iris.compilerPath` | `iris` | Path to the `iris` executable |
| `iris.enableInlayHints` | `true` | Show inline type annotations |
| `iris.maxDiagnostics` | `100` | Maximum number of diagnostics per file |

## 1.17.5 14.5 IR Inspection

The `--emit` flag controls what the compiler outputs instead of running the program:

```
iris --emit ir file.iris       # IRIS IR (human-readable SSA form)
iris --emit llvm file.iris     # LLVM IR text (.ll format)
```

Example IR output for a simple addition function:

```
// IRIS IR for: def add(a: i64, b: i64) -> i64 { a + b }

function add(a: i64, b: i64) -> i64 {
  block0:
    %2 = add %0, %1 : i64
    return %2
}
```

This is useful for:

- Understanding how the optimizer transforms your code
- Debugging unexpected behavior
- Learning how the compiler works

### 1.17.6 14.5 Optimization Passes

The compiler runs a pipeline of optimization passes. You can see the IR after each pass with `--dump-ir-after`:

```
iris --emit ir --dump-ir-after const_fold file.iris
```

Pass pipeline:

1. **ValidatePass** — SSA structural validation (catches malformed IR)
2. **TypeInferPass** — Type consistency (checks binary operand types match)
3. **ConstFoldPass** — Constant folding (e.g., `2 + 3` → `5` at compile time)
4. **OpExpandPass** — Expands activation calls to tensor operations
5. **DcePass** — Dead code elimination (removes unused computations)
6. **CsePass** — Common subexpression elimination (deduplicates repeated computations)
7. **ShapeCheckPass** — Tensor shape consistency and einsum notation validation

### 1.17.7 Try It Yourself

1. Open the REPL and experiment with `:type` to learn what type various expressions have.
2. Write a function with a bug, then use the DAP debugger to step through and find it.
3. Use `iris --emit ir` on a function with constant expressions and observe how the `ConstFoldPass` eliminates them.

---

## 1.18 Chapter 15: Building Real Programs

### 1.18.1 15.1 Project Layout

A typical IRIS project looks like this:

```
myproject/
  src/
    main.iris          # entry point
    utils.iris         # utility functions
    models.iris        # data model definitions
  data/
    input.txt
```

```
  out/
    myproject.exe      # compiled binary
```

IRIS does not have a built-in build system or package manager. You build from the entry point:

```
iris build src/main.iris -o out/myproject.exe
```

### 1.18.2 15.2 Multi-File Programs with `bring`

The `bring` statement imports another IRIS file. All `pub def` functions from that file become available:

```
// src/utils.iris
pub def clamp(x: i64, lo: i64, hi: i64) -> i64 {
    if x < lo { lo } else { if x > hi { hi } else { x } }
}

pub def square(x: i64) -> i64 {
    x * x
}
```

```
// src/main.iris
bring utils

def main() -> i64 {
    print(clamp(150, 0, 100));   // 100
    print(square(7));            // 49
    0
}
```

Only `pub def` functions are exported. Private helpers stay private to their file.

> **Note:** ALL helper functions in a file that you want to use from other files must be `pub def`.

### 1.18.3 15.3 Writing a Command-Line Tool

A number-guessing game as a complete command-line program:

```
def main() -> i64 {
    // Simple "guess the number" game
    val secret = 42;    // in a real game, use a random number
    print("Guess a number between 1 and 100:");

    var guesses = 0;
    var found = false;
    while found == false {
```

```
        val line = read_line();
        val parsed = parse_i64(trim(line));
        if is_some(parsed) {
            val guess = unwrap(parsed);
            guesses = guesses + 1;
            if guess < secret {
                print("Too low! Try again:")
            } else {
                if guess > secret {
                    print("Too high! Try again:")
                } else {
                    found = true;
                    print(f"Correct! You got it in {guesses} guesses.")
                }
            }
        } else {
            print("Please enter a valid number:")
        }
    }
    0
}
```

### 1.18.4 15.4 Writing a Word-Count Tool

A more practical command-line tool — counting words in a file:

```
bring std.string
bring std.fs

def count_words_in_text(text: str) -> i64 {
    val ws = words(text);
    list_len(ws)
}

def count_lines_in_text(text: str) -> i64 {
    val ls = lines(text);
    list_len(ls)
}

def main() -> i64 {
    val args = process_args();
    if list_len(args) < 2 {
        print("Usage: wc <filename>");
        1
    } else {
        val filename = list_get(args, 1);
        val content = read_text(filename);
        if len(content) == 0 {
            print(f"Could not read file: {filename}");
            1
        } else {
            val line_count = count_lines_in_text(content);
            val word_count = count_words_in_text(content);
```

```
            val byte_count = len(content);
            print(f"{line_count} lines, {word_count} words, {byte_count}
bytes");
            0
        }
    }
}
```

Build and run:

```
iris build wc.iris -o wc.exe
wc.exe myfile.txt
```

### 1.18.5 15.5 A Simple TCP Echo Server

IRIS has built-in TCP networking:

```
def handle_connection(conn: i64) -> i64 {
    var running = true;
    while running {
        val line = tcp_read(conn);
        if len(line) == 0 {
            running = false
        } else {
            val response = concat("echo: ", concat(line, "\n"));
            tcp_write(conn, response)
        }
    }
    tcp_close(conn);
    0
}

def main() -> i64 {
    val port = 8080;
    val listener = tcp_listen(port);
    print(f"Listening on port {port}...");

    // Accept one connection for demonstration
    val conn = tcp_accept(listener);
    print("Connection accepted");
    handle_connection(conn);
    tcp_close(listener);
    0
}
```

Connect to test it with `telnet localhost 8080` or `nc localhost 8080`.

### 1.18.6 15.6 Performance Profiling

Use `time_now_ms()` to build simple profiling wrappers:

```

```

```
def main() -> i64 {
    // Profile different implementations
    val t0 = time_now_ms();
    val r1 = fib_recursive(35);
    val t1 = time_now_ms();

    val t2 = time_now_ms();
    val r2 = fib_iter(35);
    val t3 = time_now_ms();

    val time_recursive = t1 - t0;
    val time_iterative = t3 - t2;
    print(f"Recursive: {r1} in {time_recursive}ms");
    print(f"Iterative: {r2} in {time_iterative}ms");
    0
}

def fib_recursive(n: i64) -> i64 {
    if n <= 1 { n } else { fib_recursive(n-1) + fib_recursive(n-2) }
}

def fib_iter(n: i64) -> i64 {
    if n <= 1 {
        n
    } else {
        var a = 0;
        var b = 1;
        var i = 2;
        while i <= n {
            val tmp = a + b;
            a = b;
            b = tmp;
            i = i + 1
        }
        b
    }
}
```

### 1.18.7 15.7 A Key-Value Store Server

A simple in-memory key-value store served over TCP:

```
def parse_command(line: str) -> (str, str, str) {
    val parts = split(trim(line), " ");
    val n = list_len(parts);
    val cmd = if n > 0 { list_get(parts, 0) } else { "" };
    val key = if n > 1 { list_get(parts, 1) } else { "" };
    val val_ = if n > 2 { list_get(parts, 2) } else { "" };
    (cmd, key, val_)
}

def handle_cmd(store: map<str, str>, conn: i64, cmd: str, key: str, value: str)
-> bool {
```

```
    when cmd {
        "SET" => {
            map_set(store, key, value);
            tcp_write(conn, "OK\n");
            true
        },
        "GET" => {
            val found = map_get(store, key);
            if is_some(found) {
                tcp_write(conn, concat(unwrap(found), "\n"))
            } else {
                tcp_write(conn, "NIL\n")
            };
            true
        },
        "DEL" => {
            map_remove(store, key);
            tcp_write(conn, "OK\n");
            true
        },
        "QUIT" => {
            tcp_write(conn, "BYE\n");
            false
        },
        _ => {
            tcp_write(conn, "ERR unknown command\n");
            true
        }
    }
}

def main() -> i64 {
    val store = map();
    val listener = tcp_listen(7777);
    print("KV store listening on port 7777...");

    val conn = tcp_accept(listener);
    var running = true;
    while running {
        val line = tcp_read(conn);
        if len(line) == 0 {
            running = false
        } else {
            val cmd_tuple = parse_command(line);
            val cmd = cmd_tuple.0;
            val key = cmd_tuple.1;
            val value = cmd_tuple.2;
            running = handle_cmd(store, conn, cmd, key, value)
        }
    }
    tcp_close(conn);
    tcp_close(listener);
    0
}
```

### 1.18.8 Try It Yourself

1. Extend the word-count tool to also count unique words using a `map<str, i64>`.
2. Build a simple calculator that reads expressions like `3 + 4` from stdin and prints the result.
3. Add a `KEYS` command to the KV store server that lists all stored keys.

---

## 1.19 Chapter 16: Working with Databases

IRIS includes built-in support for **SQLite** databases. You can create, query, and manage local databases without importing any libraries — the four database builtins are part of the language.

### 1.19.1 16.1 The Database API

| Function | Signature | Description |
|---|---|---|
| `db_open` | `db_open(path: str) -> i64` | Open (or create) a SQLite database file. Returns a handle. |
| `db_exec` | `db_exec(db: i64, sql: str) -> i64` | Execute a statement (CREATE, INSERT, UPDATE, DELETE). Returns 0 on success, −1 on error. |
| `db_query` | `db_query(db: i64, sql: str) -> list<list<str>>` | Execute a SELECT query. Returns a list of rows, each row a list of string columns. |
| `db_close` | `db_close(db: i64) -> i64` | Close the database handle. Returns 0. |

All values returned by `db_query` are strings — you convert to numbers with `to_i64()` or `to_f64()` as needed.

### 1.19.2 16.2 Creating a Database and Table

```
def main() -> i64 {
    val db = db_open("app.db");
    db_exec(db, "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT, age INTEGER)");
    print("Table created");
    db_close(db)
}
```

If the file `app.db` does not exist, `db_open` creates it automatically. The handle is an opaque integer — pass it to every subsequent database call.

### 1.19.3 16.3 Inserting Data

```
def main() -> i64 {
    val db = db_open("app.db");
    db_exec(db, "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY
```

```
AUTOINCREMENT, name TEXT, age INTEGER)");
    db_exec(db, "INSERT INTO users (name, age) VALUES ('Alice', 30)");
    db_exec(db, "INSERT INTO users (name, age) VALUES ('Bob', 25)");
    db_exec(db, "INSERT INTO users (name, age) VALUES ('Carol', 28)");
    print("Inserted 3 users");
    db_close(db)
}
```

Each `db_exec` call runs a single SQL statement. Check the return value: 0 means success, −1 means the statement failed.

### 1.19.4 16.4 Querying Data

```
def main() -> i64 {
    val db = db_open("app.db");
    db_exec(db, "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT, age INTEGER)");
    db_exec(db, "INSERT INTO users (name, age) VALUES ('Alice', 30)");
    db_exec(db, "INSERT INTO users (name, age) VALUES ('Bob', 25)");
    val rows = db_query(db, "SELECT name, age FROM users");
    val i = 0;
    val n = list_len(rows);
    for idx in 0..n {
        val row = list_get(rows, idx);
        val name = list_get(row, 0);
        val age = list_get(row, 1);
        print(name);
        print(age)
    }
    db_close(db)
}
```

`db_query` returns a `list<list<str>>`. Each inner list is one row. Column values are always strings — use `to_i64()` or `to_f64()` if you need numeric types.

### 1.19.5 16.5 Updating and Deleting

```
def main() -> i64 {
    val db = db_open("app.db");
    db_exec(db, "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT, age INTEGER)");
    db_exec(db, "INSERT INTO users (name, age) VALUES ('Alice', 30)");
    db_exec(db, "UPDATE users SET age = 31 WHERE name = 'Alice'");
    db_exec(db, "DELETE FROM users WHERE name = 'Alice'");
    db_close(db)
}
```

UPDATE and DELETE are executed with `db_exec` just like INSERT and CREATE.

### 1.19.6 16.6 Error Handling

Always check the return value of `db_exec`:

```
def main() -> i64 {
    val db = db_open("app.db");
    val result = db_exec(db, "THIS IS NOT VALID SQL");
    if result == -1 {
        print("SQL error")
    };
    db_close(db)
}
```

If `db_open` fails (e.g. invalid path), it returns 0. Always verify the handle before using it.

### 1.19.7 16.7 A Complete Example: Task Manager

Here is a small task-management database that creates a table, inserts tasks, marks one complete, and queries the results:

```
def main() -> i64 {
    val db = db_open("tasks.db");
    db_exec(db, "CREATE TABLE IF NOT EXISTS tasks (id INTEGER PRIMARY KEY
AUTOINCREMENT, title TEXT, done INTEGER DEFAULT 0)");
    db_exec(db, "DELETE FROM tasks");
    db_exec(db, "INSERT INTO tasks (title) VALUES ('Write docs')");
    db_exec(db, "INSERT INTO tasks (title) VALUES ('Fix bug')");
    db_exec(db, "INSERT INTO tasks (title) VALUES ('Add tests')");
    db_exec(db, "UPDATE tasks SET done = 1 WHERE title = 'Fix bug'");
    val rows = db_query(db, "SELECT title, done FROM tasks ORDER BY id");
    val n = list_len(rows);
    for idx in 0..n {
        val row = list_get(rows, idx);
        val title = list_get(row, 0);
        val done = list_get(row, 1);
        print(title);
        print(done)
    }
    db_close(db)
}
```

### 1.19.8 16.8 Best Practices

- **Always close** the database handle with `db_close` when you are finished.
- **Use** `IF NOT EXISTS` on CREATE TABLE so your program can run more than once.
- **Check return codes** from `db_exec` — a return of −1 indicates an error.
- **Delete or clean up** test databases after use (`db_exec(db, "DROP TABLE ...")` or delete the file).
- **All query values are strings** — convert with `to_i64()` or `to_f64()` when needed.

### 1.19.9 Try It Yourself

1. Build a contacts database that stores name, email, and phone number. Write functions to add, search, and delete contacts.
2. Create a simple inventory system: insert items with name, quantity, and price, then query for items below a certain stock level.

3. Write a program that imports data from a file (using `file_read_all`) and inserts each line as a row in a database table.

---

## 1.20 Appendix A: Language Grammar (BNF)

The following is an informal BNF grammar for IRIS. `*` means zero or more, `?` means optional, `|` means alternative.

```
module      ::= item*
item        ::= def_item | record_def | choice_def | const_def
              | type_alias | extern_def | bring_stmt | trait_def | impl_def

bring_stmt  ::= "bring" bring_path
bring_path  ::= IDENT ("." IDENT)*

const_def   ::= "const" IDENT ":" type "=" expr

type_alias  ::= "type" IDENT "=" type

extern_def  ::= "extern" "def" IDENT "(" param_list ")" "->" type

def_item    ::= ("pub")? ("async")? "def" IDENT "(" param_list ")" "->" type
block
param_list  ::= (param ("," param)*)?
param       ::= IDENT ":" type ("=" expr)?

record_def  ::= "record" IDENT "{" (field_def ("," field_def)*)? "}"
field_def   ::= IDENT ":" type

choice_def  ::= "choice" IDENT "{" (variant ("," variant)*)? "}"
variant     ::= IDENT ("(" type ")")?

block       ::= "{" stmt* expr? "}"
stmt        ::= ("val" | "var") IDENT (":" type)? "=" expr ";"
              | expr ";"

expr        ::= logic_expr ("to" type)?
              | "return" expr ";"
              | closure_expr

logic_expr  ::= cmp_expr (("&&" | "||") cmp_expr)*
cmp_expr    ::= add_expr (("==" | "!=" | "<" | "<=" | ">" | ">=") add_expr)*
add_expr    ::= mul_expr (("+" | "-") mul_expr)*
mul_expr    ::= unary_expr (("*" | "/" | "%") unary_expr)*
unary_expr  ::= ("-" | "!") unary_expr | postfix_expr
postfix_expr ::= primary ("." field_or_index | "[" expr "]" | "(" arg_list ")" |
"?" )*

primary     ::= INT_LIT | FLOAT_LIT | BOOL_LIT | STRING_LIT | FSTRING_LIT
              | "none" | "true" | "false"
              | IDENT ("(" arg_list ")")?
              | "(" expr ("," expr)* ")"          // tuple or grouped expr
```

```
              | "[" expr ("," expr)* "]"           // array literal
              | "if" expr block "else" block       // if expression
              | "when" expr "{" when_arm* "}"      // pattern match
              | "loop" block
              | "while" expr block
              | "for" IDENT "in" expr ".." expr block
              | "par" "for" IDENT "in" expr ".." expr block
              | "spawn" block
              | "await" expr
              | block

closure_expr ::= "|" (closure_param ("," closure_param)*)? "|" (expr | block)
closure_param ::= IDENT ":" type

when_arm    ::= pattern "=>" expr ","?
pattern     ::= IDENT "." IDENT                   // enum variant
              | IDENT "." IDENT "(" IDENT ")"     // variant with binding
              | "ok" "(" IDENT ")"                // result ok
              | "err" "(" IDENT ")"               // result err

type        ::= "i64" | "i32" | "f64" | "f32" | "bool" | "str"
              | "tensor" "<" scalar_type "," "[" dim_list "]" ">"
              | "option" "<" type ">"
              | "result" "<" type "," type ">"
              | "list" "<" type ">"
              | "map" "<" type "," type ">"
              | "channel" "<" type ">"
              | "(" type ("," type)* ")"           // tuple type
              | "[" type ";" INT_LIT "]"           // array type
              | "(" (type ("," type)*)? ")" "->" type  // function type
              | IDENT                              // named type or alias

arg_list    ::= (expr ("," expr)*)?
dim_list    ::= (dim ("," dim)*)?
dim         ::= INT_LIT | IDENT
```

---

# 1.21 Appendix B: Built-in Functions Reference

## 1.21.1 Math

| Function | Signature | Description |
|---|---|---|
| `sin(x)` | `f32 -> f32` | Sine |
| `cos(x)` | `f32 -> f32` | Cosine |
| `tan(x)` | `f32 -> f32` | Tangent |
| `exp(x)` | `f32 -> f32` | e^x |
| `log(x)` | `f32 -> f32` | Natural logarithm |
| `log2(x)` | `f32 -> f32` | Base-2 logarithm |
| `sqrt(x)` | `f32 -> f32` | Square root |
| `abs(x)` | `f32 -> f32` | Absolute value |
| `floor(x)` | `f32 -> f32` | Floor |
| `ceil(x)` | `f32 -> f32` | Ceiling |
| `round(x)` | `f32 -> f32` | Round to nearest |
| `sign(x)` | `f32 -> f32` | −1, 0, or 1 |
| `pow(base, exp)` | `(f32, f32) -> f32` | Power |
| `min(a, b)` | `(f32, f32) -> f32` | Minimum |
| `max(a, b)` | `(f32, f32) -> f32` | Maximum |
| `clamp(x, lo, hi)` | `(f32, f32, f32) -> f32` | Clamp to range |

### 1.21.2 String

| Function | Signature | Description |
|---|---|---|
| `len(s)` | `str -> i64` | Byte length |
| `concat(a, b)` | `(str, str) -> str` | Concatenate |
| `contains(s, sub)` | `(str, str) -> bool` | Substring test |
| `starts_with(s, p)` | `(str, str) -> bool` | Prefix test |
| `ends_with(s, p)` | `(str, str) -> bool` | Suffix test |
| `to_upper(s)` | `str -> str` | Uppercase |
| `to_lower(s)` | `str -> str` | Lowercase |
| `trim(s)` | `str -> str` | Strip whitespace |
| `repeat(s, n)` | `(str, i64) -> str` | Repeat `n` times |
| `to_str(v)` | `T -> str` | Convert to string |
| `split(s, d)` | `(str, str) -> list<str>` | Split by delimiter |
| `join(lst, d)` | `(list<str>, str) -> str` | Join with delimiter |
| `slice(s, a, b)` | `(str, i64, i64) -> str` | Substring `[a, b)` |
| `find(s, sub)` | `(str, str) -> option<i64>` | First occurrence index |
| `str_replace(s, a, b)` | `(str, str, str) -> str` | Replace all occurrences |
| `parse_i64(s)` | `str -> option<i64>` | Parse integer |
| `parse_f64(s)` | `str -> option<f64>` | Parse float |

### 1.21.3 I/O

| Function | Signature | Description |
|---|---|---|
| `print(v)` | `T -> ()` | Print to stdout with newline |
| `read_line()` | `() -> str` | Read line from stdin |
| `read_i64()` | `() -> i64` | Read and parse integer from stdin |
| `read_f64()` | `() -> f64` | Read and parse float from stdin |

### 1.21.4 List

| Function | Signature | Description |
|---|---|---|
| `list()` | `() -> list<i64>` | New empty list |
| `list<T>()` | `() -> list<T>` | New empty typed list |
| `push(lst, v)` | `(list<T>, T) -> ()` | Append element |
| `list_pop(lst)` | `list<T> -> T` | Remove and return last element |
| `list_len(lst)` | `list<T> -> i64` | Number of elements |
| `list_len(lst)` | `list<T> -> i64` | Same as `len` |
| `list_get(lst, i)` | `(list<T>, i64) -> T` | Get by index (panics if OOB) |
| `list_set(lst, i, v)` | `(list<T>, i64, T) -> ()` | Set by index |
| `list_contains(lst, v)` | `(list<T>, T) -> bool` | Membership test |
| `list_sort(lst)` | `list<T> -> ()` | Sort in-place |
| `list_concat(a, b)` | `(list<T>, list<T>) -> list<T>` | Concatenate two lists |
| `list_slice(lst, a, b)` | `(list<T>, i64, i64) -> list<T>` | Slice `[a, b)` |

### 1.21.5 Map

| Function | Signature | Description |
|---|---|---|
| `map<K,V>()` | `() -> map<K,V>` | New empty map |
| `map_set(m, k, v)` | Side effect | Insert/update |
| `map_get(m, k)` | `option<V>` | Lookup |
| `map_contains(m, k)` | `bool` | Key exists? |
| `map_remove(m, k)` | Side effect | Remove key |
| `map_len(m)` | `i64` | Number of entries |
| `map_keys(m)` | `list<str>` | All keys |
| `map_values(m)` | `list<V>` | All values |

### 1.21.6 Option

| Function | Signature | Description |
|---|---|---|
| `some(v)` | `T -> option<T>` | Wrap in Some |
| `none` | `option<T>` | Absent value |
| `is_some(opt)` | `option<T> -> bool` | Has value? |
| `unwrap(opt)` | `option<T> -> T` | Extract (panics on none) |

### 1.21.7 Result

| Function | Signature | Description |
|---|---|---|
| `ok(v)` | `T -> result<T,E>` | Success value |
| `err(e)` | `E -> result<T,E>` | Error value |
| `is_ok(r)` | `result<T,E> -> bool` | Is success? |
| `unwrap(r)` | `result<T,E> -> T` | Extract ok (panics on err) |

### 1.21.8 Channel / Concurrency

| Function | Signature | Description |
|---|---|---|
| `channel()` | `() -> channel<i64>` | New channel |
| `send(ch, v)` | Side effect | Send value |
| `recv(ch)` | `channel<T> -> T` | Receive value |
| `atomic(v)` | `T -> atomic<T>` | New atomic |
| `atomic_load(a)` | `atomic<T> -> T` | Read atomically |
| `atomic_store(a, v)` | Side effect | Write atomically |
| `atomic_add(a, v)` | `(atomic<T>, T) -> T` | Add and return new value |

### 1.21.9 Time

| Function | Signature | Description |
|---|---|---|
| `time_now_ms()` | `() -> i64` | Current time in milliseconds since epoch |
| `sleep_ms(ms)` | `i64 -> i64` | Sleep for `ms` milliseconds |

### 1.21.10 File I/O

| Function | Signature | Description |
|---|---|---|
| `file_read_all(path)` | `str -> result<str, str>` | Read file |
| `file_write_all(path, content)` | `(str, str) -> result<(), str>` | Write file |
| `file_exists(path)` | `str -> bool` | Check existence |
| `file_lines(path)` | `str -> list<str>` | Read lines |

### 1.21.11 Process

| Function | Signature | Description |
|---|---|---|
| `process_args()` | `() -> list<str>` | Command-line arguments |
| `env_var(name)` | `str -> option<str>` | Environment variable |
| `process_exit(code)` | `i64 -> ()` | Exit with code |

### 1.21.12 TCP Networking

| Function | Signature | Description |
|---|---|---|
| `tcp_listen(port)` | `i64 -> i64` | Bind and listen, returns fd |
| `tcp_accept(fd)` | `i64 -> i64` | Accept connection, returns connection fd |
| `tcp_connect(host, port)` | `(str, i64) -> i64` | Connect to server |
| `tcp_read(fd)` | `i64 -> str` | Read a line |
| `tcp_write(fd, data)` | Side effect | Write data |
| `tcp_close(fd)` | Side effect | Close connection |

### 1.21.13 Database (SQLite)

| Function | Signature | Description |
|---|---|---|
| `db_open(path)` | `str -> i64` | Open (or create) a SQLite database, returns handle |
| `db_exec(db, sql)` | `(i64, str) -> i64` | Execute SQL (CREATE/INSERT/UPDATE/DELETE). Returns 0 on success, −1 on error |
| `db_query(db, sql)` | `(i64, str) -> list<list<str>>` | Execute SELECT query. Returns rows of string columns |
| `db_close(db)` | `i64 -> i64` | Close database handle. Returns 0 |

### 1.21.14 Control

| Function | Signature | Description |
|---|---|---|
| `panic(msg)` | `str -> !` | Abort with message |
| `assert(cond)` | `bool -> ()` | Assert (panics if false) |

---

## 1.22 Appendix C: Type System Reference

### 1.22.1 Scalar Types

| Type | Size | Range |
|---|---|---|
| `i64` | 8 bytes | −2^63 to 2^63-1 |
| `i32` | 4 bytes | −2^31 to 2^31-1 |
| `f64` | 8 bytes | IEEE 754 double |
| `f32` | 4 bytes | IEEE 754 single |
| `bool` | 1 byte | `true` or `false` |
| `str` | heap | UTF-8 string |

### 1.22.2 Composite Types

| Type | Syntax | Description |
|---|---|---|
| Array | `[T; N]` | Fixed-size array of N elements of type T |
| Tuple | `(T1, T2, T3)` | Ordered product type |
| List | `list<T>` | Dynamically-sized array |
| Map | `map<K, V>` | Hash map |
| Option | `option<T>` | Nullable value |
| Result | `result<T, E>` | Success or failure |
| Channel | `channel<T>` | Concurrent communication |
| Atomic | `atomic<T>` | Thread-safe scalar |
| Tensor | `tensor<f32, [M, N]>` | N-dimensional array |
| Function | `(T1, T2) -> R` | Function type |

### 1.22.3 Type Casting

Use `expr to Type` to cast between compatible types:

| From | To | Notes |
|---|---|---|
| `i64` | `f64` | Exact for integers up to 2^53 |
| `i64` | `f32` | May lose precision |
| `i64` | `i32` | Truncates if out of range |
| `f64` | `f32` | May lose precision |
| `f32` | `f64` | Always exact |
| `f64` | `i64` | Truncates fractional part |
| `f32` | `i64` | Truncates fractional part |

### 1.22.4 Operator Precedence (highest to lowest)

| Level | Operators |
|---|---|
| 1 | Unary `-`, `!` |
| 2 | `*`, `/`, `%` |
| 3 | `+`, `-` |
| 4 | `<`, `<=`, `>`, `>=` |
| 5 | `==`, `!=` |
| 6 | `&&` |
| 7 | `\|\|` |
| 8 | `to` (cast) |
| 9 | `?` (error propagation, postfix) |

> **Important:** `<` has higher precedence than `+`. Write `(i + 1) < n`, not `i + 1 < n` — the latter parses as `i + (1 < n)`.

---

## 1.23 Appendix D: CLI Reference

### 1.23.1 `iris run <file.iris>`

Compile and run an IRIS source file. Uses the interpreter for quick execution.

```
iris run hello.iris
iris run myapp.iris
```

### 1.23.2 `iris build <file.iris> -o <output>`

Compile to a native binary using LLVM/clang.

```
iris build myapp.iris -o myapp.exe
iris build src/main.iris -o out/myapp.exe
```

If `-o` is omitted, the output is named `iris_out.exe` (Windows) or `iris_out` (Linux/macOS).

### 1.23.3 `iris repl`

Start the interactive REPL.

```
iris repl
```

### 1.23.4 `iris lsp`

Start the Language Server Protocol server (stdio). Used by IDE extensions.

```
iris lsp
```

### 1.23.5 `iris dap`

Start the Debug Adapter Protocol server. Used by IDE debuggers.

```
iris dap
```

### 1.23.6 `iris --emit <kind> <file.iris>`

Compile and emit intermediate output without running.

| Kind | Output |
|---|---|
| `ir` | IRIS IR (SSA text format) |
| `llvm` | LLVM IR text ( `.ll` format) |
| `llvm_complete` | Enhanced LLVM IR with named structs and attributes |
| `jit` | JIT-compile and execute, printing the result and tier used |

```
iris --emit ir myapp.iris
iris --emit llvm myapp.iris > myapp.ll
```

### 1.23.7 `iris --version`

Print the IRIS version.

### 1.23.8 `iris --help`

Print help text.

### 1.23.9 `iris --output <path> <file.iris>`

Write output to a file instead of stdout.

```
iris --emit ir myapp.iris --output myapp.ir
```

---

## 1.24 Appendix E: Compiler Error Reference

This appendix covers the IRIS compiler's diagnostic code system and the most common errors with fixes.

### 1.24.1 Diagnostic Code System

Every IRIS compiler error carries a machine-readable diagnostic code. These codes are shown in your editor alongside the error message and can be used for quick lookup.

| Code | Category | Description |
|---|---|---|
| **E0001** | Parse | Unexpected character in source |
| **E0002** | Parse | Unterminated string literal |
| **E0003** | Parse | Invalid escape sequence in string |
| **E0004** | Parse | Invalid numeric literal |
| **E0005** | Parse | Unexpected token (expected something else) |
| **E0006** | Parse | Unexpected end of file |
| **E0100** | Lower | Undefined variable or function |
| **E0101** | Lower | Type mismatch |
| **E0102** | Lower | Duplicate function definition |
| **E0103** | Lower | Unsupported language feature |
| **E0104** | Lower | Undefined layer (ML models) |
| **E0105** | Lower | Duplicate node in computation graph |
| **E0106** | Lower | Invalid layer parameter |
| **E0107** | Lower | Unknown operation |
| **E0200** | Pass | Use before definition |
| **E0201** | Pass | Multiple definitions of same name |
| **E0202** | Pass | Type error in IR |
| **E0203** | Pass | Missing block terminator |
| **E0204** | Pass | Tensor shape mismatch |
| **E0205** | Pass | Unresolved type inference variable |
| **E0300** | Codegen | Code generation error |
| **E0400** | Interp | Interpreter runtime error |
| **E0500** | I/O | File system or I/O error |

---

### 1.24.2 Common Errors and Fixes

---

### 1.24.3 E1: Missing `else` branch

**Error:**

```
error: if expression requires an else branch
```

**Cause:** You wrote `if cond { ... }` without an `else { ... }`.

**Fix:** Always add `else`. If you just want to do nothing, return a dummy value:

```
// Wrong
def bad(x: i64) -> i64 {
    if x > 0 {
        x
    }
}

// Correct
def good(x: i64) -> i64 {
    if x > 0 {
        x
    } else {
        0
    }
}
```

---

### 1.24.4 E2: Missing semicolon after non-tail statement

**Error:**

```
error: expected expression, found 'val'
```

or unexpected parse errors on the line *after* a function call.

**Cause:** You forgot the `;` after a statement that is not the tail expression.

**Fix:**

```
// Wrong — print() is not the tail, needs semicolon
def bad() -> i64 {
    print("hello")
    42
}

// Correct
def good() -> i64 {
    print("hello");
    42
}
```

---

### 1.24.5 E3: Reassigning an immutable binding

**Error:**

```
error: cannot assign to immutable binding 'x'
```

**Cause:** You used `val` and then tried to reassign it.

**Fix:** Use `var` for bindings you intend to reassign:

```
// Wrong
def bad() -> i64 {
    val x = 0;
    x = x + 1;   // error
    x
}

// Correct
def good() -> i64 {
    var x = 0;
    x = x + 1;
    x
}
```

---

### 1.24.6 E4: Type mismatch in binary operation

**Error:**

```
error: type mismatch in BinOp: left is i64, right is f32
```

**Cause:** You mixed incompatible types in arithmetic. IRIS does not auto-promote.

**Fix:** Cast one operand to match the other:

```
// Wrong
def bad(n: i64, x: f32) -> f32 {
    n + x   // error: i64 + f32 is invalid
}

// Correct
def good(n: i64, x: f32) -> f32 {
    (n to f32) + x
}
```

---

### 1.24.7 E5: Float literal type

**Error:**

```
error: expected f64, found f32
```

**Cause:** `3.14` is `f32`, but the function expects `f64`.

**Fix:** Explicitly cast:

```
// Wrong
def bad() -> f64 {
    3.14    // this is f32, not f64
}

// Correct
```

```
def good() -> f64 {
    3.14 to f64
}
```

---

### 1.24.8 E6: Calling `unwrap` on `none`

**Error (runtime):**

```
panic: unwrap called on None
```

**Cause:** You called `unwrap()` without first checking `is_some()`.

**Fix:** Always guard with `is_some()`:

```
// Wrong (runtime panic if opt is none)
def bad(opt: option<i64>) -> i64 {
    unwrap(opt)
}

// Correct
def good(opt: option<i64>) -> i64 {
    if is_some(opt) {
        unwrap(opt)
    } else {
        0   // default value
    }
}
```

---

### 1.24.9 E7: Operator precedence with comparison

**Error:** Logical, wrong result, no compile error.

**Cause:** `<` has higher precedence than `+`. `i + 1 < n` parses as `i + (1 < n)`, which is `i + 0` or `i + 1` (since `bool` coerces to `0` / `1` in arithmetic context).

**Fix:** Use parentheses:

```
// Likely wrong
while i + 1 < n {
    // ...
}

// Correct — explicit parentheses
while (i + 1) < n {
    // ...
}
```

---

### 1.24.10 E8: `find` result used as number

**Error (type):**

```
error: expected i64, found option<i64>
```

**Cause:** `find(s, sub)` returns `option<i64>`, not `i64`.

**Fix:**

```
// Wrong
def bad(s: str, sub: str) -> bool {
    find(s, sub) >= 0   // error: option<i64> >= i64
}

// Correct
def good(s: str, sub: str) -> bool {
    is_some(find(s, sub))
}
```

---

### 1.24.11 E9: Function not exported

**Error:**

```
error: undefined function 'my_helper'
```

**Cause:** The function in another file is not declared `pub def`.

**Fix:** Add `pub` to the function declaration in the source file:

```
// In mylib.iris
pub def my_helper() -> i64 {   // was: def my_helper
    42
}
```

---

### 1.24.12 E10: Using `%` modulo vs `/` division

**Behavior note:**

IRIS uses `/` for integer division (floor division toward zero) and `%` for modulo (remainder). There is no `//` operator.

```
def examples() -> i64 {
    val a = 10 / 3;    // 3  (integer division)
    val b = 10 % 3;    // 1  (remainder)
    val c = -7 / 2;    // -3 (truncates toward zero)
    val d = -7 % 2;    // -1 (remainder, same sign as dividend)
    a
}
```

---

*End of The IRIS Programming Language: A Complete Guide*

---

**Version**: Corresponds to IRIS compiler version 0.2.x **Platform**: Primarily tested on Windows 10/11 with LLVM 17+ and MinGW ucrt64 **License**: See the IRIS project for licensing terms